SASC 2006

Stream Ciphers Revisited

Workshop Record

Leuven, Belgium

February 2-3, 2006

ECRYPT Network of Excellence in Cryptology

This work has been supported by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Table of Contents

Programme

Thursday, Feb 2nd, 2006

8.15	Registration
9.00	Opening Remarks
	Cryptanalysis I
9.05-9.25	**Cryptanalysis of Pomaranch**
	Carlos Cid, Henri Gilbert and Thomas Johansson
9.25-9.35	**On IV Setup of Pomaranch**
	Mahdi M. Hasanzadeh, Shahram Khazaei and Alexander Kholosha
9.35-9.55	**Pomaranch - Design and Analysis of a Family of Stream Ciphers**
	Tor Helleseth, Cees J.A. Jansen and Alexander Kholosha
10.05-10.25	**Guess-and-Determine Attacks against SOSEMANUK Stream Cipher**
	Yukiyasu Tsunoo, Teruo Saito, Maki Shigeri, Tomoyasu Suzaki, Hadi Ahmadi, Taraneh Eghlidos and Shahram Khazaei
10.30-10.55	***Coffee Break***
	Cryptanalysis II
10.55-11.15	**Resynchronization Attack on WG and LEX**
	Hongjun Wu and Bart Preneel
11.20-11.40	**Chosen Ciphertext Attack on SSS**
	Joan Daemen, Joseph Lano and Bart Preneel
11.45-12.05	**Improved cryptanalysis of Py**
	Paul Crowley
12.10-12.30	**Practical Attacks on one Version of DICING**
	Gilles Piret
12.35-14.00	***Lunch***
	Salons Georges
	SW Performance and Statistical Testing
14.00-14.20	**The eSTREAM Software performance testing**
	Christophe De Cannière
14.25-14.45	**Comparison of 256-bit stream ciphers**
	Daniel J. Bernstein
14.50-15.00	**Statistical Analysis of Synchronous Stream Ciphers**
	Meltem Sonmez Turan, Ali Doganaksoy and Cagdas Calik
15.05-15.25	**d-Monomial Tests are Effective Against Stream Ciphers**
	Markku-Juhani O. Saarinen
15.30-15.55	***Coffee Break***

HW Performance	
15.55-16.05	**Testing Framework for eSTREAM Profile II Candidates**
	L. Batina, S. Kumar, J. Lano, K. Lemke, N. Mentens, C. Paar, B. Preneel, K. Sakiyama and I. Verbauwhede
16.10-16.30	**Hardware Evaluation of eSTREAM Candidates**
	F. Gürkaynak, P. Lüthi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber and W. Fichtner
16.35-16.55	**Review of stream cipher candidates from a low resource hardware perspective**
	Tim Good, William Chelton and Mohammed Benaissa
Public discussion on the performance aspects: 17.00-17.30	
19.00	***Conference Dinner***
	Faculty Club

Friday, Feb 3d, 2006

Cryptanalysis III	
9.00-9.20	**Cryptanalysis of Polar Bear**
	John Mattsson, Mahdi M. Hasanzadeh, Elham Shakour and Shahram Khazaei
9.25-9.45	**Linear Distinguishing Attack on NLS**
	Joo Yeon Cho and Josef Pieprzyk
9.50-10.10	**Cryptanalysis of Grain**
	Come Berbain, Henri Gilbert and Alexander Maximov
10.15-10.35	**Cryptanalysis of Mir-1, a T-function Based Stream Cipher**
	Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo and Maki Shigeri
10.35-11.00	***Coffee Break***
11.00-11.20	**Truncated differential cryptanalysis of five rounds of Salsa20**
	Paul Crowley
Updates on Algorithms I	
11.25-11.45	**A Stream Cipher Construction Inspired by Block Cipher Design Principles**
	Christophe De Cannière and Bart Preneel
11.50-12.10	**On periods of Edon-(2m,2k) Family of Stream Ciphers**
	Danilo Gligoroski, Smile Markovski and Svein Johan

	Knapskog
12.15-14.00	***Lunch***
	Salons Georges
Updates on Algorithms II	
14.00-14.20	**CryptMT: effect of huge prime period and multiplicative filter, and a tweak on faster initialization.**
	Makoto Matsumoto, Mutsuo Saito, Takuji Nishimura and Mariko Hagita
14.25-14.35	**T-function based streamcipher TSC-4**
	Dukjae Moon, Daesung Kwon, Daewan Han, Jooyoung Lee, Gwon Ho Ryu, Dong Wook Lee, Yongjin Yeom and Seongtaek Chee
14.40-14.50	**Update on F-FCSR Stream Cipher**
	Francois Arnault, Thierry Berger and Cédric Lauradoux
14.55-15.05	**Security and Implementation Properties of ABC v.2**
	Vladimir Anashin, Andrey Bogdanov and Ilya Kizhvatov
15.10-15.20	**DecimV2**
	Come Berbain et al
15.25-15.35	**Status of Achterbahn and Tweaks**
	Berndt M. Gammel, Rainer Goettfert and Oliver Kniffler
15.35-16.00	***Coffee Break***
Rump Session and Open Discussion : 16.00-17.30	

Program Committee

Program Chair : Anne Canteaut, INRIA, France

Members:

- Steve Babbage, Vodafone, UK
- Carlos Cid, Royal Holloway, University of London, U.K.
- Nicolas Courtois, Axalto Smart Cards Crypto Research, France
- Henri Gilbert, France Telecom R&D, France
- Thomas Johansson, Lund University, Sweden
- Joseph Lano, Katholieke Universiteit Leuven, Belgium
- Christof Paar, Ruhr-University of Bochum, Germany
- Matthew Parker, University of Bergen, Norway
- Bart Preneel, Katholieke Universiteit Leuven, Belgium
- Matt Robshaw, France Telecom R&D, France

Organising Committee

Organising Chair : Joseph Lano

Members:

- Thomas Herlea
- Ozgul Kucuk
- Pela Noe
- Panagiotis Rizomiliotis
- Elvira Wouters

Cryptanalysis of Pomaranch

Carlos Cid[1], Henri Gilbert[2] and Thomas Johansson[3]

[1] Information Security Group,
Royal Holloway, University of London
Egham, Surrey TW20 0EX, United Kingdom
carlos.cid@rhul.ac.uk
[2] France Télécom, R&D Division
38–40, rue du Général Leclerc
92794 Issy les Moulineaux, Cedex 9, France
henri.gilbert@francetelecom.com
[3] Dept. of Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden
thomas@it.lth.se

Abstract Pomaranch [3] is a synchronous stream cipher submitted to eSTREAM, the ECRYPT Stream Cipher Project. The cipher is constructed as a cascade clock control sequence generator, which is based on the notion of jump registers. In this paper we present an attack which exploits the cipher's initialization procedure to recover the 128-bit secret key. The attack requires around 2^{65} computations. An improved version of the attack is also presented, with complexity 2^{52}.
Keywords: Pomaranch Stream Cipher, Jump Registers, Chosen IV Attack.

1 Introduction

Pomaranch[1] is one of the 34 stream ciphers submitted to eSTREAM, the ECRYPT Stream Cipher Project [1]. The cipher is implemented as a binary one clock pulse cascade clock control sequence generator, and uses 128-bit keys and IVs of length between 64 and 112 bits [3]. The construction is based on the notion of *jump registers*.

Jump controlled LFSRs were introduced in [2] as alternative to traditional clock-controlled registers. In jump controlled LFSRs, the registers are able to move to a state that is more than one step ahead without having to step through all the intermediate states (thus the name jump registers). The main motivation for the proposal of jump registers is to construct LFSR-based ciphers that can be efficiently protected against side-channel attacks while preserving the advantages of irregular clocking.

2 Outline of Pomaranch

Pomaranch is depicted in Figure 1, where only the key stream generation phase is represented (called Key Stream Generation Mode). The cipher consists of nine cascaded jump registers R_1 to R_9. The jump registers are implemented as autonomous

[1] The cipher is also referred in the specification document [3] as Cascade Jump Controlled Sequence Generator (CJCSG).

Linear Finite State Machine (LFSM), built on 14 memory cells, which behave either as simple delay shift cells or feedback cells, depending on the value of the so-called Jump Control (JC) signal. At any moment, half of the cells in the registers are shift cells, while the other half are feedback cells. The initial configuration of cells is determined by the LFSM transition matrix A, and is used if the JC value is zero. If JC is one, all cells are switched to the opposite mode. This is equivalent to switching the transition matrix to $(A + 1)$ [3].

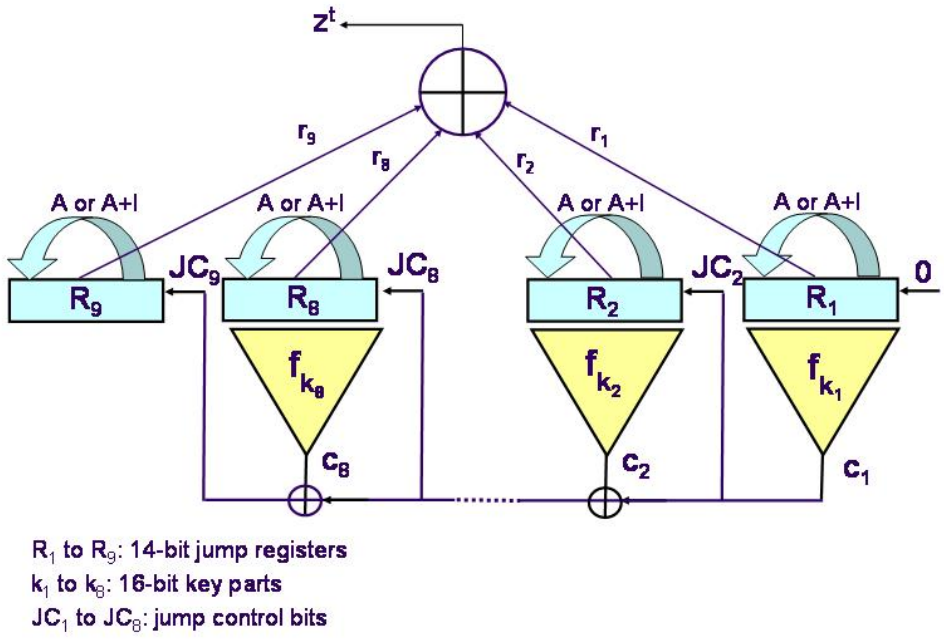


Figure1. The Pomaranch stream cipher

The 128-bit key K is divided into eight 16-bit subkeys k_1 to k_8. At time t, the current state of the registers R_1^t to R_8^t are non-linearly filtered, using a function that involves the corresponding subkey k_i. These functions provide as output eight bits c_1^t to c_8^t, which are used to produce the jump control bits JC_2^t to JC_9^t controlling the registers R_2 to R_9 at time t, as following:

$$JC_i^t = c_1^t \oplus \ldots \oplus c_{i-1}^t \quad \text{for } i = 2, \ldots, 9.$$

The jump control bit JC_1 of register R_1 is permanently set to zero. The key stream bit z^t produced at time t is the XOR of nine bits r_1^t to r_9^t selected at fixed positions of the current register states R_1^t to R_9^t.

Key and IV Loading. During the cipher initialization, the content of registers R_1 to R_9 are first set to non-zero constant 14-bit values derived from π, then the

subkeys k_i are loaded and the registers are run for 128 steps in a special mode (called Shift Mode). The main difference between the Key Stream Generation Mode and the Shift Mode is that, in the latter the output of the filtering function of register R_i (denoted by c_i) is added to the feedback of register R_{i+1}, with the tap from cell 1 in the register R_9 being added to the register R_1, making then what can be seen as a "big loop". Note that the configuration of the jump registers do not change in this mode (they all operate as if $JC_i = 0$). This process ensures that the states of the registers R_1 and R_9 after this key loading phase depend upon the entire key K. We denote these states by R_1^K to R_9^K.

Next the IV is loaded into the registers. The IV can have any arbitrary length between 64 and 112 bits. If the IV length is shorter than 112 bits, it is expanded by cyclically repeating it until a length of exactly 112 bits is obtained. This new string is then loaded into the registers as described below. In the remaining of this paper, for the sake of simplicity, we assume that the IV length is exactly 112 bits.

The IV is loaded into the registers in the following manner: the 112-bit IV is split into eight 14-bit parts IV_1 to IV_8, which are XORed with the 14-bit states of registers R_1^K to R_8^K obtained at the end of the key loading. If any of the resulting states consists of 14 null bits, its lowest weight bit is set to one (this ensures that no state will be made up entirely of null bits[2]). The resulting register states R_1 to R_8 form together with R_9^K the nine initial states. We denote these resulting 14-bit state values by R_1^{-128} to R_9^{-128}. The key stream generation mode of Figure 1 is now activated, and the runup consists of 128 steps in which the produced key stream bits are discarded.

3 Description of the Attack

We have identified the following weakness in the Pomaranch IV initialization procedure: if for a given key K and IV value IV, we only modify the IV part IV_8 and keep the remaining parts IV_1 to IV_7 unchanged (thus obtaining a modified IV value IV'), on comparing the key stream generation under the key K with IV and IV', we have that for every $t \geq -128$

$$R_i^t(IV) = R_i^t(IV') \quad \text{for } i = 1, \ldots, 7 \ .$$

In other words, the Key and IV loading procedure does not diffuse all IV bits into the whole state of the generator. Consequently, if IV and IV' are chosen as above, the contributions from registers R_1 to R_7 cancel out on each key stream XOR $z^t(IV) \oplus z^t(IV')$, and we obtain the relation

$$z^t(IV) \oplus z^t(IV') = r_8^t(IV) \oplus r_8^t(IV') \oplus r_9^t(IV) \oplus r_9^t(IV').$$

[2] The Pomaranch specification does not mention this feature, which is described in the source code provided with the submission and has been confirmed by one of the designers [4]. We will show in the next section that, although the cipher can be attacked even if this feature is withdrawn, this represents an additional weakness that leads to improved attacks.

We now show how to exploit this weakness to recover the subkey k_8 of an unknown key K, in a chosen IV attack. Consider 3 distinct chosen IV values IV, IV' and IV'', which only differ by their part IV_8, IV'_8 and IV''_8. We can obtain the corresponding first m-bit key stream $z^t(IV)_{t=0\ \mathtt{to}\ m-1}$, $z^t(IV')_{t=0\ \mathtt{to}\ m-1}$, and $z^t(IV'')_{t=0\ \mathtt{to}\ m-1}$, which in turn provide the pairwise XOR values

$$\begin{aligned}\delta^t &= z^t(IV) \oplus z^t(IV')_{t=0\ \mathtt{to}\ m-1},\\ \delta'^t &= z^t(IV') \oplus z^t(IV'')_{t=0\ \mathtt{to}\ m-1},\end{aligned}$$

In order to recover the value of k_8, we guess the following values:

- Subkey k_8:	16 bits;
- Registers R_8^K and R_9^K:	28 bits;
- $n_8 = \sharp\{t \in \{-128, .., -1\} \mid JC_8^t(IV) = 1\}$:	129 possible values;
- $n_9 = \sharp\{t \in \{-128, .., -1\} \mid JC_9^t(IV) = 1\}$:	129 possible values;
- $n'_9 = \sharp\{t \in \{-128, .., -1\} \mid JC_9^t(IV') = 1\}$:	129 possible values;
- $n''_9 = \sharp\{t \in \{-128, .., -1\} \mid JC_9^t(IV'') = 1\}$:	129 possible values.

The attack exploits the jump registers property that since the transition matrices A and $A + I$ commute, the transition matrix associated with a number s of steps can only take one of the at most $s + 1$ values $A^p(A + I)^q$, with $p + q = s$. Due to this property, the knowledge of the values of (n_8, n_9, n'_9, n''_9) is sufficient to derive the R_8 and R_9 transition matrices of the form $A^{128-n}(A + I)^n$ associated with the 128-step runup for IV values IV, IV' and IV''. Note that although n_8, n_9, n'_9, n''_9 can take any of the 129 values in the $[0 \cdots 128]$ interval, their values are binomially distributed, so that in practice the $2^5 - 1$ middle values in the interval $[49 \cdots 79]$ have an overwhelming occurrence probability.

Now since we have[3]

$$\begin{aligned}&R_8^{-128}(IV) = R_8^K \oplus IV,\\ &R_8^{-128}(IV') = R_8^K \oplus IV',\\ &R_8^{-128}(IV'') = R_8^K \oplus IV'',\\ &R_9^{-128}(IV) = R_9^{-128}(IV') = R_9^{-128}(IV'') = R_9^K,\end{aligned}$$

it follows that knowledge of R_8^K, R_9^K, n_8, n_9, n'_9 and n''_9 allows us to compute $R_8^0(IV)$, $R_8^0(IV')$, $R_8^0(IV'')$, $R_9^0(IV)$, $R_9^0(IV')$, and $R_9^0(IV'')$.

To test a $(k_8, R_8^K, R_9^K, n_8, n_9, n'_9, n''_9)$ assumption we need to compute the resulting values of $R_8^0(IV)$, $R_8^0(IV')$, $R_8^0(IV'')$, $R_9^0(IV)$, $R_9^0(IV')$, and $R_9^0(IV'')$ and iteratively try, for consecutive values of m, to guess the m-bit value $JC_8^t(IV)_{t=0\ \mathtt{to}\ m-1}$ in order to derive the resulting values of $R_8^t(IV)$, $R_8^t(IV')$, $R_8^t(IV'')$, $R_9^t(IV)$, $R_9^t(IV')$, and $R_9^t(IV'')$. Following we verify whether the predicted values $(\delta^t, \delta'^t)_{t=0\ \mathtt{to}\ m-1}$ are in agreement with the observed ones. The average number of m values to be tested until a wrong assumption is discarded (because no $JC_8^t(IV)_{t=0\ \mathtt{to}\ m-1}$ m-tuple fits the observed values) is about 2.

[3] We are ignoring the cipher's non-zero state forcing feature at this stage.

Indeed, for a certain (k_8, R_8^K, R_9^K, n_8, n_9, n'_9, n''_9) assumption and a choice of $JC_8^t(IV)$, the pair (δ^t, δ'^t) can take one of four possible values. Assuming the values are randomly generated, there are three events to consider. First the case in which the pairs (δ^t, δ'^t) for both the choices of $JC_8^t(IV) = 0$ and $JC_8^t(IV) = 1$ are in agreement with the observed value. Its probability is 1/16, and it leaves us with two possible configurations that need to be further tested. The second event is when only one pair (δ^t, δ'^t) for either the choices of $JC_8^t(IV) = 0$ or $JC_8^t(IV) = 1$ is in agreement with the observed one. Its probability is 3/8, and it leaves us with one possible configuration that need to be further tested. The third event is when neither the pairs (δ^t, δ'^t) for the choices of $JC_8^t(IV) = 0$ and $JC_8^t(IV) = 1$ is in agreement with the observed one (i.e. the configuration is inconsistent). Its probability is 9/16, and no further tests using this configuration is necessary. Thus if X denotes the number of tests we need to perform, then

$$E(X) = 1 + \frac{1}{16} \cdot 2 \cdot E(X) + \frac{3}{8} \cdot 1 \cdot E(X) + \frac{9}{16} \cdot 0 \cdot E(X),$$

and $E(X) = 2$.

The attack described above allows us to recover the value of k_8. Its complexity is bounded over by $2^{16} \times 2^{28} \times (2^5)^4 \times 2 = 2^{65}$. Note that the attack also recovers the correct values for R_8^K and R_9^K. To recover the other key parts, we can proceed as following: repeat the same attack for another value of (IV, IV', IV''), call it $(\overline{IV}, \overline{IV'}, \overline{IV''})$, such that IV and $\overline{IV}$ only differ by their part IV_7 and $\overline{IV_7}$. Since we know already k_8, R_8^K and R_9^K, this second attack can be mounted much faster. Finally, we can guess the values of R_7^K and n_7 and check whether there exists a sequence $JC_7^t(IV)_{t=0 \texttt{ to } m-1}$ that is consistent with the already known sequences $JC_8^t(IV)_{t=0 \texttt{ to } m-1}$ and $JC_8^t(\overline{IV})_{t=0 \texttt{ to } m-1}$. This can be done for all the remaining key parts, until the entire key K has been recover. The complexity of the entire attack remains about 2^{65}.

Improved Attack. Note that so far we have not exploited the non-zero state forcing feature of Pomaranch, and the above attack works whether this feature is present or not. We now show that this feature results in a low complexity distinguisher, and also allows us to reduce the complexity of the key derivation procedure described above.

The distinguisher works as following: given an unknown key K, we can try the 2^{14} possible IV values obtained by keeping (say) IV_1 to IV_7 unchanged and taking all possible values for (say) IV_8. Now two of these 2^{14} IVs result in exactly the same states R_1^{-128} to R_9^{-128} after key and IV loading, namely the IV value resulting on a 14-bit R_8 state equal to zero (which will have one bit switched to 1 by the cipher non-zero state forcing procedure), and the IV value derived from the former one by swapping the same bit position. The key streams for these two IV values are exactly the same. If the key stream is sufficient long (e.g. more than 27 bits in order for collisions of a pair of IV values to be unlikely), this provides an efficient chosen IV

distinguisher of distinguishing probability close to 1, requiring generation of only 2^{14} key stream sequences of length (say) 64 bits each.

This distinguisher can be used to improve the key derivation attack described above. Indeed, the distinguisher allows us to recover the register value R_8^K up to one single bit, so that a factor of 2^{13} can be saved in the search of (k_8, R_8^K, R_9^K, n_8, n_9,n'_9, n''_9), and the attack complexity is reduced to 2^{52}.

4 Conclusion

We showed in this paper how to mount a chosen IV attack to recover the secret key of Pomaranch with complexity much lower than the one expected with 128-bit keys. The attack exploits a weakness in the cipher initialization procedure, namely the process does not diffuse all the IV bits into the whole state of the key stream generator. By exploiting another feature of the IV loading, we were able to substantially improve the attack.

References

1. eSTREAM, the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`.
2. C. J. Jansen. Streamcipher Design: Make your LFSRs jump! In *SASC, Workshop Record, ECRYPT Network of Excellence in Cryptology*, pages 94–108, 2004.
3. C.J. Jansen, T. Helleseth, and A. Kholosha. Cascade Jump Controlled Sequence Generator (CJCSG). In *SKEW, Workshop Record, ECRYPT Network of Excellence in Cryptology*, 2005.
4. A. Kholosha. Personal Communication.

On IV Setup of Pomaranch

Mahdi M. Hasanzadeh[†] Shahram Khazaei[†] Alexander Kholosha[‡]

[†] Zaeim Electronic Industries Company, P.O. BOX 14155-1434, Tehran, Iran
[‡] The Selmer Center, University of Bergen, P.O. Box 7800, 5020, Bergen, Norway
{Hasanzadeh, khazaei}@Zaeim.com, Alexander.Kholosha@ii.uib.no

Abstract. Pomaranch is a synchronous bit-oriented stream cipher submitted to eSTREAM, the ECRYPT Stream Cipher Project. Following the recently published chosen IV [1] and correlation [7] key-recovery attacks, the authors changed the configuration of jump registers and introduced two new key-IV setup procedures for the cipher. We call the updated version as Tweaked Pomaranch vs. Original Pomaranch [4]. In this paper we use the findings of [7] to mount a chosen IV key-recovery attack on the Original Pomaranch with computational complexity of $O(2^{73.5})$. The attack is also applicable to the first key-IV setup proposal for Tweaked Pomaranch with computational complexity of $O(2^{117.7})$. The alternative key-IV setup for Tweaked Pomaranch is immune against our attack. Both versions of Pomaranch deal with 128 bit keys.

Keywords. ECRYPT Stream Cipher Project, Pomaranch, CJCSG, Jump Register, Cryptanalysis, Linear Equivalence Bias, Clock-Controlled LFSR, Security Evaluation.

1 Introduction

Pomaranch (also known as a Cascade Jump Controlled Sequence Generator or CJCSG) [4] is a synchronous bit-oriented stream cipher, one of the ECRYPT Stream Cipher Project [2] candidates. It uses 128-bit keys and in its original design - which we call Original Pomaranch - accommodates an Initial Value (IV) of 64 up to 112 bits long. The algorithm uses a one-clock-pulse cascade construction of so called jump registers [3] being essentially linear finite state machines with a special transition matrix. Moreover, the characteristic polynomial of the transition matrix was made to be primitive and satisfying additional constraints that arise from the need to use the register in a cascade jump control setup. The principal advantage of jump registers over the classical clock-controlled arrangements is their ability to move a Linear Feedback Shift Register (LFSR) to a state that is more than one step ahead but without having to step through all the intermediate states. The transition matrix of the jump registers in Pomaranch has been chosen so to secure the design against side-channel attacks while preserving all the advantages of irregular clocking.

Following the recently published chosen IV [1] and correlation [7] key-recovery attacks, the authors made some tweaks on the cipher. Firstly, they changed the configuration of jump registers and then introduced two different key-IV setup procedures for the cipher - one mixes the IV and key similarly to Original Pomaranch limiting the IV length to 78 bits and the other is totally different from the original version and can accommodate IV's up to 126 bits long [6]. These changes effectively counter the attacks introduced in [7, 1]. We call this updated version as Tweaked Pomaranch.

Paper [7] describes a new inherent property of jump registers that allows constructing their linear equivalences. This property was further investigated in [5]. In this paper we use the same idea to mount a resynchronization attack (IV attack) on Original Pomaranch and the first key-IV setup of Tweaked Pomaranch. The second key-IV setup of Tweaked Pomaranch is immune against our attack. In the rest of the paper we just consider Tweaked Pomaranch with the first key-IV setup and refer to Tweaked Pomaranch for convenience.

Our results show that the key of both Original and Tweaked Pomaranch can be found when a key is used with about 2^{35} chosen IV's. The required computational complexities are $O(2^{73.5})$ and $O(2^{117.7})$ for Original and Tweaked Pomaranch respectively. There are also many tradeoffs between the number of IV's and the required bit-stream from each IV.

2 Outline of Original and Tweaked Pomaranch

The key-stream generator of Pomaranch is depicted in Figure 1. The cipher consists of nine cascaded JR denoted by R_1 to R_9. Each JR is built on 14 memory cells which behave either as simple delay shift cells or feedback cells, depending on the value of JC sequence. At any moment, half of the cells in the registers are shift cells, while the other half is feedback cells. The initial configuration of cells is determined by the transition matrix A, and is used if the JC value is zero. If JC is one, all cells are switched to the opposite mode. This is equivalent to switching the transition matrix to $(A + I)$ [4].

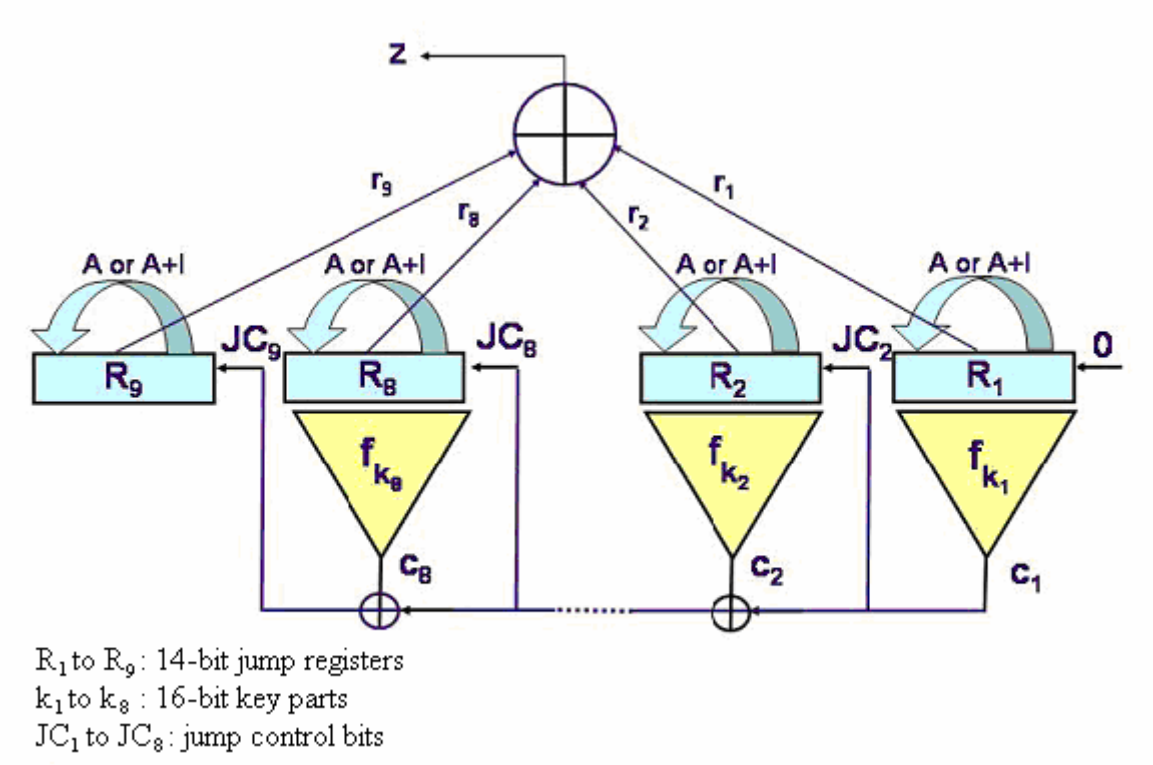


Figure 1. Schematic of the Pomaranch

The 128-bit key K is divided into eight 16-bit sub keys k_1 to k_8. At time t, the current states of the registers R_1^t to R_8^t are non-linearly filtered, using a function that involves the corresponding sub key k_i. These functions provide as output eight bits c_1^t to c_8^t, which are used to produce the jump control bits JC_1^t to JC_8^t controlling the registers R_2 to R_9 at time t, as following:

$$JC_i^t = c_1^t \oplus \cdots \oplus c_{i-1}^t, \qquad 2 \le i \le 9. \tag{1}$$

The jump control bit JC_1 of register R_1 is permanently set to zero. The key-stream bit z^t produced at time t is the XOR of nine bits r_1^t to r_9^t selected at second position of the registers R_1 to R_9, that is $z^t = r_1^t \oplus \cdots \oplus r_9^t$.

The only difference between the key-stream generator of Original and Tweaked Pomaranch is the configuration of the jump registers or equivalently the A matrix.

Key-IV Setup of Original Pomaranch [4]: During the cipher initialization, the content of registers R_1 to R_9 is first set to non-zero constant 14-bit values derived from π, then the sub keys k_i are loaded and the registers are run for 128 steps in a special mode (called Shift Mode). The main difference between the Key-Stream Generation Mode and the Shift Mode is that, in the latter the output of the filter function of

register R_i (denoted by c_i) is added to the feedback of register R_{i+1}, with the tap from cell 1 in the register R_9 being added to the register R_1, making then what can be seen as a "big loop". Note that the configuration of the jump registers does not change in this mode (they all operate as if $JC_i = 0$). This process ensures that the states of the registers R_1 to R_9 after this key loading phase depend upon the entire key K. We denote these states by $R_1(K)$ to $R_9(K)$.

Next the IV is loaded into the registers. The IV can have any arbitrary length between 64 and 112 bits. First, the IV is expanded by cyclically repeating it until a length of exactly 126 (= 9×14) bits is obtained. This new string is then split into nine 14-bit parts, denoted by IV_1 to IV_9, which are XORed with the 14-bit states of registers $R_1(K)$ to $R_9(K)$ obtained at the end of the key loading. If any of the resulting states consists of 14 null bits, its least significant bit is set to one (this ensures that no state will be made up entirely of null bits). The resulting register states R_1 to R_9 form the nine initial states. The key-stream generation mode showed in Figure 1 is now activated, and the run-up consists of 128 steps in which the produced key-stream bits are discarded.

Key-IV Setup of Tweaked Pomaranch [6]: Following the recently published chosen IV attack [1], the authors introduced two different tweaks in key-IV setup of the cipher. In the first version, the length of IV is limited to 78 (= 6×13) bits; all IV's are expanded by cyclically repeating IV-bits until a length of exactly 117 (= 9×13) bits is obtained. First, the key K is loaded into the registers the same way as in the original version. Then for IV loading, the IV-bits are split into groups of 13 bits denoted by IV_i , $1 \le i \le 9$. These 13 bit IV-values are XORed with the 13 most significant bits of the registers R_i, that is $R_i(K)$, $1 \le i \le 9$. Now all registers are checked for the all-zero state and if all-zero the least significant bit of the register is set to one.

The second proposed version for key-IV setup is totally different from the old version and uses IV's up to 126 bits length. Since our attack is just applicable on the first version of the newly proposed key-IV setup, we skip the description of this alternative and refer the reader to [6]. Both versions of key-IV setup effectively counter the chosen IV attack introduced in [1]. Note a slight difference between what the authors of [1] considered in their paper as the IV loading procedure and what is in Original Pomaranch. However, this modification does not affect their attack.

3 Description of the Attack

In [7, 5] it has been shown that there are certain linear relations in the output sequence of a Jump Register Section which hold with a fixed bias. Define the correlation coefficient of a binary random variable x as $\varepsilon = 1 - 2\Pr\{x = 1\}$. In particular, for JR's of Original Pomaranch the correlation coefficient of the linear relation $r^t \oplus r^{t+8} \oplus r^{t+14}$ is equal to $\varepsilon = 840/2^{14}$ provided that the JC sequence is purely random [7]. This value was called the Linear Equivalent Bias (LEB) in [5]. In [7] using this bias a correlation based key-recovery attack mounted on Original Pomaranch which has computational complexity of $O(2^{95.4})$ and requires $2^{71.8}$ bits of the key-stream generated using a single key and IV pair. In this section we explain how to improve this attack using different IV's.

3.1 Application to the Original Pomaranch

Suppose that we are given the first T bits of the Pomaranch key-stream generated from an unknown fixed key and $l+1$ known random IV's whose first part corresponding to R_1 (14 bits in Original Pomaranch and 13 bits in Tweaked Pomaranch) are the same. Let us denote the IV's by IV^i ($0 \le i \le l$) and the output sequence corresponding to IV^i by $\{z^t(i)\}_{t=0}^{\infty}$.

We also denote the output sequence of the n^{th} register by $\{r_n^t(i)\}_{t=0}^{\infty}$ when IV^i is used, thus $z^t(i) = r_1^t(i) \oplus \cdots \oplus r_9^t(i)$. Let introduce the following sequences:

$$e_n^t(i) = r_n{}^t(i) \oplus r_n{}^{t+8}(i) \oplus r_n{}^{t+14}(i),\ 0 \le i \le l,\ 2 \le n \le 9 \quad \textbf{(2)}$$

$$u_n^t(i) = e_{10-n}^t(i) \oplus \cdots \oplus e_9^t(i),\ 0 \le i \le l,\ 1 \le n \le 8 \quad \textbf{(3)}$$

$$Z^t(i) = z^t(i) \oplus z^{t+8}(i) \oplus z^{t+14}(i),\ 0 \le i \le l. \quad \textbf{(4)}$$

Using this notation the following relation holds for every $0 \le i \le l$:

$$Z^t(i) = r_1^t(i) \oplus r_1^{t+8}(i) \oplus r_1^{t+14}(i) \oplus u_8^t(i). \quad \textbf{(5)}$$

Since the correlation coefficient of the sequence $e_n^t(i)$, $2 \le n \le 9$, is equal to $\varepsilon = 840/2^{14}$, the correlation coefficient of the sequence $u_n^t(i)$, $1 \le n \le 8$, is equal to ε^n under the independence assumption of $e_n^t(i)$ sequences, $2 \le n \le 9$, for every $0 \le i \le l$.

In [7] the equation (5) has been used in a correlation attack to recover the initial sate of R_1 using a single IV (the assumption of using just one IV has been implicitly used). The required key-stream length and computational complexity are $N_0 = 14/C(0.5(1-\varepsilon^8)) \approx 2^{72.8}$ and $2^{14}N_0 \approx 2^{86.8}$ respectively (see [7] for details).

The main contribution of this paper is to increase first the correlation coefficient of $u_8^t(i)$ for a fixed value of *i*, i.e. $i = 0$ and then apply correlation attack. This method will considerably improve the attack. The idea of increasing the correlation coefficient of $u_8^t(0)$ is based on trying to estimate it using the following group of relations

$$Z^t(0) \oplus Z^t(i) = r_1^t(0) \oplus r_1^{t+8}(0) \oplus r_1^{t+14}(0) \oplus r_1^t(i) \oplus r_1^{t+8}(i) \oplus r_1^{t+14}(i) \oplus u_8^t(0) \oplus u_8^t(i). \quad \textbf{(6)}$$

Since the first part of IV's ($IV^i, 0 \le i \le l$) are the same, we have $r_1^t(0) \oplus r_1^t(i) = 0$. Therefore, the relation (6) can be rewritten as

$$\Delta(i) = u_8^t(0) \oplus u_8^t(i),\ 1 \le i \le l, \quad \textbf{(7)}$$

where $\Delta(i) = Z^t(0) \oplus Z^t(i)$ is completely known.

The ML estimation of $u_8^t(0)$ denoted by $\hat{u}_8^t(0)$ is achieved by comparing $\sum_{i=1}^{l} \Delta(i)$ with the threshold *l*/2. That is, we decide on $\hat{u}_8^t(0) = 0$, if $\sum_{i=1}^{l} \Delta(i) < l/2$ and on $\hat{u}_8^t(0) = 1$ otherwise. The error probability of this estimation is approximately equal to $Q(\sqrt{l}\varepsilon^8)$, where $Q(x) = \int_x^{\infty} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$. The variable $u_8^t(0)$ can be related to its estimation, $\hat{u}_8^t(0)$, by the relation $u_8^t(0) = \hat{u}_8^t(0) \oplus w_8^t(0)$ where $w_8^t(0)$ is the estimation error whose correlation coefficient is equal to $\varepsilon' = 1 - 2Q(\sqrt{l}\varepsilon^8)$.

Using this estimation, the relation (5) for $i = 0$ turns into

$$Z^t(0) = r_1^t(0) \oplus r_1^{t+8}(0) \oplus r_1^{t+14}(0) \oplus \hat{u}_8^t(0) \oplus w_8^t(0). \quad \textbf{(8)}$$

Now the equation (8) can be used in a correlation attack to recover the initial state of R_1 for IV^0. The required key-stream length and computational complexity are $T = 14/C(0.5(1-\varepsilon'))$ and $2^{14}T$ respectively (see [7] for details).

Since in the first phase we must estimate $u_8^t(0)$ for $0 \le t \le T-1$ using *l* different IV's, the required computational complexity of this phase is *Tl* resulting in a total computational complexity of $C = T(l + 2^{14})$ for the initial state recovery of R_1.

For every value of l between 2^{18} and 2^{64}, the minimum amount of the computational complexity is obtained which is equal to $C = 2^{73.5}$. The required key-stream length from each IV is equal to $T = 2^{73.5}/l$, where an attacker can choose the parameter l on his/her fitness.

After finding the initial state of R_1, we can eliminate the portion of $r_1^t(i)$ from the output sequence of Pomaranch for each IV. Define the sequence $z_1^t(i)$ as an XOR of $z^t(i)$ and $r_1^t(i)$ which is now available. Then similarly to (5) we have

$$Z_1^t(i) = r_2^t(i) \oplus r_2^{t+8}(i) \oplus r_2^{t+14}(i) \oplus u_7^t(i), \tag{9}$$

where

$$Z_1^t(i) = z_1^t(i) \oplus z_1^{t+8}(i) \oplus z_1^{t+14}(i). \tag{10}$$

The sequence $r_2^t(i) \oplus r_2^{t+8}(i) \oplus r_2^{t+14}(i)$ can be generated if we know both the 14-bit initial state of R_2 and 16-bit sub-key k_1 (totally 30 bits). In [7] the equation (9) has been used in a correlation attack to recover these 30 bits using a single IV. The required key-stream length and computational complexity are $N_0 = 30/C(0.5(1-\varepsilon^7)) \approx 2^{65.4}$ and $2^{30} N_0 \approx 2^{95.4}$ respectively (see [7] for details).

Again we can increase the correlation coefficient of $u_7^t(i)$ for a fixed value of i, i.e. $i = 0$, and then apply correlation attack. The following group of relations

$$Z_1^t(0) \oplus Z_1^t(i) = r_2^t(0) \oplus r_2^{t+8}(0) \oplus r_2^{t+14}(0) \oplus r_2^t(i) \oplus r_2^{t+8}(i) \oplus r_2^{t+14}(i) \oplus u_7^t(0) \oplus u_7^t(i) \tag{11}$$

can be used to estimate $u_7^t(0)$ similarly to (6). In the first part we assumed that the IV's IV^i are the same in the first part. Here, we must force the IV's IV^i to be the same in the first two parts. Under this condition we have $r_2^t(0) \oplus r_2^t(i) = 0$. Therefore we can compute an estimation of $u_7^t(0)$ denoted by $\hat{u}_7^t(0)$ where $u_7^t(0) = \hat{u}_7^t(0) \oplus w_7^t(0)$ and $w_7^t(0)$ is the estimation error whose correlation coefficient is equal to $\varepsilon'' = 1 - 2Q(\sqrt{l}\varepsilon^7)$. Using this estimation, the relation (9) for $i = 0$ turns into

$$Z_1^t(0) = r_2^t(0) \oplus r_2^{t+8}(0) \oplus r_2^{t+14}(0) \oplus \hat{u}_7^t(0) \oplus w_7^t(0). \tag{12}$$

Now the equation (12) can be used in a correlation attack to recover the initial state of R_2 for IV^0 and key segment k_1. The required key-stream length and computational complexity are $T = 30/C(0.5(1-\varepsilon''))$ and $2^{30}T$ respectively (see [7] for details). The total computational complexity of initial state recovery of R_2 and key segment k_1 is equal to $C = T(l + 2^{30})$.

For every value of l between 2^{35} and 2^{55}, the minimum amount of the computational complexity is obtained which is equal to $C = 2^{66}$. The required key-stream length from each IV is equal to $T = 2^{66}/l$, where an attacker can choose the parameter l on his/her fitness. Similarly these parameters can be computed for other registers and key parts. These parameters are summarized in Table 1 for the initial state recovery of R_1 to R_5 and key segments k_1 to k_4.

Table 1. Different parameters of finding different sections of Original Pomaranch

Recovered Sections	l	T	Complexity	Number of fixed part of IV's
R_1	$2^{18} \le l \le 2^{64}$	$2^{73.5}/l$	$2^{73.5}$	1
R_2, k_1	$2^{35} \le l \le 2^{55}$	$2^{66}/l$	2^{66}	2
R_3, k_2	$2^{31} \le l \le 2^{51}$	$2^{57.5}/l$	$2^{57.5}$	3
R_4, k_3	$2^{30} \le l \le 2^{35}$	$2^{41}/l$	2^{41}	4
R_5, k_4	$l = 2^{28}$	$2^{5.3}$	$2^{35.8}$	5

After finding key parts k_1 to k_4, the rest part of the key can be found by exhaustive search with computational complexity $O(2^{64})$. Therefore the total computational complexity of our key-recovery attack is

$O(2^{73.5})$, and the required number of IV's, the imposed condition on IV's and the required number of key-stream bits from each IV are determined by Table 1 which provides many tradeoffs.

3.2 Application to Tweaked Pomaranch

Following the recently published key-recovery attack [7], the authors changed the configuration of jump registers. The Linear Equivalent Bias (LEB) of new configuration of jump registers is equal to $\varepsilon = 124/2^{14}$ [5] which effectively counters the attack introduced in [7]. For JR's of Tweaked Pomaranch the LEB value is held for the linear relation $r^t \oplus r^{t+5} \oplus r^{t+6} \oplus r^{t+10} \oplus r^{t+14}$. Although the change in configuration counters the attack in [7], the chosen IV attack introduced in section 3.1 is still applicable to the first key-IV setup of Tweaked Pomaranch. A similar procedure to what explained in Section 3.1 leads to the following numbers.

Table 2. Different parameters of finding different sections of Tweaked Pomaranch

Recovered Sections	l	T	Complexity	Number of fixed part of IV's
R_1	$2^{28} \le l \le 2^{78}$	$2^{117.7}/l$	$2^{117.7}$	1
R_2, k_1	$2^{35} \le l \le 2^{52}$	$2^{104.7}/l$	$2^{104.7}$	2

After finding key part k_1, the rest part of the key can be found by exhaustive search with computational complexity $O(2^{112})$. Therefore the total computational complexity of our key-recovery attack is $O(2^{117.7})$, and the required number of IV's, the imposed condition on IV's and the required number of key-stream bits from each IV are determined by Table 2 which provides many tradeoffs.

5. Conclusion

In this paper we presented a chosen IV key-recovery attack on Original Pomaranch. In our attack we used the idea of Linear Equivalence Bias which was introduced in [7, 5]. The complexity of our chosen IV attack is $O(2^{73.5})$ on Original Pomaranch which is not less than $O(2^{52})$, the complexity achieved in [1]. However, our attack is applicable to the first version of proposed key-IV setup of Tweaked Pomaranch with computational complexity of $O(2^{117.7})$ while the attack of [1] is not applicable. The second version of the proposed key-IV setup for Tweaked Pomaranch is immune against our attack.

References

1. Cid, C., Gilbert, H., Johansson, T.: Cryptanalysis of Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/060 (2005) http://www.ecrypt.eu.org/stream/papersdir/060.pdf.
2. eSTREAM, the ECRYPT Stream Cipher Project http://www.ecrypt.eu.org/stream.
3. Jansen C. J.A.: Stream cipher Design: Make your LFSRs jump! In SASC, Workshop Record, ECRYPT Network of Excellence in Cryptology, pages 94:108, 2004.
4. Jansen, C.J.A., Helleseth, T., Kholosha, A.: Cascade jump controlled sequence generator (CJCSG). In: Symmetric Key Encryption Workshop, Workshop Record, ECRYPT Network of Excellence in Cryptology (2005) http://www.ecrypt.eu.org/stream/ciphers/pomaranch/pomaranch.pdf.
5. Jansen, C.J.A., Kholosha, A.: Countering the Correlation Attack on Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/070 (2005) http://www.ecrypt.eu.org/stream/papersdir/070.pdf.
6. Jansen, C.J.A., Kholosha, A.: Pomaranch is Sound and Healthy. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/074 (2005) http://www.ecrypt.eu.org/stream/papersdir/074.pdf.
7. Khazaei, S.: Cryptanalysis of Pomaranch (CJCSG). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/065 (2005) http://www.ecrypt.eu.org/stream/papersdir/065.pdf.

Pomaranch - Design and Analysis of a Family of Stream Ciphers*

Tor Helleseth[1], Cees J.A. Jansen[2] and Alexander Kholosha[1]

[1] The Selmer Center
Department of Informatics, University of Bergen
P.O. Box 7800, N-5020 Bergen, Norway
[2] Banksys NV
Haachtsesteenweg 1442
1130 Brussels, Belgium
{Tor.Helleseth,Alexander.Kholosha}@ii.uib.no; cja@iae.nl

Abstract. Pomaranch is a synchronous, hardware-oriented stream cipher submitted to eSTREAM, the ECRYPT Stream Cipher Project. The cipher is designed as a cascade clock-controlled key-stream generator built on jump registers. This paper presents a discussion over the attacks on Pomaranch discovered so far. Particular focus is made on a new inherent property of jump registers that allows to construct their linear equivalences. For the concrete configuration of the registers in Pomaranch this allows to build an efficient key-recovery attack. Finally, a few tweaks that secure the cipher against the known attacks are suggested.

Key words: cryptanalysis, jump register, key-recovery attack, linear equivalences, Pomaranch, stream cipher.

1 Introduction

Pomaranch (also known as a Cascade Jump Controlled Sequence Generator or CJCSG) [3] is a synchronous bit-oriented stream cipher, one of the ECRYPT Stream Cipher Project [4] candidates. It uses 128-bit keys and in its original design accommodates an Initial Value (IV) of 64 up to 112 bits long.

The algorithm uses a one clock pulse cascade construction of so called jump registers [5] being essentially linear finite state machines with a special transition matrix. Moreover, the characteristic polynomial of the transition matrix was made to be primitive and satisfying additional constraints that arise from the need to use the register in a cascade jump control setup. The principal advantage of jump registers over the classical clock-controlled arrangements is their ability to move a Linear Feedback Shift Register (LFSR) to a state that is more than one step ahead but without having to step through all the intermediate states. The transition matrix of the jump registers in Pomaranch has been chosen so to

* The work of the authors from the Selmer Center was supported by the Norwegian Research Council. The results of this paper are contained in part in [1, 2].

secure the design against side-channel attacks while preserving all the advantages of irregular clocking.

Pomaranch was analyzed quite intensively in the recent period of time since it was submitted. A few efficient key-recovery attacks were found. The first one in [6] exploits the weakness of the IV setup procedure. This attack allows finding the key with the computational complexity $O(2^{52})$ if the attacker can obtain a few key-stream sections generated using specially chosen IV's. Another attack described in [7] is a key-recovery correlation attack that works with the complexity $O(2^{95.4})$ requiring about $2^{71.8}$ bits of the key-stream. This computational complexity can be decreased to $O(2^{73.5})$ if the chosen IV scenario is assumed (see [8]). In this paper we analyze linear equivalences in the key-stream generated by Pomaranch. This provides an interesting insight in the design rationale of the cipher and allows to secure it against known correlation attacks. We also suggest a new IV setup procedure that provides better diffusion of IV bits giving protection against chosen IV attacks.

In Section 2 we outline some details of Pomaranch key-stream generator that are important for understanding the analysis that follows. Also here we present a new IV setup procedure that provides better security against the chosen IV attacks. Section 3 contains main theoretical results about finding linear equivalences for jump registers and calculating corresponding biases. We apply the theory to the concrete configuration of Pomaranch registers in Section 4 that leaded to the efficient key-recovery attack in [7]. Slight modification of the Pomaranch jump register configuration allows to protect against this type of attacks increasing the complexity to $O(2^{133.4})$ (higher than the exhaustive key search) and this is discussed in Section 5. In Section 6 we show the ways for an efficient hardware implementation of the cipher. We conclude with Section 7 presenting the list of tweaks to the original version of Pomaranch that secure the cipher against all the so far known attacks.

2 Outline of Pomaranch

Pomaranch follows a classical design of a synchronous, additive, bit-oriented stream cipher and consists of a key-stream generator producing a secure sequence of bits that is further bitwise XORed with the plain text previously converted into bits. After the initialization that comprises key setup, IV setup and the runup, the key-stream generator of Pomaranch is run in the generation mode showed in Fig. 1.

The generator consists of nine irregularly clocked registers R_1 to R_9 (also called Jump Registers (JR)) that are combined in a cascade construction. Each register implements an autonomous Linear Finite State Machine (LFSM) and is built on 14 memory cells each of them acting either as a simple delay shift cell (S-cell) or feedback cell (F-cell), depending on the value of the Jump Control (JC) bit. At any moment, half of the cells in each register are S-cells, while the others are F-cells which is seen as an important feature against power and side-channel attacks. A LFSM implemented by the JR has the following transition

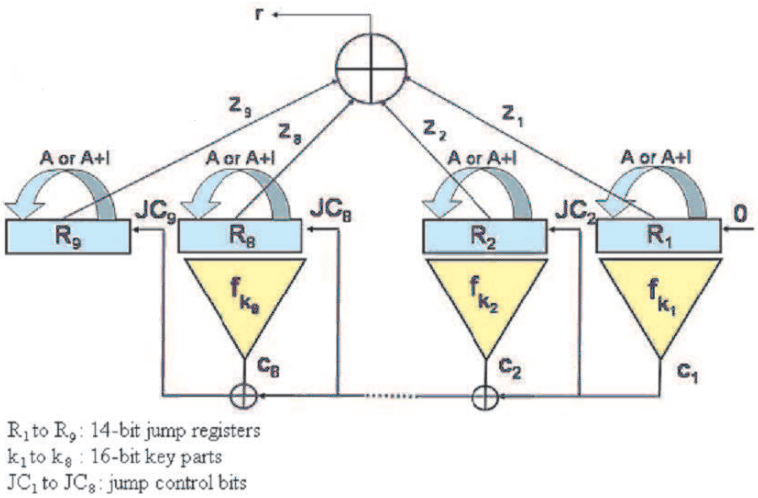


Fig. 1. Key-Stream Generation Mode of Pomaranch

matrix

$$A = \begin{pmatrix} d_L & 0 & 0 & \cdots & 0 & 1 \\ 1 & d_{L-1} & 0 & \cdots & 0 & t_{L-1} \\ 0 & 1 & d_{L-2} & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & 1 & d_2 & t_2 \\ 0 & 0 & \cdots & 0 & 1 & d_1 + t_1 \end{pmatrix} \tag{1}$$

over GF(2), where $t_1, \ldots, t_{L-1}$ are defined by the positions of feedback taps and nonzero $d_1, \ldots, d_L$ correspond to the positions of F-cells in the register. In the particular case of Pomaranch $L = 14$, only $t_6 = 1$ and $d_1 = d_3 = d_7 = d_8 = d_9 = d_{11} = d_{13} = 1$. Transition matrix A is applied if the JC value is zero, otherwise, all cells are switched to the opposite mode which is equivalent to changing the transition matrix to $A + I$ with I being the identity matrix. Let R_i^t denote the state of the register R_i at a time $t \geq 0$. Then

$$R_i^{t+1} = (A + JC_i^t \cdot I)R_i^t \quad (i = 1, \ldots, 9) \ ,$$

where JC_i^t denotes the jump control bit for R_i at time t.

The 128-bit key K is divided into eight 16-bit subkeys k_1 to k_8. The current states of the registers R_1^t to R_8^t are nonlinearly filtered using a function that involves the corresponding subkey k_i $(i = 1, \ldots, 8)$. These functions provide an output of eight bits c_1^t to c_8^t which are used to produce the bits JC_2^t to JC_9^t controlling the registers R_2 to R_9 at time t as follows

$$JC_i^t = c_1^t \oplus \ldots \oplus c_{i-1}^t \quad (i = 2, \ldots, 9) \ .$$

The jump control bit JC_1 of register R_1 is permanently set to zero. The key-stream bit generated at time t (denoted r^t) is the XOR of nine bits z_1^t to z_9^t tapped from the second cell of the register states R_1^t to R_9^t so $r^t = z_1^t \oplus \ldots \oplus z_9^t$.

Shift Mode. This mode is used during the initialization and IV setup of the key-stream generator. In this mode the output bit c_i^t from section $i = 1, \ldots, 8$ is XORed with the feedback of the register R_{i+1}. The tap from cell 1 in R_9 is XORed with the feedback of R_1 and this closes "the big loop". Configuration of the jump registers does not change in the Shift Mode, they all operate as if the JC bit was constantly zero.

The Shift Mode is used to make the register contents depend on all initial content bits and all key bits. This mode defines a key dependent one-to-one mapping of the set of all 126-bit states onto itself. Indeed, let $R_i^t = (r_{i,14}^t, \ldots, r_{i,1}^t)$ $(1 \le i \le 9)$ and $c_i^t = f_i(R_i^t)$ be the output bit of the Key Map of section i $(1 \le i \le 8)$ at a time t. If A denotes the transition matrix (1) which is fixed for all the registers as if the JC bit was constantly zero, then the following equations define the Shift Mode:

$$\begin{aligned} R_1^{t+1} &= R_1^t A \oplus (0, \ldots, 0, r_{9,1}^t) \\ R_i^{t+1} &= R_i^t A \oplus (0, \ldots, 0, f_{i-1}(R_{i-1}^t)) \quad (i = 2, \ldots, 9) \ . \end{aligned}$$

From the concrete form of matrix A applied in the Shift Mode it is clear that $r_{i,2}^{t+1} = r_{i,1}^t$ $(1 \le i \le 9)$. So the inverse of the above equations can be written as

$$\begin{aligned} R_1^t &= \left(R_1^{t+1} \oplus (0, \ldots, 0, r_{9,2}^{t+1})\right) A^{-1} \\ R_i^t &= \left(R_i^{t+1} \oplus (0, \ldots, 0, f_{i-1}(R_{i-1}^t))\right) A^{-1} \quad (i = 2, \ldots, 9) \ . \end{aligned}$$

This shows that the Shift Mode defines an invertible onto mapping which needs to be a bijection. Also note that in the Shift Mode the worst case diffusion of all IV bits is achieved after 27 steps and the IV-plus-key bits diffusion is achieved after 36 steps.

IV Setup. The original IV setup turned out to be extremely weak against chosen IV attacks (see [6, 8]) since it provided no diffusion of IV bits into the whole internal state of the key-stream generator. Therefore, the setup procedure had to be changed considerably. We skip here the description of the original IV setup and present the following new procedure:

1. The IV can have an arbitrary length in the range from 64 to 126 bits. If the IV length is less than 126 then extend the IV to 126 bits by cyclically repeating its bits.
2. XOR the 126-bit (extended) IV with the Initialization Vector saved after the key setup (see [3]) and load the result into the 9 jump registers.
3. Run the generator in the Shift Mode for 96 steps.
4. If any of the 9 registers has the all-zero state then set its least significant bit to 1.
5. Perform a runup of 64 steps in the Key-Stream Generation Mode discarding the output bits.

3 Linear Equivalences of Jump Registers

Configuration of the jump registers in Pomaranch is chosen in such a way that the characteristic polynomial $C(x)$ of the binary transition matrix A in (1) is primitive and is neither self-reciprocal nor self-dual nor dual-reciprocal, i.e., A belongs to a primitive S_6 set, that is a set of six primitive polynomials which are each others reciprocals and duals (for the details see [5]). Obviously, the characteristic polynomial of $A+I$ is the dual $C^{\perp}(x) = C(x+1)$ and is primitive. Clocking of the jump registers is implemented by multiplying the state by the transition matrix A or $A + I$.

Let $Z = \{z^t\}_{t=0}^{\infty}$ denote the output sequence of a jump register being any component in the sequence of register states. Starting from some state R^t, the first output bit z^t is not affected by the jump control bits in $(JC^t, \ldots, JC^{t+L-1})$, the second output bit z^{t+1} is defined by JC^t, the third z^{t+2} is defined by (JC^t, JC^{t+1}) and so on.

Every output bit can be presented as a linear combination of L bits from the initial state R^0 and thus any $L + 1$ bits of the output sequence are linearly dependent. The linear relation is defined by the relevant jump control bits and does not depend on the initial state of the register. Take such a relation that holds on $L+1$ consecutive bits of Z at the shift position t. Also assume that this relation holds for every component sequence of the register (i.e., irrespective of position the output sequence is tapped from). This means that for some set of binary coefficients $(\ell_0, \ell_1, \ldots, \ell_L)$ and any initial state we have $\ell_0 z^t + \ell_1 z^{t+1} + \ldots + \ell_L z^{t+L} = 0$ or equivalently that the following identity holds

$$\ell_0 I + \sum_{i=1}^{L} \ell_i \prod_{k=0}^{i-1} (A + JC^{t+k} I) = 0 \ .$$

Since $C(x)$, the characteristic polynomial of A, is in particular, irreducible, it coincides with the minimal polynomial of A. Thus, the latter identity holds if and only if

$$\ell_0 + \sum_{i=1}^{L} \ell_i \prod_{k=0}^{i-1} (x + JC^{t+k}) = \sum_{i=0}^{L} \ell_i x^{i-k_i} (x+1)^{k_i} = C(x) \ , \qquad (2)$$

where $0 \le k_i \le i$ are defined by the control bits $JC^t, \ldots, JC^{t+L-1}$, namely, $k_0 = 0$ and k_i is equal to the binary weight of vector $(JC^t, \ldots, JC^{t+i-1})$. Thus, if assuming the jump control sequence is purely random, then the values of k_i are binomially distributed. Since the degree of $C(x)$ is L and $C(0) = 1$ then the coefficients at the highest-order and the constant term of the polynomial standing on the left hand side of (2) should be nonzero, i.e., $\ell_0 = \ell_L = 1$ for any linear relation in the jump register output. Given an arbitrary jump control sequence (that provides the values of k_i) the solution of (2) for the unknowns ℓ_i can be found applying a simplified version of Gaussian elimination. Such a solution always exists and, in particular, this can be easily seen from the matrix of the system which is triangular and contains ones on the main diagonal.

The complexity of solving the system is linear in L (if counting word operations). Indeed, let the binary coefficients of the binomial expansion of an additive term $x^{i-k_i}(x+1)^{k_i}$ be packed into words. Then, starting with $i=0$ and $x^{0-k_0}(x+1)^{k_0}=1$, every next term, depending on the value of JC^{t+i}, is equal to the previous one multiplied by x (shift the coefficient vector by one bit) or multiplied by $x+1$ (shift and add). Thus, expansions of all $L+1$ terms can be computed with $O(L)$ word operations. Further set $\ell_L=1$ and add the coefficient vector of $x^{L-k_L}(x+1)^{k_L}$ to $C(x)$. If the degree of the obtained polynomial is equal $L-1$ then set $\ell_{L-1}=1$, otherwise set $\ell_{L-1}=0$. Proceed further in a similar way till all the unknowns ℓ_i are found. The total complexity remains linear in L.

Take a linear relation defined by the set of binary coefficients $\ell_0,\ldots,\ell_L$ with $\ell_0=\ell_L=1$ and take a set of weights $\{k_i \mid i=0,\ldots,L;\ell_i=1\}$ with $k_0=0$, $k_i\le k_j$ if $i<j$ and $k_j-k_i\le j-i$ such that

$$\sum_{i=0,\ldots,L;\ \ell_i=1} x^{i-k_i}(x+1)^{k_i}=C(x)\ . \tag{3}$$

Now take two neighboring additive terms from the left hand side of the last identity being $x^{i-k_i}(x+1)^{k_i}$ and $x^{j-k_j}(x+1)^{k_j}$ with $i<j$. Then the number of possible $(j-i)$-long sections of the jump control sequence leading from $x^{i-k_i}(x+1)^{k_i}$ to $x^{j-k_j}(x+1)^{k_j}$ is equal to $\binom{j-i}{k_j-k_i}$ (these are exactly the sequences with the binary weight of $(JC^{t+i},\ldots,JC^{t+j-1})$ equal to k_j-k_i). In a similar manner, starting from the constant term $x^0(x+1)^0$ at $\ell_0=1$ and proceeding till the highest-order term at $\ell_L=1$ is reached we can find the total number of L-long jump control sequences that correspond to the given linear relation and the set of weights. This number is obtained as a product of the relevant binomial coefficients for all $\ell_i\ne 0$ and $i>0$.

As can be seen from (2), the set of all possible linear relations that correspond to different control sequences and the number of their occurrences only depend on the characteristic polynomial $C(x)$ of the jump register. As the linear relation occurring most often plays an essential role in the key-recovery attack, we will call its occurrence number the *Linear Equivalence Bias* (LEB) of the polynomial. All occurrence numbers together form a *Linear Equivalence Spectrum* (LES) of the polynomial. It can be easily seen by interchanging the roles of x and $x+1$ that $C(x)$ and $C^{\perp}(x)$ have the same LES. The LES value for any linear relation can be calculated as a sum consisting of terms being the product of binomial coefficients. Every set of weights k_i satisfying (3) provides one additive term to the sum.

Again take a linear relation and a set of weights satisfying (3). Applying the following *Doubling Rule*

$$x^a(x+1)^b=\begin{cases} x^{a-1}(x+1)^b+x^{a-1}(x+1)^{b+1},\\ x^a(x+1)^{b-1}+x^{a+1}(x+1)^{b-1}\end{cases}$$

to different additive terms $x^{i-k_i}(x+1)^{k_i}$ in the left hand side of (3) we can find other relations that have a nonzero LES value. If the original relation has

$\ell_{i-1} = 0$ and $\ell_i = 1$ then the new one has $\ell_{i-1} = \ell_i = 1$ that can be seen as doubling of the coefficient ℓ_i. It is not difficult to see that having the LES value of a linear relation that is expressed as a sum of products and applying the doubling rule to any $\ell_i = 1$ (assuming $\ell_{i-1} = 0$) gives us another relation with $\ell_i = \ell_{i-1} = 1$ and a sum with a doubled number of additive terms that is equal to the LES value of a new relation.

The most obvious example is to apply the doubling rule to the highest-order term at the coefficient $\ell_L = 1$ when $\ell_{L-1} = 0$ which leaves ℓ_L unchanged and gives rise to $\ell_{L-1} = 1$. Due to the binomial identity $\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$ the LES value computed for the new linear relation will be the same as for the old one. This, in particular, implies that all the values in an LES appear even number of times. Applying the doubling rule to other terms results in new relations having higher or lower LES values. This feature will be illustrated in Section 4. Applying the doubling rule in the opposite direction results in the merge of two terms.

Note that using the presented technique we can evaluate LES values for some linear relations of length $L+1$ in the output sequence of a jump control register. In some cases this value is equal to the LEB of a polynomial meaning that we have found a relation that belongs to the ones occurring most often. However, we can not currently provide the algorithm for evaluating the LEB with the complexity lower than $O(L\,2^L)$ (checking through all JC sequences of length L and each time implementing a simple version of Gaussian elimination of length L). Finding a less complex algorithm remains an interesting open problem.

4 Key-Recovery Attack using Linear Equivalences

In this section we calculate the LEB for the concrete configuration of jump registers in Pomaranch as well as for some minor modifications of the cipher. We also give some intuitive technique for finding the LEB in general. The key-recovery correlation attack suggested in [7] uses exactly those linear relations in the key-stream with the LES value equal to the LEB.

The characteristic polynomial of the transition matrix (1) can be found directly as follows

$$C(x) = 1 + \sum_{i=0}^{L-1} t_i \prod_{j=i+1}^{L} (d_j + x) \ ,$$

where $t_0 = 1$ is introduced for simplicity of the formula. Now assume L is even, a jump register of length L has two feedback taps (i.e., only $t_0 = t_n = 1$ for some $0 < n < L$), there are k F-cells among the first n cells (i.e., only k values from $d_1, \ldots, d_n$ are nonzero) and the total number of F-cells is $L/2$. Then

$$C(x) = 1 + x^{\frac{L}{2}+k-n}(x+1)^{\frac{L}{2}-k} + x^{\frac{L}{2}}(x+1)^{\frac{L}{2}} \ . \tag{4}$$

Placing this in (2) one immediately spots the evident linear relation $z^t + z^{t+L-n} + z^{t+L} = 0$ that we call *basic*. The corresponding equation coming from (2)

$$1 + x^{L-n-k_{L-n}}(x+1)^{k_{L-n}} + x^{L-k_L}(x+1)^{k_L} = 1 + x^{\frac{L}{2}+k-n}(x+1)^{\frac{L}{2}-k} + x^{\frac{L}{2}}(x+1)^{\frac{L}{2}}$$

can be shown to be satisfied only by $k_L = L/2$ and $k_{L-n} = L/2 - k$. Thus, this trinomial linear relation has the LES value given by

$$\binom{L-n}{\frac{L}{2}-k}\binom{n}{k} \ . \tag{5}$$

Assuming $n > 1$ and applying the doubling rule to the senior term in (4) we get another relation $z^t + z^{t+L-n} + z^{t+L-1} + z^{t+L} = 0$ having the same LES value.

Restricting our options further to the registers of length $L = 14$ that have a characteristic polynomial belonging to a primitive S_6 set, we are left with the following five alternative (n, k)-pairs of parameters $(6, 2)$, $(7, 2)$, $(7, 3)$, $(8, 3)$ and $(11, 5)$. Note that two polynomials corresponding to parameters (n, k) and $(n, n-k)$ form a dual pair. By (5), the corresponding LES values for the basic trinomial relations are 840, 441, 1225, 840 and 1386 respectively. For all the configurations except $(7, 2)$ these turned out to be the LEB values of the characteristic polynomials (4). For the remaining case $(n, k) = (7, 2)$ the basic relation is $z^t + z^{t+7} + z^{t+14} = 0$. Applying the doubling rule to the middle term here we obtain a new linear relation $z^t + z^{t+6} + z^{t+7} + z^{t+14} = 0$ and a new LES value $\binom{6}{1} \cdot \binom{7}{1} + \binom{6}{2} \cdot \binom{7}{3} = 567$ equal to the LEB in this case. On the other hand, for $(n, k) = (6, 2)$, applying the doubling rule to the middle term in the basic relation $z^t + z^{t+8} + z^{t+14} = 0$ we obtain a new linear relation $z^t + z^{t+7} + z^{t+8} + z^{t+14} = 0$ having the LES value $\binom{7}{5} \cdot \binom{6}{1} + \binom{7}{4} \cdot \binom{6}{3} = 826$, the second largest for this polynomial. We believe that in general, starting from the basic relation and consecutively applying the doubling rule, splitting and merging various terms, one can find all linear relations that hold at least for one control sequence. Tracking the LES values computed after each split or merge one can also find the LEB of the characteristic polynomial.

The concrete parameters initially chosen for Pomaranch are $(6, 2)$ giving the basic trinomial relation $z^t + z^{t+8} + z^{t+14} = 0$. The resulting LEB of $\binom{8}{5} \cdot \binom{6}{2} = 840$ is high enough to mount the key-recovery correlation attack (see [7]). Another linear relation $z^t + z^{t+8} + z^{t+13} + z^{t+14} = 0$ with the same LES value 840 is obtained applying the doubling rule to the senior term of the basic relation. The use of both relations makes the attack more efficient. The LES of the corresponding characteristic polynomial contains just 334 linear relations having nonzero occurrence numbers out of $2^{13} = 8192$ possible.

Suppose the LEB value of the characteristic polynomial of a jump register on L memory cells is $F > 0$ that corresponds to the linear relation on the output bits Z defined by $\boldsymbol{\ell} = (\ell_0, \ell_1, \ldots, \ell_L)$. Assume that the jump control sequence $\{JC^t\}_{t=0}^{\infty}$ is a sequence of independent and identically uniformly distributed random variables. In our case it is convenient to define the distribution bias of a binary random variable x as $\epsilon = 1 - 2\Pr\{x = 1\}$. Then the following random binary sequence $e^t = \sum_{i=0}^{L} \ell_i z^{t+i}$ $(t = 0, 1, 2, \ldots)$ has a nonuniform distribution with the bias $\epsilon = F/2^L$. Indeed, let H denote the event that a random subsection $(JC^t, \ldots, JC^{t+L-1})$ is one of those F that correspond to $\boldsymbol{\ell}$. The complementary event of H is denoted by $\overline{H}$. It is clear that $\Pr\{H\} = F/2^L$ and $\Pr\{e^t = 1 \mid H\} = 0$. The probability $\Pr\{e^t = 1 \mid \overline{H}\}$ can be considered equal

to $1/2$ because in this case e^t has a uniform distribution. Therefore, by the rule of total probability

$$\begin{aligned}\Pr\{e^t = 1\} &= \Pr\{e^t = 1 \mid H\}\Pr\{H\} + \Pr\{e^t = 1 \mid \overline{H}\}\Pr\{\overline{H}\} \\ &= 1/2(1 - F/2^L) \ .\end{aligned}$$

For the characteristic polynomial in Pomaranch the LEB is equal to 840 and $\epsilon = 840/2^{14} \approx 2^{-4.3}$. This number was first discovered by Khazaei in [7] using exhaustive computations.

It can be concluded that output bits of every jump register section, except for the first one that is clocked regularly, satisfy the following linear relations

$$z^t + z^{t+8} + z^{t+14} = 1 \quad \text{and} \quad z^t + z^{t+8} + z^{t+13} + z^{t+14} = 1$$

with the bias $\epsilon \approx 2^{-4.3}$. This was used in [7] to mount a key-recovery correlation attack on the cipher. If using both of the above relations the required minimum amount of key-stream bits is one half of the number given by the formula

$$N_0 = 14/C(0.5(1 - \epsilon^8)) \tag{6}$$

where $C(p) = 1 + p\log_2(p) + (1-p)\log_2(1-p)$ is the Channel Capacity of the corresponding Binary Memoryless Symmetric Channel. In total, the secret key of Pomaranch can be found using $2^{71.8}$ bits of the output sequence with the computational complexity $O(2^{95.4})$. Needed amount of the key-stream can be reduced if all 334 linear relations found in the LES are used.

5 Modified Jump Registers for Pomaranch

It is clear that the ideal configuration of a jump register should provide a lowest possible LEB value. Note that parameter pair $(7, 2)$ with LEB of 567 would have been a better choice, but even with this configuration our attack recovers the key with the complexity lower than the exhaustive key search. We conclude that all the characteristic polynomials having two feedback taps are not secure enough to counter the attack. Thus, in order to find a characteristic polynomial with a sufficiently low LEB, the Pomaranch jump register has to be changed to have three or more feedback taps.

Consider the registers having exactly three taps. Assume there is one tap, the rightmost, at position n_1 with k_1 feedback cells among cells 1 to n_1. The other tap is at position $n_2 > n_1$, with k_2 feedback cells among cells $n_1 + 1$ to n_2. The modified characteristic polynomial now becomes

$$C(x) = 1 + x^{\frac{L}{2}+k_1+k_2-n_2}(x+1)^{\frac{L}{2}-k_1-k_2} + x^{\frac{L}{2}+k_1-n_1}(x+1)^{\frac{L}{2}-k_1} + x^{\frac{L}{2}}(x+1)^{\frac{L}{2}}$$

for $L = 14$. The LES of this polynomial contains the basic relation $z_t + z_{t+L-n_2} + z_{t+L-n_1} + z_{t+L} = 0$.

Searching through all relevant (n_1, n_2, k_1, k_2) quadruplets results in a set of 16 primitive S_6-set polynomials, amongst which are the five polynomials already

obtained for two taps. The polynomial with the least LEB in this set $x^{14}+x^{13}+x^{12}+x^{11}+x^9+x^7+x^5+x^4+x^2+x+1$ is obtained for $n_1=4$, $n_2=8$, $k_1=k_2=1$ and has an LEB equal to 124 and an LES containing 1088 nonzero values. The linear relation $z_t+z_{t+6}+z_{t+10}+z_{t+14}=0$ occurs $\binom{6}{1}\cdot\binom{4}{3}\cdot\binom{4}{3}=96$ times. Performing a doubling operation on the 6th order term yields a relation which occurs 124 times that is equal to the LEB value.

Plugging in the bias of $124/2^{14}\approx 2^{-7.05}$ of the jump register in (6) results in the attack complexity of $O(2^{133.4})$ with $2^{116.9}$ bits of the key-stream required (the latter can be reduced if all 1088 relations are used). This complexity exceeds the one of the exhaustive search over the key space containing 2^{128} elements.

Note that an alternative way to secure Pomaranch against the described key-recovery attack is to make the sections have different characteristic polynomials. Hereupon each section would have a different most probable linear equivalence. Thus, adding the outputs from all the sections will compensate for the bias. However, keeping all the sections the same definitively looks like a more "elegant" solution.

6 Efficient Hardware Implementation

In this section we consider the tweaked hardware version of Pomaranch that consists of 6 sections and accommodates the key of 80 bits. The list of all tweaks is presented in Section 7.

The CJCSG is ideally suited for hardware implementation since it requires standard components and has no complex circuits causing timing bottlenecks. The hardware version of the CJCSG consists of 6 sections with 5 of them containing the Key Map. The linear shift register part (jump register) uses 14 memory cells, each with an XOR and a switch. Typically, this takes about 175 gates (two-input equivalent). The 9-to-7 S-box in the Key Map is the most expensive real-estate, followed by the 7-to-1 Boolean function and 16 XOR's. Implementation of these components by direct synthesis of the Boolean circuitry is estimated at 1000 gates. No attempts have been made to optimize the footprint of these circuits by means of a silicon compiler. For the complete design a total estimate is obtained of $5\cdot 1000+6\cdot 175\approx 6000$ gates. Reduction of the gate-complexity of the S-box can lower this number substantially as can be seen from the following.

First note that the 9-to-7 S-box presented in Appendix A is defined by the inversion operation in the multiplicative group of GF(2^9) when the finite field is defined by the irreducible polynomial $f(x)=x^9+x+1$. Further the most and the least significant bits (msb and lsb) of the result are deleted to obtain a 7-bit value. We can define a more efficient (having lower gate-complexity) implementation of the inverse in GF(2^9) using inverses in the subfield GF(8), i.e., inverses are calculated in GF(8^3) instead. The elements of GF(8^3) are represented by polynomials of degree at most 2 over GF(8) and operations in the field are carried out modulo an irreducible polynomial $Q(x)=x^3+a_2x^2+a_1x+a_0$ over GF(8). Operations in GF(8) can be implemented with low complexity by table lookups using one of the following moduli x^3+x+1 or x^3+x^2+1. Sum-

ming up all the above said, the following steps could lead to a lower complexity implementation of the S-box:

1. Find a primitive element of GF(2^9) modulo $x^9 + x + 1$ and calculate the polynomial $Q(x)$ (see [9]).
2. Let $b_2x^2+b_1x+b_0$ be the inverse modulo $Q(x)$ of a polynomial $c_2x^2+c_1x+c_0$ over GF(8). Find analytical expressions for the coefficients b_2, b_1, b_0 as a function of c_2, c_1, c_0 and a_2, a_1, a_0. These are found as a solution of a system of three linear equations in three unknowns that can be solved applying Cramer's rule. The operations required to calculate the b_i from the given c_i and a_i ($i = 0, 1, 2$) are multiplications, additions and inverse in GF(8).
3. The number of subfield operations for finding the solutions amounts to 18 multiplications, 6 constant multiplications, 8 XOR's and 1 inverse.
4. The gate-complexity of multiplication and inverse in GF(8) is determined by finding the ANF's for the two irreducible polynomials and two bases each (Galois counter and LFSR basis). This results in: inverse between (6 gates and 1 inverter) and (10 gates and 3 inverters), where inverter means binary inverter, so say 10 gates; multiplication 17 or 18 gates; constant multiplication costs only 1 or 2 gates (XOR's). The total cost is therefore $18 \cdot 17 + 6 \cdot 2 + 8 + 10 = 336$ gates.
5. A linear transform and its inverse are needed to map 9-bit vectors to vectors over GF(8^3) and back, where the inverse transform is combined with the 7-to-1 Boolean function. The cost of these 9-by-9 matrices is estimated at 40 XOR's. Hence, the total cost is estimated at 400 gates (two-input AND, OR, XOR, etc).

We conclude that for a hardware implementation of 6 sections with $5 \cdot 16 = 80$ key bits the total gate-count would amount to $5 \cdot 400 + 6 \cdot 175 \approx 3000$ gates. Note two things here: in practice a good silicon compiler may even do better by reusing intermediate results at several places; the estimate for the gate-complexity needed to implement the full inverse while deleting the msb and lsb can further reduce the gate-count.

7 Conclusion and Tweaks to Pomaranch

We considered a jump register arrangement that proved to be a powerful and efficient building block for stream ciphers that use irregular clocking of shift registers. We have identified a new inherent property of such arrangements which should always be observed in the relevant types of cipher design. Jump registers with badly chosen parameters allow building linear equivalences providing a close approximation of the output sequences.

Using the discovered property a 128-bit key of Pomaranch can be recovered with the complexity $O(2^{95.4})$ requiring less than $2^{71.8}$ bits of the key-stream. Therefore, we have to introduce a minor change in the configuration of the jump register section in Pomaranch that gives protection against this attack bringing its complexity up to $O(2^{133.4})$ with at most $2^{116.9}$ bits of the key-stream required

that exceeds the complexity of the exhaustive key search. Moreover, this new potential weakness can be exploited to attack other stream ciphers that use irregular clocking. The suggested technique has a general character and can be dangerous to other clock-controlled arrangements. This issue will become a focus for our future research.

Following is the list of tweaks we apply to Pomaranch as specified in its original version [3]. The second and the third items are the response to the attack in [7] (these changes where discussed in all the details in Section 5) and the last change prevents the chosen IV attacks from [6, 8].

1. Hardware-oriented 80-bit key version of the CJCSG is added. The only difference between the full 128-bit version and the 80-bit version is the total number of jump register sections that is equal respectively to 9 and 6 and the number of Shift Mode steps during the IV setup that is equal to 96 and 80 respectively.
2. Feedback taps of jump registers are taken now from cells number 4, 8 and 14. The positions of the F- and S-cells in the registers are FFSFFFSSFSSFSS.
3. Input to the Key Map is taken from the cells of the jump registers number 1, 2, 3, 5, 6, 7, 9, 10, 11.
4. The new IV setup procedure is defined as described in Section 2 under the subtitle "IV Setup".

References

1. Jansen, C.J.A., Kholosha, A.: Countering the correlation attack on Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/070 (2005) http://www.ecrypt.eu.org/stream/papersdir/070.pdf.
2. Jansen, C.J.A., Kholosha, A.: Pomaranch is sound and healthy. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/074 (2005) http://www.ecrypt.eu.org/stream/papersdir/074.pdf.
3. Jansen, C.J.A., Helleseth, T., Kholosha, A.: Cascade jump controlled sequence generator (CJCSG). In: Symmetric Key Encryption Workshop, Workshop Record, ECRYPT Network of Excellence in Cryptology (2005) http://www.ecrypt.eu.org/stream/ciphers/pomaranch/pomaranch.pdf.
4. eSTREAM: The ECRYPT stream cipher project (2005) http://www.ecrypt.eu.org/stream/.
5. Jansen, C.J.A.: Stream cipher design based on jumping finite state machines. Cryptology ePrint Archive, Report 2005/267 (2005) http://eprint.iacr.org/2005/267/.
6. Cid, C., Gilbert, H., Johansson, T.: Cryptanalysis of Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/060 (2005) http://www.ecrypt.eu.org/stream/papersdir/060.pdf.
7. Khazaei, S.: Cryptanalysis of pomaranch (CJCSG). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/065 (2005) http://www.ecrypt.eu.org/stream/papersdir/065.pdf.
8. Hasanzadeh, M., Khazaei, S., Kholosha, A.: On IV setup of Pomaranch. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/082 (2005) http://www.ecrypt.eu.org/stream/papersdir/082.pdf.
9. Sunar, B., Savas, E., Çetin K. Koç: Constructing composite field representations for efficient conversion. IEEE Transactions on Computers **52** (2003) 1391–1398

Evaluation of SOSEMANUK With Regard to Guess-and-Determine Attacks

Yukiyasu Tsunoo[1], Teruo Saito[2], Maki Shigeri[2], Tomoyasu Suzaki[2], Hadi Ahmadi[3], Taraneh Eghlidos[4], and Shahram Khazaei[5]

[1] NEC Corporation
1753 Shimonumabe, Nakahara-Ku, Kawasaki, Kanagawa 211-8666, Japan
tsunoo@BL.jp.nec.com
[2] NEC Software Hokuriku Ltd.
1 Anyoji, Hakusan, Ishikawa 920-2141, Japan
{t-saito@qh, m-shigeri@pb, t-suzaki@pd}.jp.nec.com
[3] School of Electrical Engineering, Sharif University of Technology, Tehran, Iran.
hadimc@mehr.sharif.edu
[4] Electronics Research Center, Sharif University of Technology, Tehran, Iran.
teghlidos@sharif.edu
[5] Zaeim Electronic Industries Company, P.O. BOX 14155-1434, Tehran, Iran.
khazaei@zaeim.com

Abstract. This paper describes the attack on SOSEMANUK, one of the stream ciphers proposed at eSTREAM (the ECRYPT Stream Cipher Project) in 2005. The cipher features the variable secret key length from 128-bit up to 256-bit and 128-bit initial vector. The basic operation of the cipher is performed in a unit of 32 bits i.e. "word", and each word generates keystream.
This paper shows the result of guess-and-determine attack made on SOSEMANUK. The attack method enables to determine all of 384-bit internal state just after the initialization, using only 2^4-word keystream. This attack needs about 2^{224} computations. Thus, when secret key length is longer than 224-bit, it needs less computational effort than an exhaustive key search, to break SOSEMANUK. The results show that the cipher has still the 128-bit security as claimed by its designers.
Key words: SOSEMANUK, ECRYPT, eSTREAM, stream cipher, pseudo-random number generator, guess-and-determine attack

1 Introduction

Everywhere, cipher standardization project has been encouraged vigorously. It is exemplified by the Advanced Encryption Standard (AES) [1], or the New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project whose goal is to establish European standard cipher [3]. NESSIE project aims to choose secure cipher primitives, and in fact, they chose a stream cipher. However, many attacks against the stream ciphers proposed for NESSIE project were proposed during the 3-year evaluation phase, and finally, no stream cipher candidate remained. Thus, widespread attention is focused on stream cipher design and attacks against them.

In February 2004, European Network of Excellence for Cryptology (ECRYPT) was established. Its goal is to encourage the cooperation among European researchers on information security. In 2005, Symmetric Techniques Virtual Lab (STVL), a working group for ECRYPT established the ECRYPT Stream Cipher Project (eSTREAM), to call for papers on new stream ciphers [2]. Finally, 34 candidates were submitted to eSTREAM, which will complete 2 evaluation phases for those candidates by January 2008.

Stream ciphers submitted to eSTREAM include SOSEMANUK, which is proposed by Berbain et al. and features variable secret key length from 128-bit up to 256-bit and 128-bit initial vector [4]. The cipher allows faster software implementation, since its basic operation is performed in a unit of 32 bits i.e. "word", to generate keystreams. The structure and the name of SOSEMANUK are based on SNOW 2.0 stream cipher [6] and Serpent block cipher [5]. Berbain et al. assert that SOSEMANUK has overcome the vulnerability of SNOW 2.0, while reducing it in internal state size. However, they also assert that SOSEMANUK guarantees up to 128-bit security, regardless of the secret key length.

According to the evaluation made by designers of SOSEMANUK, with 2^{256} computations or less, guess-and-determine attack can not be made on the cipher. However, this paper reports that the attack can be made on it, with less computations than the necessary computational effort that those designers claim. This attack can recover all of 384 bits of internal state just after the initialization. The amount of data required for this attack is only about 2^4 words, which attackers can easily collect. The needed amount of computation is approximately 2^{224}. Thus, when secret key length is longer than 224-bit, it needs less computational effort than an exhaustive key search, to break SOSEMANUK.

Section 2 describes the structure of SOSEMANUK stream cipher and Section 3 is about how to make guess-and-determine attack against SOSEMANUK. Section 4 considers the structural vulnerability of SOSEMANUK and the countermeasure to the attack. Section 5 concludes this paper.

2 Description of SOSEMANUK

This section describes the structure of SOSEMANUK stream cipher. Since it employs the techniques originally used for Serpent, explanation on Serpent and its derivatives is given for the first, and then that on SOSEMANUK.

2.1 Serpent and Derivatives

Serpent [5] is the block cipher proposed by Biham et al. in 1998, and one of AES candidates. Serpent performs the operation called bit-slice to divide 32-bit 4-word data. Divided data are mixed and then, reunited into 32-bit 4-word data. SOSEMANUK defines two functions, *Serpent1* and *Serpent24*, as its derivatives.

Serpent1 is the round function of Serpent, with neither subkey addition by bitwise exclusive OR nor linear transformation. 8 distinct S-boxes $(S_0, \cdots, S_7)$ are used for Serpent, while *Serpent1* uses S_2 only. *Serpent1* performs bit-slice to

divide 32-bit 4-word data, mix the divided data using S_2, and reunite them into 32-bit 4-word data, which is used as an output data.

Under full round, Serpent takes 32 round, and *Serpent24* is a reduced function of Serpent, which takes 24 rounds. Note that *Serpent24* inserts the 25th subkey after performing a linear transformation in the round function at the 24th round. Thus, *Serpent24* uses twenty-five 128-bit subkeys.

2.2 Keystream Generation

This subsection explains the keystream generation of SOSEMANUK, which can be grouped under roughly 3 parts; Linear Feedback Shift Register (LFSR), Finite State Machine (FSM), and Output Transformation.

LFSR consists of ten 32-bit registers, and is defined by the feedback polynomial over $GF(2^{32})$ as follows;

$$\pi(X) = \alpha X^{10} + \alpha^{-1} X^7 + X + 1$$

Here, α is a root of primitive polynomial $P(X)$ over $GF(2^8)$.

$$P(X) = X^4 + \beta^{23} X^3 + \beta^{245} X^2 + \beta^{48} X + \beta^{239}$$

β is a root of primitive polynomial $Q(X)$ over $GF(2)$.

$$Q(X) = X^8 + X^7 + X^5 + X^3 + 1$$

Since LFSR consists of primitive polynomial over $GF(2^{32})$, its 32-bit output sequence $\{s_t\}$ offers the maximal length cycle of $2^{320} - 1$.

FSM consists of two 32-bit registers. $(R1_t, R2_t)$ denotes the FSM registers at the given time t ($t \geq 1$). With the equations described below, FSM updates registers $(R1_t, R2_t)$ and generates 32-bit output f_t. Hereafter, $\oplus$ denotes bit-wise exclusive OR, whereas $+$ and $\times$ mean addition and multiplication over mod 2^{32}.

$$\begin{aligned} R1_t &= (R2_{t-1} + mux(lsb(R1_{t-1}), s_{t+1}, s_{t+1} \oplus s_{t+8})) \\ R2_t &= Trans(R1_{t-1}) \\ f_t &= (s_{t+9} + R1_t) \oplus R2_t \end{aligned}$$

Here, $lsb(x)$ means the least significant bit of data x, and $mux(c, x, y)$ means the function where x is output, if $c = 0$, while y is output, if $c = 1$. The function $Trans(x)$ is defined as follows;

$$Trans(x) = (M \times x) \lll 7$$

Here, constant $M = 0x54655307$, and $x \lll 7$ denotes that 32-bit data x is 7-bit rotated to the left (towards the most significant bit). Figure 1 is an overview of SOSEMANUK.

Using FSM output f_t and LFSR register s_t, Output Transformation generates keystreams. z_t represents the keystream at given time t ($t \geq 1$). Output Transformation processes 32-bit 4-word data, i.e. the data for 4 different ts at a time, and uses the equation below to generate the keystream for 4 different ts $(z_{t+3}, z_{t+2}, z_{t+1}, z_t)$;

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = \mathit{Serpent1}(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t)$$

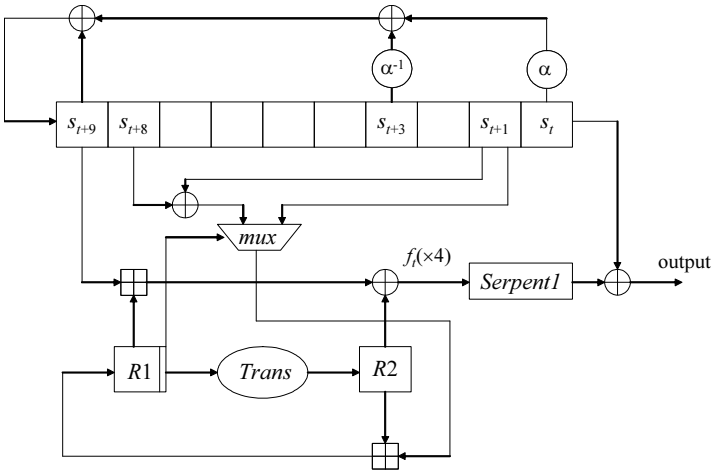


Fig. 1. An overview of SOSEMANUK

2.3 Initialization

This subsection describes the initialization of SOSEMANUK. As initialization, SOSEMANUK generates the initial value of internal state, using the key schedule of Serpent and *Serpent24*. Figure 2 illustrates the initialization of SOSEMANUK.

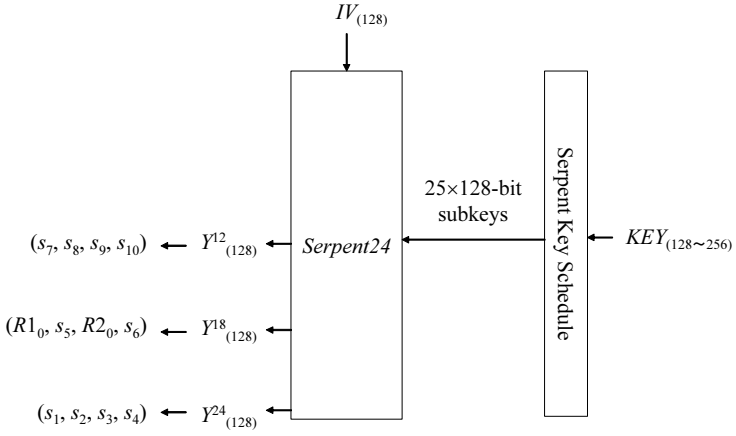


Fig. 2. Initialization of SOSEMANUK

SOSEMANUK takes secret key KEY as an input to key schedule of Serpent, to generate twenty-five 128-bit subkeys. Though the secret key of SOSEMANUK is variable, ranging from 128-bit up to 256-bit, Serpent's secret key is also variable, ranging from 1-bit from 256-bit. Thus, key scheduler can be operated in

accordance with the secret key length. After the subkey generation, initial vector IV is taken as an input to *Serpent24*. Then, intermediate data of rounds 12 and 18 of *Serpent24*, and output data of round 24 of *Serpent24* are used as initial values of internal state. Providing that Y^{12}, Y^{18}, and Y^{24} denote the outputs of rounds 12, 18, and 24, respectively, these three data are substituted in the equations below, as the initial values of their respective registers. Here, LFSR register and FSM register at the completion of initialization are represented by $(s_{10}, s_9, \cdots, s_1)$ and $(R1_0, R2_0)$, respectively.

$$\begin{aligned} s_7 \parallel s_8 \parallel s_9 \parallel s_{10} &= Y^{12} \\ R1_0 \parallel s_5 \parallel R2_0 \parallel s_6 &= Y^{18} \\ s_1 \parallel s_2 \parallel s_3 \parallel s_4 &= Y^{24} \end{aligned}$$

3 Cryptanalysis of SOSEMANUK

This subsection explains how to apply guess-and-determine attack against SOSEMANUK whose secret key size is 256-bit. Followings are the preconditions for the attack;

- Fixed secret key value during the attack.
- Attackers can obtain some quantity of keystream.

Assume that time t satisfies the assumption given below, when guess-and-determine attack is made against SOSEMANUK.

Assumption : $lsb(R1_{t-1}) = 0$

If the Assumption is satisfied, register $R1_t$ is updated with the following equation;

$$R1_t = R2_{t-1} + s_{t+1}$$

As is apparent from the equation given above, $R1_t$ is not influenced by s_{t+8}, when the Assumption is satisfied. Thus, it can be used for cryptanalysis.

Assuming that $t = 1$ satisfies the Assumption, guess the internal state just after the initialization described below.

Guess 1 : $s_1, s_2, s_3, s_4, R1_0, R2_0$

Since $lsb(R1_0) = 0$, from Assumption, 191 bits need to be guessed. Eqs. (1) and (2) are the ones to update registers $(R1_1, R2_1)$, respectively, where $t = 1$, while Eqs. (3) and (4) are used to update register s_{11}, and to generate FSM output f_1.

$$R1_1 = (R2_0 + s_2) \tag{1}$$
$$R2_1 = Trans(R1_0) \tag{2}$$
$$s_{11} = s_{10} \oplus \alpha^{-1}(s_4) \oplus \alpha(s_1) \tag{3}$$
$$f_1 = (s_{10} + R1_1) \oplus R2_1 \tag{4}$$

Based on Guess 1, the values of registers $R1_1$ and $R2_1$ can be calculated. (f_1, f_2, f_3, f_4), FSM outputs for 4 consecutive times can also be obtained, using the equation of Output Transformation, where the inverse function of *Serpent1* is represented by $Serpent1^{-1}$.

$$(f_4, f_3, f_2, f_1) = Serpent1^{-1}(z_4 \oplus s_4, z_3 \oplus s_3, z_2 \oplus s_2, z_1 \oplus s_1)$$

Thus, s_{10} and s_{11} can be determined, using Eqs. (4) and (3), respectively. In this way, guessing a part of internal state allows determining remaining undetermined part of internal state. Figure 3 shows how the cryptanalysis is performed, where $t = 1$.

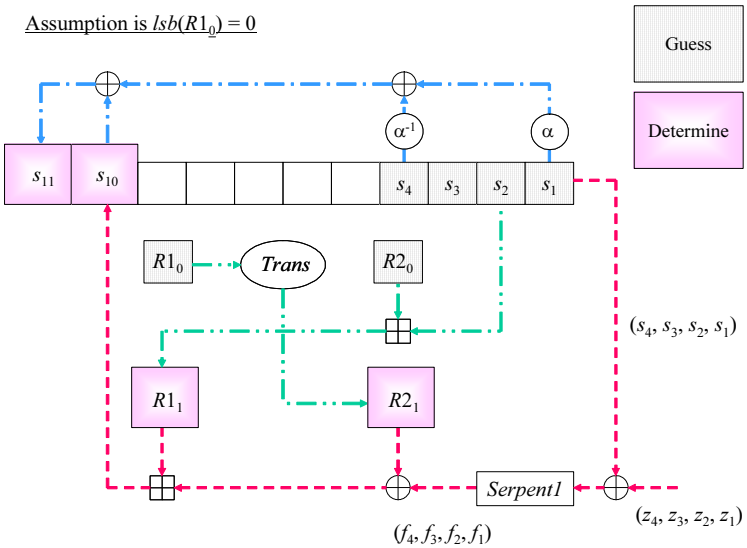


Fig. 3. Cryptanalysis steps where $t = 1$

Then, calculate $(R1_2, R2_2)$ where $t = 2$, using the equation to update FSM. Now, pay attention to the equation to generate FSM output f_2.

$$f_2 = (s_{11} + R1_2) \oplus R2_2 \tag{5}$$

Here, s_{11} is known data determined where $t = 1$. As f_2, $R1_2$, and $R2_2$ are also the data determined by Guess 1, any errors in Guess 1 must result in contradiction in Eq. (5). Thus, if any contradiction is found in Eq. (5), it means some error in Guess 1. Then, attackers can drop the candidate data. These steps narrow down the candidates for Guess 1 to about 2^{-32} of its original number.

Similarly, the data for $t = 3$ can be determined. Using the equation to update FSM, calculate $R1_3$, and $R2_3$. With equation to generate FSM output f_3, it is

possible to calculate s_{12}. Now, substituting s_{12} into equation to update LFSR, where $t = 2$, it becomes as follows;

$$s_5 = \alpha(s_{12} \oplus s_{11} \oplus \alpha(s_2))$$

Thus, s_5 can be determined, too.

Taking similar steps to above allows to determine unknown internal state where $t = 4$, and then, $R1_4$, $R2_4$, s_6, and s_{13} can be calculated. So far, 191 bits needed to be guessed, and candidates for those were narrowed down to about 2^{159}

Take similar steps to determine the internal state where $t = 5$ through 8. Since s_5, and s_6 are known data where $t = 1$ through 4, Guess 2 given below is to be done.

Guess 2 : s_7, s_8

Beside the bits determined already, 64 bits need to be guessed. With Guess 2, (f_5, f_6, f_7, and f_8) can be determined, using Output Transformation equation. Also, in the course of determining the internal state in a similar way, attackers can check to see if there is any contradiction in Guess 1 and/or Guess 2 at 3 times, using the equation to generate FSM output f_t. This step narrow down the number of candidates to around 2^{-96} of its original number. Thus, attackers have guessed 223 bits so far, and the candidates for them can be narrowed down to 2^{127}. Consecutive registers of internal state, i.e. $R1_t$, $R2_t$ ($t = 1, \ldots, 8$) and s_t ($t = 1, \ldots, 18$) can also be determined.

Take similar steps to determine the internal state where $t = 9$ through 12. Since s_9, s_{10}, s_{11}, and s_{12} are known data where $t = 1$ through 8 no more guess has to be performed. As attackers can check to see if there is any contradiction at 4 times, using the equation to generate FSM output f_t, In the course of determining the internal state, guessed candidates at Guess 1 and Guess 2 can be narrowed down to about 2^{-128} of its original number. This means that theoretically, all the 384 bits of internal state just after the initialization can be determined uniquely. Table 1 lists determined registers at each t and the number of candidates to be narrowed down at Guess 1 and Guess 2.

Finally, the amount of computation required for this attack is estimated. At Guess 1 and Guess 2, 223 bits at most have to be guessed. The success probability of Assumption is the probability that the least significant bit of register $R1_0$ becomes 0. Thus, it becomes 1/2, if initialization of SOSEMANUK offers completely random transformation. Consequently, the amount of computation required for the cryptanalysis T is determined as follows; [1]

$$T = 2^{223} \times 2^1 = 2^{224}$$

[1] Large memory is not needed for this attack, because the candidates for the registers are tried one by one.

Table 1. Registers determined at t and number of candidates at Guess 1 and Guess 2

t	Registers to Guess	Determined Registers	Contradiction Check	Number of Candidates
1	$s_1, s_2, s_3, s_4, R1_0, R2_0$	$s_{10}, s_{11}, R1_1, R2_1$		2^{191}
2		$R1_2, R2_2$	$\surd$	2^{159}
3		$s_5, s_{12}, R1_3, R2_3$		
4		$s_6, s_{13}, R1_4, R2_4$		
5	s_7, s_8	$s_{14}, s_{15}, R1_5, R2_5$	$\surd$	2^{191}
6		$R1_6, R2_6$	$\surd$	2^{159}
7		$s_9, s_{16}, s_{17}, R1_7, R2_7$		
8		$s_{18}, R1_8, R2_8$	$\surd$	2^{127}
9		$s_{19}, R1_9, R2_9$	$\surd$	2^{95}
10		$s_{20}, R1_{10}, R2_{10}$	$\surd$	2^{63}
11		$s_{21}, R1_{11}, R2_{11}$	$\surd$	2^{31}
12		$s_{22}, R1_{12}, R2_{12}$	$\surd$	1

If Assumption is not satisfied at $t = 1$, the attack will fail. However, assuming that Assumption is satisfied at $t = 5$, that is shifted from $t = 1$ by 4 times, the similar attack seems to break the cipher, successfully.

Guess-and-determine attack needs to compare the keystream that attackers obtained with keystreams output by cipher for 12 times, in order to determine candidates for Guess 1 and Guess 2, uniquely. Thus, taking the probability that Assumption is satisfied at any given t into account, only about 2^4 word of keystream is enough for success of the attack.

4 Discussion

This section discusses the vulnerability in SOSEMANUK structure and countermeasures to guess-and-determine attack. Designers of SOSEMANUK claimed that they eliminated the weakness in SNOW 2.0 structure and reduced the internal state size, in order to increase the suitability of SOSEMANUK to be implemented on a processor of any kind. They also employ reduced version of Serpent block cipher, as initialization process in SOSEMANUK, to increase its security.

However, when SOSEMANUK uses a longer than 224-bit secret key, guess-and-determine attack could be made successfully, with less amount of computation than an exhaustive key search. This indicates that the security provisions for SOSEMANUK made against existing attacks were not enough. It is considered that guess-and-determine attack was made successfully, directly because of

– smaller internal state size

– less LFSR feedback polynomial taps

To increase the suitability to be implemented, internal state size of SOSEMANUK is 384 bits, rather smaller than 576 bits, that of SNOW 2.0. It is considered that this modification allows attackers guess most part of internal state by guessing less bit-size than secret key size. Thus, the change in internal state size helps attackers to apply guess-and-determine attack. It is also considered that more registers in LFSR feedback polynomial, which are responsible for data updating, perhaps provide SOSEMANUK with more security, because they increase the required amount of bits to be guessed. Though the designers regard using reduced version of Serpent for initialization as a refinement in SOSEMANUK, it does not seem to work, as far as the attack proposed in this paper concerns, because even completely random transformation as initialization has no effect on the attack.

Hereafter, countermeasures to the attack proposed in this paper is discussed. Countermeasures described below are suggested;

– Elongate the internal state size enough.
– Increase LFSR feedback polynomial taps in number.

As described earlier, taking two countermeasures makes it difficult for attackers to make guess-and-determine attack, since those countermeasures increase amount of computation that is required for cryptanalysis. The countermeasures described above merely aim to provide the cipher with more resistance to guess-and-determine attack. We have not studied how they work on other existing attacks. When making improvements to the cipher, application of the countermeasure must be examined, so that it may not reduce the resistance to each of existing attacks.

5 Conclusion

This paper describes the attack on SOSEMANUK proposed as an improved algorithm of SNOW 2.0 in 2005. It is true that those who proposed the cipher added more security to SOSEMANUK. However, we demonstrated that guess-and-determine attack can be made on SOSEMANUK. This attack method can determine all of 384-bit internal state just after the initialization, using only 2^4-word keystream, the amount of data that attackers can easily collect. This attack needs about 2^{224} computations. Thus, when secret key length is longer than 224-bit, it needs less computational effort than an exhaustive key search, to break SOSEMANUK.

Our way of applying guess-and-determine attack proposed in this paper breaks SOSEMANUK more efficiently than the designers of SOSEMANUK expected. However, note that our method does not break the cipher whose security level is 128-bit, with 2^{128} computations or less. Since this attack method requires very large amount of computation, it can not be a practical threatening to SOSEMANUK. But, in the terms of extremely little amount of data needed for the attack, it can be said a very strong attack.

This paper discusses not only the structural vulnerability of SOSEMANUK but also countermeasures to guess-and-determine attack. Designers of SOSEMANUK pursued the advantages in implementation, to reduce the internal state size. This, however, resulted in vulnerability to guess-and-determine attack. Judging from existing attacks against stream ciphers, the size of internal state is a critical point for the security. Consequently, in terms of security as well as implementation.

Acknowledgement

The authors would like to thank the anonymous referees of SASC 2006 whose helpful remarks that improved the paper. The authors also would like to thank Shunsuke Ando for his useful comments.

References

1. AES, the Advanced Encryption Standard, NIST, FIPS-197.
   Available at http://csrc.nist.gov/CryptoToolkit/aes/
2. eSTREAM, the ECRYPT Stream Cipher Project.
   Available at http://www.ecrypt.eu.org/stream/
3. NESSIE, the New European Schemes for Signatures, Integrity, and Encryption.
   Available at https://www.cosic.esat.kuleuven.ac.be/nessie/
4. C. Berbain et al.: "SOSEMANUK, a fast software-oriented stream cipher," *eSTREAM, the ECRYPT Stream Cipher Project*, Report 2005/027, 2005.
5. E. Biham, R. Anderson, and L. Knudsen: "Serpent: A New Block Cipher Proposal," *Fast Software Encryption, FSE 1998*, LNCS 1372, pp.222-238, Springer Verlag, 1998.
6. P. Ekdahl and T. Johansson: "A New Version of the Stream Cipher SNOW," *Selected Areas in Cryptography, SAC 2002*, LNCS 2295, pp.47-61, Springer Verlag, 2002.

Resynchronization Attack on WG and LEX

Hongjun Wu and Bart Preneel

Katholieke Universiteit Leuven, Dept. ESAT/COSIC
{wu.hongjun,bart.preneel}@esat.kuleuven.be

Abstract. WG and LEX are two stream ciphers submitted to eStream – ECRYPT stream cipher project. In this paper, we point out security flaws in the resynchronization of these two ciphers. The resynchronization of WG is vulnerable to the differential attack. For WG with 80-bit key and 80-bit IV, 48 bits of the secret key could be recovered with about $2^{31.3}$ chosen IVs . For each chosen IV, only the first four keystream bits are needed in the attack. The resynchronization of LEX is vulnerable to the slide attack. If a key is used with about $2^{60.8}$ random IVs, and 20,000 keystream bytes are generated from each IV, then the key of the strong version of LEX could be recovered easily with the slide attack. The resynchronization attack on WG and LEX shows that the block cipher related attacks are powerful in analyzing the non-linear resynchronization.

Keywords: cryptanalysis, stream cipher, resynchronization attack, differential attack, slide attack, WG, LEX

1 Introduction

For the research on stream cipher, the resynchronization is not studied as thoroughly as the keystream generation algorithm. Ten years ago, Daemen et al. analyzed the weakness related to the linear resynchronization with known output Boolean function [4]. Recently Golić and Morgari studied the problem related to the linear resynchronization with unknown output function [6]. However almost all the stream ciphers proposed recently use non-linear resynchronization, so the previous attacks on the linear resynchronization could no longer be applied. In this paper, we apply the differential attack and slide attack to stream ciphers with non-linear resynchronization. And it shows that the cryptanalysis techniques used to attack block ciphers are also useful in the analysis of non-linear resynchronization.

WG [10] and LEX [3] are two stream ciphers submitted to eStream, the ERYPT stream cipher project [5]. The keystream generation algorithms of WG and LEX are quite strong. The keystream generation of WG is based on the WG transformations which have excellent cryptographic properties [7]. The keystream generation of LEX is based on the Advanced Encryption Standard [9]. However, the resynchronization of WG and LEX are insecure. The resynchronization of WG is vulnerable to the differential attack [1] and that of LEX is vulnerable to the slide attack [2]. Breaking WG requires $2^{31.3}$ chosen IVs, and breaking the strong version of LEX requires about $2^{60.8}$ random IVs.

This paper is organized as follows. WG and LEX are introduced in Section 2. The differential attack on WG is given in Section 3, and the slide attack on LEX is given in Section 4. Section 5 concludes this paper.

2 Description of WG and LEX

WG and LEX are described in Subsection 2.1 and 2.2, respectively.

2.1 Stream cipher WG

WG is a hardware oriented stream cipher with key length up to 128 bits. And it supports IV size range from 32 bits to 128 bits. The main feature of the WG stream cipher is the use of the WG transformation to generate keystream from the LFSR.

Keystream Generation

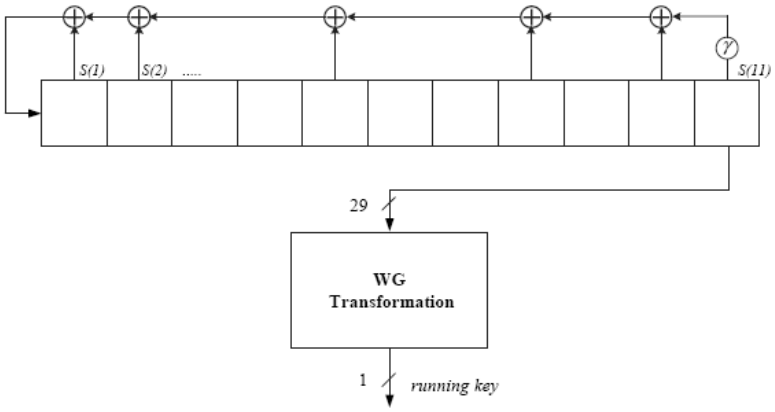


Fig. 1. Keystream Generation Diagram of WG [10]

The keystream generation diagram of WG is given in Fig. 1. WG has a regularly clocked LFSR which is defined by the feedback polynomial

$$p(x) = x^{11} + x^{10} + x^9 + x^6 + x^3 + x + \gamma \tag{1}$$

over $GF(2^{29})$, where $\gamma = \beta^{464730077}$ and β is the primitive root of $g(x)$

$$g(x) = x^{29} + x^{28} + x^{24} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + x^{14} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x + 1 \tag{2}$$

Then the non-linear WG transformation, $GF(2^{29}) \rightarrow GF(2)$, is applied to generate the keystream from the LFSR.

Resynchronization (Key/IV setup)

The key/IV setup of WG is given in Fig. 2. After the key and IV being loaded into LFSR, the LFSR is clocked 22 steps. During each of these 22 steps, 29 bits from the middle of the WG transformation are XORed to the feedback of LFSR, as shown in Fig. 2.

One step of the key/IV setup could be expressed as follows.

$$T = S(1) \oplus S(2) \oplus S(5) \oplus S(8) \oplus S(10) \oplus (\gamma \times S(11)) \oplus WG'(S(11))$$
$$S(i) = S(i-1) \text{ for } i = 11 \cdots 2;\ S(1) = T$$

where the $WG'(S(11))$ denotes the 29 bits extracted from the WG transformation, as shown in Fig. 2.

The WG cipher supports a number of key and IV sizes. The key size can be 80 bits, 96 bits, 112 bits and 128 bits. The IV sizes can be 32 bits, 64 bits, 80 bits, 96 bits, 112 bits, and 128 bits. Slightly different resynchronizations are used for different IV sizes. The details are given in Section 3.

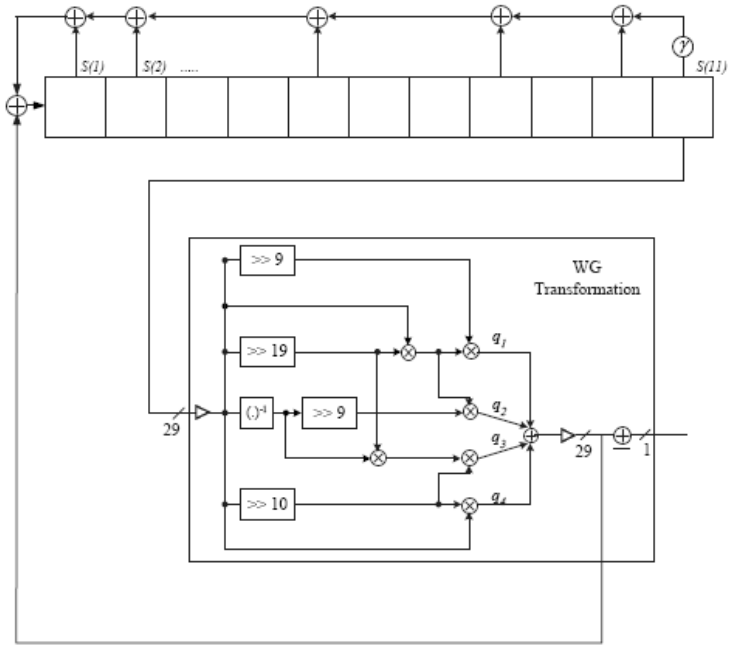


Fig. 2. Key/IV setup of WG [10]

2.2 Stream cipher LEX

LEX is based on block cipher AES. The keystream bits are generated by extracting 32 bits from each round of AES in the 128-bit Output Feedback (OFB) mode [8]. The LEX is about 2.5 times faster than AES. Fig. 3 shows how the AES is initialized and chained. First a standard AES key-schedule for a secret 128-bit key K is performed. Then a given 128-bit IV is encrypted by a single AES invocation: $S = AES_K(IV)$. The S and the subkeys are the output of the initialization process.

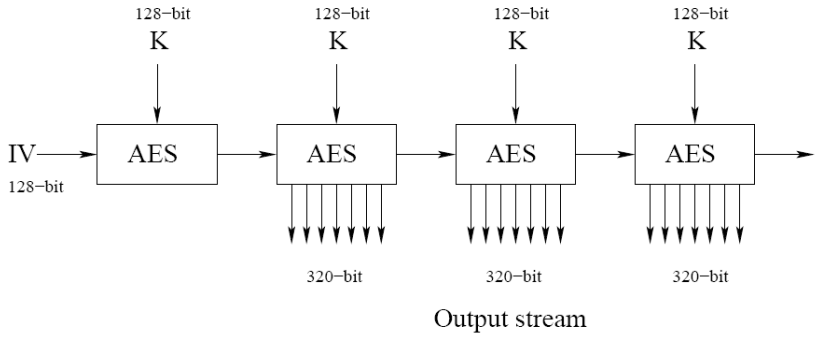


Fig. 3. Initialization and stream generation [3]

S is encrypted by K in the 128-bit OFB mode (for more secure variant, K is changed every 500 AES encryptions). At each round, 32 bits of the middle value of AES are extracted to form the keystream. The bytes $b_{0,0}$, $b_{0,2}$, $b_{2,0}$, $b_{2,2}$ at every odd round and the bytes $b_{0,1}$, $b_{0,3}$, $b_{2,1}$, $b_{2,3}$ at every even round are selected, as shown in Fig. 4.

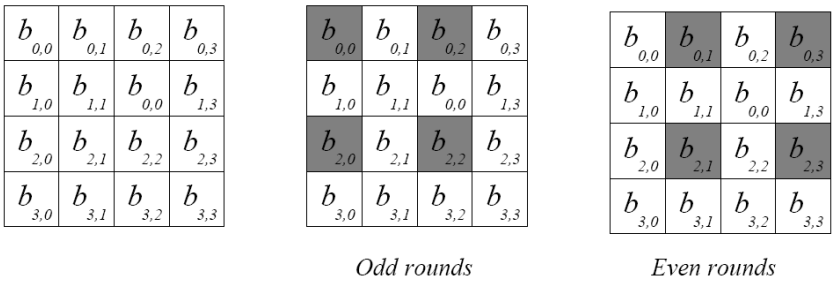


Fig. 4. The positions of leak in the even and odd rounds [3]

3 Differential Attack on the Resynchronization of WG

The resynchronization of WG could be broken with the chosen IV attack based on the differential cryptanalysis technique. WG with 32-bit IV size is not vulnerable to the attack given in this section (since no special differential could be introduced into this short IV). In Subsection 3.1 the attack is applied to break the WG with 80-bit key and 80-bit IV. The attacks on the WG with IV sizes larger than 80 bits are given in Subsection 3.2. The attack on the WG with 64-bit IV size is given in Subsection 3.3.

3.1 Attack on WG with 80-bit key and 80-bit IV

In this subsection, we will investigate the security of the key/IV setup of WG with 80-bit key and 80-bit IV. For this version of WG, denote the key as $K = k_1, k_2, k_3, \cdots, k_{80}$ and the IV as $IV = IV_1, IV_2, IV_3, \cdots, IV_{80}$. They are loaded into the LFSR as follows.

$$
\begin{array}{ll}
S_{1,\ldots,16}(1) = k_{1,\ldots,16} & S_{17,\ldots,24}(1) = IV_{1,\ldots,8} \\
S_{1,\ldots,8}(2) = k_{17,\ldots,24} & S_{9,\ldots,24}(2) = IV_{9,\ldots,24} \\
S_{1,\ldots,16}(3) = k_{25,\ldots,40} & S_{17,\ldots,24}(3) = IV_{25,\ldots,32} \\
S_{1,\ldots,8}(4) = k_{41,\ldots,48} & S_{9,\ldots,24}(4) = IV_{33,\ldots,48} \\
S_{1,\ldots,16}(5) = k_{49,\ldots,64} & S_{17,\ldots,24}(5) = IV_{49,\ldots,56} \\
S_{1,\ldots,8}(6) = k_{65,\ldots,72} & S_{9,\ldots,24}(6) = IV_{57,\ldots,72} \\
S_{1,\ldots,8}(7) = k_{73,\ldots,80} & S_{17,\ldots,24}(7) = IV_{73,\ldots,80}
\end{array}
$$

All the remaining bits of the LFSR are set to zero. Then the LFSR is clocked 22 steps with the middle value from the WG transformation being used in the feedback.

The chosen IV attack on WG goes as follows. For each secret key K, we choose two IVs, IV' and IV'', so that IV' and IV'' are identical at 8 bytes, but are different at two bytes: $IV'_{17,\ldots,24} \neq IV''_{17,\ldots,24}$ and $IV'_{49,\ldots,56} \neq IV''_{49,\ldots,56}$. The differences satisfy $IV'_{17,\ldots,24} \oplus IV''_{17,\ldots,24} = IV'_{49,\ldots,56} \oplus IV''_{49,\ldots,56}$.

Denote the $S(i)$ $(1 \leq i \leq 11)$ at the end of the j-th step as $S^j(i)$, and denote loading the key/IV as the 0th step. After loading the key and the chosen IV into LFSR, we know that the difference at $S(2)$ and $S(5)$ are the same, i.e., $S'^0(2) \oplus S''^0(2) = S'^0(5) \oplus S''^0(5)$. We denote this difference as $\triangle_1$, i.e., $\triangle_1 = S'^0(2) \oplus S''^0(2) = S'^0(5) \oplus S''^0(5)$.

We now examine the differential propagation during the 22 steps in the key/IV setup. The complete differential propagation is shown in Table 1, where the differences at the i-th step indicate the differences at the end of the i-th step. The difference $\triangle_2 = (\gamma \times S'^6(11) \oplus WG'(S'^6(11)) \oplus (\gamma \times S''^6(11) \oplus WG'(S''^6(11)) = (\gamma \times S'^0(5) \oplus WG'(S'^0(5)) \oplus (\gamma \times S''^0(5) \oplus WG'(S''^0(5))$. Similarly, we obtain that $\triangle_3 = (\gamma \times S'^0(2) \oplus WG'(S'^0(2)) \oplus (\gamma \times S''^0(2) \oplus WG'(S''^0(2))$.

From Table 1, we notice that at the end of the 22th step, the difference at $S^{22}(10)$ is $\triangle_2 \oplus \triangle_3$. From the above description of $\triangle_2$ and $\triangle_3$, we know that

$$
\begin{aligned}
\triangle_2 \oplus \triangle_3 = &((\gamma \times S'^0(5) \oplus WG'(S'^0(5)) \oplus (\gamma \times S''^0(5) \oplus WG'(S''^0(5))) \oplus \\
&((\gamma \times S'^0(2) \oplus WG'(S'^0(2)) \oplus (\gamma \times S''^0(2) \oplus WG'(S''^0(2))) \quad (3)
\end{aligned}
$$

It shows that the value of $\triangle_2 \oplus \triangle_3$ is determined by $k_{17,\ldots,24}$, $k_{49,\ldots,64}$, $IV'_{9,\ldots,24}$, $IV''_{49,\ldots,56}$, $IV''_{9,\ldots,24}$, $IV''_{49,\ldots,56}$.

From the keystream generation of WG, we notice that the first keystream bit is generated from $S^{22}(10)$ (after the key/IV setup, the LFSR is clocked, and the $S^{23}(11)$ is used to generate the first keystream bit). If $\triangle_2 \oplus \triangle_3 = 0$, then the first keystream bits for IV' and IV'' should be the same. This property is applied in the attack to determine whether the value of $\triangle_2 \oplus \triangle_3$ is 0.

Table 1. The differential propagation in the key/IV setup of WG

	S(1)	S(2)	S(3)	S(4)	S(5)	S(6)	S(7)	S(8)	S(9)	S(10)	S(11)
step 0	0	$\triangle_1$	0	0	$\triangle_1$	0	0	0	0	0	0
step 1	0	0	$\triangle_1$	0	0	$\triangle_1$	0	0	0	0	0
step 2	0	0	0	$\triangle_1$	0	0	$\triangle_1$	0	0	0	0
step 3	0	0	0	0	$\triangle_1$	0	0	$\triangle_1$	0	0	0
step 4	0	0	0	0	0	$\triangle_1$	0	0	$\triangle_1$	0	0
step 5	0	0	0	0	0	0	$\triangle_1$	0	0	$\triangle_1$	0
step 6	$\triangle_1$	0	0	0	0	0	0	$\triangle_1$	0	0	$\triangle_1$
step 7	$\triangle_2$	$\triangle_1$	0	0	0	0	0	0	$\triangle_1$	0	0
step 8	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0	0	0	0	$\triangle_1$	0
step 9	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0	0	0	0	$\triangle_1$
step 10	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0	0	0	0
step 11	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0	0	0
step 12	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0	0
step 13	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0	0
step 14	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0	0
step 15	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$	0
step 16	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$	$\triangle_1$
step 17	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$	$\triangle_2$
step 18	$\triangle_3 \oplus \triangle_4 \oplus \triangle_5$	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0	$\triangle_1 \oplus \triangle_2$
step 19	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3 \oplus \triangle_5 \oplus \triangle_6$	$\triangle_3 \oplus \triangle_4 \oplus \triangle_5$	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	0
step 20	$\triangle_4 \oplus \triangle_6$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3 \oplus \triangle_5 \oplus \triangle_6$	$\triangle_3 \oplus \triangle_4 \oplus \triangle_5$	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$
step 21	$\triangle_4 \oplus \triangle_5 \oplus \triangle_7$	$\triangle_4 \oplus \triangle_6$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3 \oplus \triangle_5 \oplus \triangle_6$	$\triangle_3 \oplus \triangle_4 \oplus \triangle_5$	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$	$\triangle_2 \oplus \triangle_3$
step 22	$\triangle_2 \oplus \triangle_3 \oplus \triangle_4 \oplus \triangle_5 \oplus \triangle_6 \oplus \triangle_7 \oplus \triangle_8$	$\triangle_4 \oplus \triangle_5 \oplus \triangle_7$	$\triangle_4 \oplus \triangle_6$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3 \oplus \triangle_5 \oplus \triangle_6$	$\triangle_3 \oplus \triangle_4 \oplus \triangle_5$	$\triangle_1 \oplus \triangle_4$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2 \oplus \triangle_3$	$\triangle_3$	$\triangle_2 \oplus \triangle_3$	$\triangle_1 \oplus \triangle_2$

Assume that the value of $\triangle_2 \oplus \triangle_3$ is randomly distributed, then $\triangle_2 \oplus \triangle_3 = 0$ with probability 2^{-29}. We thus need to generate about 2^{29} pairs $(\triangle_2, \triangle_3)$ in

order to obtain a pair satisfying $\triangle_2 \oplus \triangle_3 = 0$. Note that the key is fixed and that $S'^0(2) \oplus S''^0(2) = S'^0(5) \oplus S''^0(5)$ must be satisfied. There are 3 bytes of IV and one-byte difference can be chosen, so there are about $2^{24} \times 255/2 \approx 2^{31}$ pairs of $(\triangle_2, \triangle_3)$ are available. Thus there is no problem to generate 2^{29} pairs of $(\triangle_2, \triangle_3)$.

Then we proceed to determine which pair $(\triangle_2, \triangle_3)$ satisfies $\triangle_2 \oplus \triangle_3 = 0$. For each pair $(\triangle_2, \triangle_3)$, we modify the values of $IV'_{1,\ldots,8}$ and $IV''_{1,\ldots,8}$, but we ensure that $IV'_{1,\ldots,8} = IV''_{1,\ldots,8}$. This modification does not affect the value of $\triangle_2 \oplus \triangle_3$, but it effects the value of $S^{22}(10)$. We generate keystream and examine the first keystream bits. If the values of the first keystream bits are the same, then the chance that $\triangle_2 \oplus \triangle_3 = 0$ is improved. In that case, we modify the $IV'_{1,\ldots,8}$ and $IV''_{1,\ldots,8}$ again and observe the first keystream bits. This process ends when the first keystream bits are not the same or this process is repeated for 40 times. If one $(\triangle_2, \triangle_3)$ passes the test for 40 times, then we know that $\triangle_2 \oplus \triangle_3 = 0$ with probability extremely close to 1. (Each wrong pair could pass this filtering process with probability 2^{-40}. One pair of 2^{29} wrong pairs could pass this process with probability 2^{-11}.) Thus with about $2 \times 2^{29} \times \sum_{i=1}^{40} \frac{i}{2^i} = 2^{31}$ chosen IVs, we can find a pair $(\triangle_2, \triangle_3)$ satisfying $\triangle_2 \oplus \triangle_3 = 0$. Subsequently according to Eqn. (3) and $\triangle_2 \oplus \triangle_3 = 0$, we recover 24 bits of the secret key, $k_{17,\ldots,24}$ and $k_{49,\ldots,64}$.

The above attack can be improved if we consider the differences at $S^{22}(7)$ and $S^{22}(8)$. The differences there are both $\triangle_1 \oplus \triangle_2 \oplus \triangle_3$. If the value of $\triangle_1 \oplus \triangle_2 \oplus \triangle_3$ is 0, then the third and fourth bits of the two keystreams would be the same. If we only observe the third and fourth keystream bits, the $k_{17,\ldots,24}$ and $k_{49,\ldots,64}$ can be recovered with $2 \times 2^{29} \times \sum_{i=1}^{20} (\frac{1}{2^{i-1}} - \frac{1}{2^i}) \times i = 2^{30.4}$ chosen IVs.

In the attack, we observe the first, third and fourth keystream bits, then recovering $k_{17,\ldots,24}$ and $k_{49,\ldots,64}$ requires about $2 \times 2^{28} \times 2^{1.13} = 2^{30.1}$ chosen IVs (the value $2^{1.13}$ is obtained through numerical computation).

By setting the difference at $S^0(3)$ and $S^0(6)$ and observing the second and third bits of the keystream, we can recover another 24 bits of the secret key, $k_{25,\ldots,40}$ and $k_{65,\ldots,72}$. We need $2^{30.4}$ chosen IVs.

So with about $2^{30.1} + 2^{30.4} = 2^{31.3}$ chosen IVs, we can recover 48 bits of the 80-bit secret key. It shows that the key/IV setup of WG stream cipher is insecure.

3.2 Attacks on WG with key and IV sizes larger than 80 bits

The WG ciphers with the key and IV sizes larger than 80 bits are all vulnerable to the chosen IV attack. The attacks are very similar to the above attack. We omit the details of the attacks here. The results are given below.

1. For WG with 96-bit key and 96-bit IV, 48 bits of the key can be recovered.
2. For WG with 112-bit key and 112-bit IV, 72 bits of the key can be recovered.
3. For WG with 128-bit key and 128-bit IV, 72 bits or 96 bits of the key can be recovered.

3.3 Attacks on WG with 64-bit IV size

We use the WG with 80-bit key and 64-bit IV as an example to illustrate the attack. For the WG cipher with 80-bit key and 64-bit IV, the key and IV are loaded into the LFSR as follows:

$$
\begin{array}{ll}
S_{1,\ldots,16}(1) = k_{1,\ldots,16} & S_{1,\ldots,16}(2) = k_{17,\ldots,32} \\
S_{1,\ldots,16}(3) = k_{33,\ldots,48} & S_{1,\ldots,16}(4) = k_{49,\ldots,64} \\
S_{1,\ldots,16}(5) = k_{65,\ldots,80} & S_{1,\ldots,16}(9) = k_{1,\ldots,16} \\
S_{1,\ldots,16}(10) = k_{17,\ldots,32} \oplus 1 & S_{1,\ldots,16}(11) = k_{33,\ldots,48}
\end{array}
$$

$$
\begin{array}{ll}
S_{17,\ldots,24}(1) = IV_{1,\ldots,8} & S_{17,\ldots,24}(2) = IV_{9,\ldots,16} \\
S_{17,\ldots,24}(3) = IV_{17,\ldots,24} & S_{17,\ldots,24}(4) = IV_{25,\ldots,32} \\
S_{17,\ldots,24}(5) = IV_{33,\ldots,40} & S_{17,\ldots,24}(6) = IV_{41,\ldots,48} \\
S_{17,\ldots,24}(7) = IV_{49,\ldots,56} & S_{17,\ldots,24}(8) = IV_{57,\ldots,64}
\end{array}
$$

In the attack, we set the differences at $S(2)$ and $S(5)$, we can only generate about 2^{23} pairs of $(\triangle_2, \triangle_3)$ since we can only modify $IV_{9,\ldots,16}$ and $IV_{33,\ldots,40}$. Thus we can obtain a pair $(\triangle_2, \triangle_3)$ satisfying $\triangle_2 \oplus \triangle_3 = 0$ or $\triangle_1 \oplus \triangle_2 \oplus \triangle_3 = 0$ with probability 2^{-5}. Once we know $\triangle_2 \oplus \triangle_3 = 0$ or $\triangle_1 \oplus \triangle_2 \oplus \triangle_3 = 0$, we can recover 29-bit information of $k_{17,\ldots,32}$ and $k_{65,\ldots,80}$. It shows that 29-bit information of the secret key could be recovered with probability 2^{-5}. This attack requires about $2^{25.1}$ chosen IVs.

The attack on WG with 96-bit key and 64-bit IV is similar to the above attack. We can set the differences at $S(2)$ and $S(5)$ or at $S(3)$ and $S(6)$. In the attack 29-bit information of $k_{17,\ldots,32}$ and $k_{65,\ldots,80}$ can be recovered with probability 2^{-5}, and another 29-bit information of $k_{33,\ldots,48}$ and $k_{81,\ldots,96}$ can be recovered with probability 2^{-5}.

The attack on WG with 112-bit key and 64-bit IV is also similar. The result is that 29-bit information of $k_{17,\ldots,32}$ and $k_{65,\ldots,80}$ can be recovered with probability 2^{-5}, 29-bit information of $k_{33,\ldots,48}$ and $k_{81,\ldots,96}$ can be recovered with probability 2^{-5}, and 29-bit information of $k_{49,\ldots,64}$ and $k_{97,\ldots,112}$ can be recovered with probability 2^{-5}.

The attack on WG with 128-bit key and 64-bit IV is also similar. The result is that 29-bit information of $k_{17,\ldots,32}$ and $k_{65,\ldots,80}$ can be recovered with probability 2^{-5}, 29-bit information of $k_{33,\ldots,48}$ and $k_{81,\ldots,96}$ can be recovered with probability 2^{-5}, 29-bit information of $k_{49,\ldots,64}$ and $k_{97,\ldots,112}$ can be recovered with probability 2^{-5}, and 29-bit information of $k_{64,\ldots,80}$ and $k_{113,\ldots,128}$ can be recovered with probability 2^{-5}.

4 Slide Attack on the Resynchronization of LEX

The security of LEX depends heavily on the fact that only small amount of information is released for each round (including the input and output) of AES. The slide attack intends to retrieve all the information of one AES round input (or output) in LEX.

Denote $S_i = E_K^i(IV)$, where $E^i(m)$ means that m is encrypted by i times, $S_0 = IV$. And denote the 320 bits extracted from the i-th encryption as k_i for $i \geq 2$. For two IVs, IV' and IV'', if $k'_2 = k''_j$ $(j > 2)$, then we know that $S'_1 = S''_{j-1}$. Immediately, we know that $S''_{j-2} = S'_0 = IV'$. Note that k''_{j-1} are extracted from $E_K(S''_{j-2})$, so k''_{j-1} are extracted from $E_K(IV')$, it means that we know the input to AES, and we know 32 bits from the output of the first round. In the following, we show that it is easy to recover the secret key from this 32-bit information of the first round output.

Denote the 16-byte output of the r-th round of AES as $m^r_{i,j}$ $(0 \leq i, j \leq 3)$. And denote the 16-byte round key at the end of the r-th round as $w^r_{i,j}$ $(0 \leq i, j \leq 3)$. Now if $m^1_{0,0}$, $m^1_{0,2}$, $m^1_{2,0}$, $m^1_{2,2}$ are known, i.e, four bytes of the first round output are known, then we obtain the following four equations:

$$\begin{aligned}
m^1_{0,0} \oplus w^1_{0,0} = \text{MixColumn}((m^0_{0,0} \oplus w^0_{0,0})||(m^0_{1,3} \oplus w^0_{1,3}) \\
||(m^0_{2,2} \oplus w^0_{2,2})||(m^0_{3,1} \oplus w^0_{3,1}))\&0xFF \quad (4) \\
m^1_{2,0} \oplus w^1_{2,0} = (\text{MixColumn}((m^0_{0,0} \oplus w^0_{0,0})||(m^0_{1,3} \oplus w^0_{1,3}) \\
||(m^0_{2,2} \oplus w^0_{2,2})||(m^0_{3,1} \oplus w^0_{3,1})) >> 16)\&0xFF \quad (5) \\
m^1_{0,2} \oplus w^1_{0,2} = \text{MixColumn}((m^0_{0,2} \oplus w^0_{0,2})||(m^0_{1,1} \oplus w^0_{1,1}) \\
||(m^0_{2,0} \oplus w^0_{2,0})||(m^0_{3,3} \oplus w^0_{3,3}))\&0xFF \quad (6) \\
m^1_{2,2} \oplus w^1_{2,2} = (\text{MixColumn}((m^0_{0,2} \oplus w^0_{0,2})||(m^0_{1,1} \oplus w^0_{1,1}) \\
||(m^0_{2,0} \oplus w^0_{2,0})||(m^0_{3,3} \oplus w^0_{3,3})) >> 16)\&0xFF \quad (7)
\end{aligned}$$

Each equation leaks one-byte information of the secret key. In the above four equations, 12 bytes of the subkey are involved. To recover all those 12 bytes, we need three inputs to AES and the related 32-bit first round outputs so that we could obtain 12 equations. Those 12 equations can be solved with about $\alpha \times 2^{32}$ operations, where α is a small constant. With 96 bits of the key being recovered, the rest of the 32 bits of AES could be recovered by exhaustive search.

We now compute the number of IVs required to generate three collisions. Suppose that a secret key is used with about $2^{65.3}$ random IVs, and each IV^i is used to generate a 640-bit keystream k^i_2, k^i_3. Since the block size of AES is 128 bits, we know that with high chance there are three collisions $k^i_2 = k^j_3$ for different i and j since $\frac{2^{65.3} \times (2^{65.3}-1)}{2} \times 2^{-128} \approx 3$.

The number of IVs could be reduced if more keystream bits are generated from each IV. In [3], it is suggested to change the key every 500 AES encryptions for strong variant of LEX. Suppose that each IV is applied to generate 500 320-bit outputs, then with $2^{60.8}$ IVs, we could find three collisions $k^i_2 = k^j_x$ $(2 < x < 500)$ and recover the key of LEX. For the original version of LEX, the AES key is not changed during the keystream generation. Suppose that each IV is used to generate 2^{50} keysteam bytes, then the key could be recovered with about 2^{43} random IVs (here we need to consider that the state update function of LEX is reversible; otherwise, the amount of IV required in the attack could be greatly reduced).

5 Conclusion

In this paper, we show that the resynchronizations of WG and LEX are vulnerable to the differential cryptanalysis and the slide attack, respectively. It shows that the block cipher cryptanalysis techniques are powerful in analyzing the non-linear resynchronization of stream cipher.

References

1. E. Biham, A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems", in *Advances in Cryptology – Crypto'90*, LNCS 537, pp. 2-21, Springer-Verlag, 1991.
2. A. Biryukov and D. Wagner, "Slide Attacks", *Fast Soft Encryption – FSE'99*, LNCS 1636, pp. 245-259, Springer-Verlag, 1999.
3. A. Biryukov, "A New 128-bit Key Stream Cipher LEX", *ECRYPT Stream Cipher Project Report 2005/013*. Available at http://www.ecrypt.eu.org/stream/
4. J. Daemen, R. Govaerts, and J. Vandewalle, "Resynchronization weakness in synchronous stream ciphers", *Advances in Cryptology - EUROCRYPT'93*, Lecture Notes in Computer Science, vol. 765, pp. 159-167, 1994.
5. ECRYPT Stream Cipher Project, at http://www.ecrypt.eu.org/stream/
6. J. D. Golić and G. Morgari, "On the Resynchronization Attack", *Fast Software Encryption – FSE2003*, LNCS 2887, pp. 100-110, Springer-Verlag, 2003.
7. G. Gong, and A. Youssef. "Cryptographic Properties of the Welch-Gong Transformation Sequence Generators", *IEEE Transactions on Information Theory*, vol. 48, No. 11, pp. 2837-2846, Nov. 2002.
8. National Institute of Standards and Technology, "DES Modes of Operation", Federal Information Processing Standards Publication (FIPS) 81. Available at http://csrc.nist.gov/publications/fips/
9. National Institute of Standards and Technology, "ADVANCED ENCRYPTION STANDARD (AES) ", Federal Information Processing Standards Publication (FIPS) 197. Available at http://csrc.nist.gov/publications/fips/
10. Y. Nawaz, G. Gong. "The WG Stream Cipher". *ECRYPT Stream Cipher Project Report 2005/033*. Available at http://www.ecrypt.eu.org/stream/

Chosen Ciphertext Attack on SSS

Joan Daemen[1], Joseph Lano[2] *, and Bart Preneel[2]

[1] STMicroelectronics Belgium
joan.daemen@st.com
[2] Katholieke Universiteit Leuven, Dept. ESAT/SCD-COSIC
{joseph.lano,bart.preneel}@esat.kuleuven.ac.be

Abstract. The stream cipher Self-Synchronizing Sober (SSS) is a candidate in the ECRYPT stream cipher competition. In this paper, we describe a chosen ciphertext attack on SSS. Our implementation of the attack recovers the entire secret state of SSS in around 10 seconds on a 2.8 GHz PC, and requires a single chosen ciphertext of less than 10 kByte. The designers of SSS state that chosen ciphertext attacks were considered to fall outside of the threat model. Hence the relevance of such attacks is also discussed in this paper.

1 Introduction

In the ECRYPT Stream Cipher Project [6], 34 stream cipher primitives have been submitted for evaluation. Of these 34 proposals, 31 are synchronous, 2 are self-synchronizing, and one design, Phelix, is neither synchronous nor self-synchronizing. This division reflects the fact that synchronous stream ciphers have been more widely studied in the past years.

In a synchronous stream cipher, the internal state of the stream cipher is independent of the plaintext and ciphertext. Hence the only relevant attack model is the known plaintext attack. Also, the attacker can influence the internal state through a resynchronization attack with chosen or known IV [3, 1]. A strong resynchronization mechanism is therefore needed to prevent such attacks.

Another attack model applies to the self-synchronizing stream ciphers, where the ciphertext needs to enter the state to ensure the self-synchronization property. This makes chosen plaintext (at encryption) and chosen ciphertext (at decryption) attacks interesting. Because of this property, the design and analysis of self-synchronizing stream ciphers is much closer related to the field of block ciphers than to the field of synchronous stream ciphers [4]. Note that the same applies to the design of

* Research financed by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

a good resynchronization mechanism for synchronous stream ciphers, as evidenced by several ECRYPT candidates.

In this paper, we describe such a chosen ciphertext attack on the ECRYPT candidate Self-Synchronizing SOBER (SSS) [8]. We also implemented the attack in C. Our implementation recovers the secret key with a chosen ciphertext of around 10 kByte and runs in 10 seconds on a 2.8 GHz PC.

The designers of SSS state that chosen ciphertext attacks were considered to fall outside of the security model. However, chosen ciphertext attacks have previously been considered when evaluating the security of self-synchronizing stream ciphers. Self-synchronizing stream encryption with DES in CFB mode was analyzed with respect to chosen ciphertext attacks in [7]. The stream cipher KNOT [2] has been broken by differential attacks using chosen ciphertext in [5]. The other self-synchronizing ECRYPT stream cipher candidate is MOSQUITO, the successor of KNOT. In the paper on MOSQUITO [4], the security analysis is mainly devoted to differential and linear attacks using chosen ciphertext. We will give arguments for the importance of this type of attacks in this paper.

The outline of this paper is as follows. A brief explanation on self-synchronizing stream ciphers is given in Sect. 2 and the design of SSS is briefly presented in Sect. 3. A chosen ciphertext attack on SSS is described in Sect. 4, and the relevance of such an attack on self-synchronizing stream ciphers is discussed in Sect. 5. The paper concludes in Sect. 6.

2 Self-Synchronizing Stream Ciphers

A simplified representation of a self-synchronizing stream cipher is given in Fig. 1. In such a design, the next key stream bit z_t is fully determined by the last n_m ciphertext bits and the cipher key K. This can be modelled as the key stream symbol being computed by a keyed cipher function f_c operating on a shift register that contains the last n_m ciphertexts. This conceptual model can be implemented in various ways, with the design of SSS described in Sect. 3 as an example.

For the first n_m plaintext or ciphertext symbols, the previous n_m ciphertexts do not exist. Hence the self-synchronizing stream cipher must be initialized by loading n_m *dummy* ciphertext symbols, called the initialization vector IV.

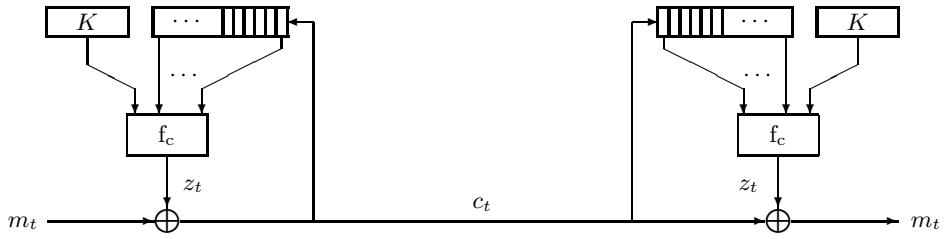


Fig. 1. Self-synchronizing stream encryption.

3 Brief Description of SSS

We only describe the aspects of the design that are relevant for the analysis performed in this article. For a complete description of the design, including the initialization and authentication mechanism, we refer to [8].

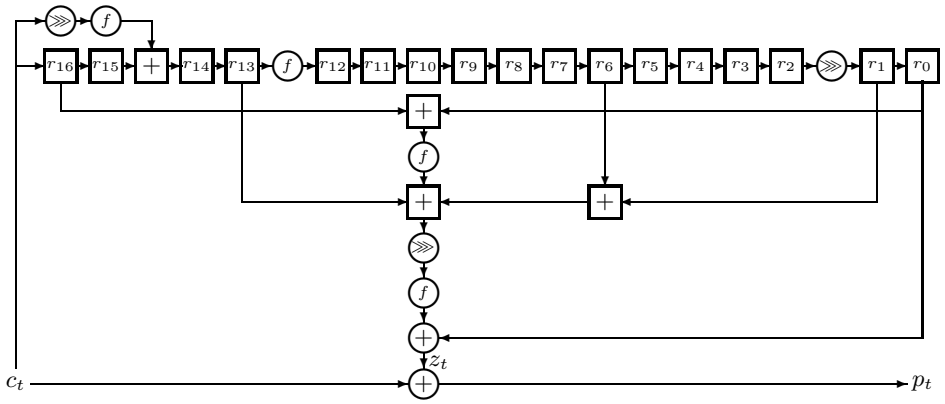


Fig. 2. Layout of SSS at the decryption side

A layout of SSS at the decryption side can be found in Fig. 2. In this figure, $\oplus$ represents exclusive or, $\boxplus$ represents addition and $\ggg$ represents a rotation by 8 positions to the right (or byteswap). The internal state of SSS consists of a 17-word shift register $r_0, \ldots, r_{16}$, where each word is 16 bits in size. The main building block is the key-dependent function f, which can be seen as a key-dependent permutation of a 16-bit word. The function f is built as follows:

$$f(x) = SBox(x_H) \oplus x\,, \tag{1}$$

where x_H stands for the Most Significant Byte (MSB) of x and where SBox is a key-dependent substitution box from 8 to 16 bits determined at key setup. In the rest of the paper we will assume that this SBox is a random unknown table of 256 16-bit words.

4 The Chosen Ciphertext Attack on SSS

From the description of SSS follows that its secret key consists of a table of 256 values of 16 bits. The aim of our attack is to recover this secret table.

We are going to decrypt a single ciphertext string that consists of a succession of 263 similar patterns and obtain the corresponding plaintext (and hence the key stream). The pattern i ($i = 0, 1, 2, \ldots, 263$) consists of 18 16-bit words and always has the following format:

$$\begin{cases} c_t^i = 0 \text{ for } t = 0, \ldots, 12, 14, \ldots 18 \\ c_{13}^i = b^i, \end{cases} \tag{2}$$

where b^i takes some value in each pattern, to be determined as explained below. Note that the values that we have chosen to be 0 could take any value for our attack to work, as long as they are constant across all patterns.

When generating key stream word z_{18}^i, we can see from Fig. 2 that the following words are needed[3]:

$$z_{18}^i = f((f(r[0] + r[16]) + r[1] + r[6] + r[13]) \ggg 8) \oplus r[0]\,. \tag{3}$$

It is easy to derive that these registers have the following content at $t = 18$:

$$\begin{cases} r[0] = f^2(0) \ggg 8 \\ r[1] = f^2(0) \ggg 8 \\ r[6] = f^2(0) \\ r[13] = f(0) + b^i \\ r[16] = 0\,. \end{cases} \tag{4}$$

In other words, all these registers are constant for each pattern i except for register $r[13]$. We can hence regroup all the constants inside $f()$ of (3) into a single (yet unknown) constant a as follows:

$$\begin{aligned} a &= f(r[0] + r[16]) + r[1] + r[6] + f(0) \\ &= f(f^2(0) \ggg 8) + (f^2(0) \ggg 8) + f^2(0) + f(0)\,, \end{aligned} \tag{5}$$

[3] At first sight, one may think that this should be z_{17}, but the designers have built a delay into their design, as can be deducted from the source code, which can be obtained at [8].

and then (3) simplifies to:

$$z_{18}^i = f((a + b^i) \ggg 8) \oplus r_0\,. \tag{6}$$

We use the notation r_0 to indicate that the content of $r[0]$ is also constant for all patterns. a is a two-byte word and we denote its MSB byte by a_H and its LSB byte by a_L. In the same way we split b^i in its two bytes b_H^i and b_L^i.

In a first phase, we will determine the 7 least significant bits of a_H. We will not be able to recover the most significant bit of a_H, but this is not a problem as the value of this bit is irrelevant to our analysis. To recover these 7 bits, we need 8 patterns of the type described above with $b_L^i = 0$, $b_H^0 = 0$ and $b_H^i = 2^{i-1}$ for $i = 1, 2, \ldots, 7$.

We now rewrite the above equation by splitting up the f function and some words of interest to get:

$$z_{18}^i = SBox(a_L) \oplus (2^8 \cdot a_L + a_H + 2^{i-1}) \oplus r_0\,, \tag{7}$$

By XORing the above equation for each $i \neq 0$ with the equation for $i = 0$ and eliminating terms we obtain:

$$z_{18,L}^i \oplus z_{18,L}^0 = a_H \oplus (a_H + 2^{i-1})\,. \tag{8}$$

In these equations a_H is the only unknown, and we can easily deduce its 7 least significant bits from the above equations bit by bit: A difference in $v_{18,L}^i \oplus v_{18,L}^0$ equal to 2^{i-1} implies that the corresponding bit of a_H is 0, otherwise it is 1.

Now that the relevant bits of a_H have been recovered, we will try to extract the entire secret $SBox$ table in the second phase of our attack. In short, this phase operates as follows. First we guess the value of a_L and $SBox(a_L)$ (24 bits in total). Then we reconstruct the remaining 255 entries of $SBox$ using key stream symbols $z_{18,L}^j$ obtained from decrypting 256 patterns. We then use this value to decrypt some ciphertext from the above patterns and check whether the plaintext matches. We now describe this reconstruction phase into more detail.

Our ciphertext contains 256 patterns that have $b_L^j = j$ and $b_H^j = 0$ for $j = 0, 1, \ldots, 255$. We obtain the following equations, again after XORing with the equation for $j = 0$:

$$z_{18}^j \oplus z_{18}^0 = \begin{array}{l} SBox(a_L) \oplus SBox(a_L + j) \oplus \\ (((2^8 \cdot a_H + a_L) \oplus (2^8 \cdot a_H + a_L + j)) \ggg 8)\,. \end{array} \tag{9}$$

Assuming a guess for a_L and $SBox(a_L)$ we can deduce $SBox(a_L + j)$ from this equation:

$$SBox(a_L + j) = \begin{array}{l} z_{18}^{j} \oplus z_{18}^{0} \oplus SBox(a_L) \oplus \\ (((2^8 \cdot a_H + a_L) \oplus (2^8 \cdot a_H + a_L + j)) \ggg 8)\,. \end{array} \quad (10)$$

Because j takes all 255 nonzero values we recover the entire SBox. We then verify whether, for the current guess of a_L and of $SBox(a_L)$ and the deduced $SBox$, the ciphertext decrypts to the corresponding plaintext. If it does, we have found the entire secret key.

We have implemented this attack in C. It recovers the entire secret key in on average 10 seconds on a 2.8 GHz Pentium IV PC running gcc under Linux. The single chosen ciphertext consists of 263 patterns of 36 bytes (note that the patterns for $i = 0$ and for $j = 0$ are the same), or 9468 byte in total. It is possible to reduce the data complexity even further by overlapping the patterns.

5 On the Relevance of Chosen Ciphertext Attacks

In the security claims for SSS [8], the authors state that they did not consider chosen ciphertext attacks in their threat model. One of the security requirements for their design is that "the result of decrypting altered ciphertext is not made available to the attacker". They motivate this requirement as follows: "This should be a standard requirement for any self-synchronizing stream cipher, since the attacker has complete control over the state of the cipher." However, it seems to be logical to us to include chosen ciphertext attacks in the security model of a self-synchronizing stream cipher, both from a theoretical as from a practical perspective.

From a theoretical perspective, a self-synchronizing stream cipher is functionally equivalent to a block cipher used in CFB mode. Chosen ciphertext attacks do apply on this mode of operation of a block cipher, an example is a chosen ciphertext attack on DES in CFB mode [7]. To enable a fair comparison of primitives aiming at the same applications, we believe that a uniform threat model should apply.

From a practical perspective, we see several scenarios where chosen ciphertext attacks can apply, just like with block ciphers. Preventing such an attack would require authenticating the plaintext before it is released. This suffers from two serious problems. First, buffering and secure storage of large amounts of texts is necessary, and this is impractical in several environments of interest. Second, this authentication requirement is orthogonal to the concept of self-synchronization: we do not see the point

of designing self-synchronizing stream ciphers when transmission errors are not allowed.

Another remark is that a self-synchronizing stream cipher resistant to chosen ciphertext attacks will result in a more elegant design. No special IV loading mechanism will be necessary as in SSS; loading a nonce into the state will be sufficient to start encryption and decryption.

6 Conclusion

In this note, we have described an attack on the ECRYPT candidate SSS, a self-synchronizing stream cipher. Our attack recovers the secret key of the design with a single chosen ciphertext of less than 10 kByte in about ten seconds on a modern PC. We believe that our attack is a practical attack on SSS. SSS is hence insecure and should not be used.

References

1. Frederik Armknecht, Joseph Lano, and Bart Preneel. Extending the resynchronization attack. In Helena Handschuh and Anwar Hasan, editors, *Selected Areas in Cryptography, SAC 2004*, number 3357 in Lecture Notes in Computer Science, pages 19–38. Springer-Verlag, 2004.
2. Joan Daemen, Rene Govaerts, and Joos Vandewalle. A practical approach to the design of high speed self-synchronizing stream ciphers. In O. Hirota and P. Y. Kam, editors, *Singapore ICCS/ISITA '92*, pages 279–293. IEEE, 1992.
3. Joan Daemen, Rene Govaerts, and Joos Vandewalle. Resynchronization weaknesses in synchronous stream ciphers. In T. Helleseth, editor, *Advances in Cryptology - EUROCRYPT 1993*, number 765 in Lecture Notes in Computer Science, pages 159–167. Springer-Verlag, 1993.
4. Joan Daemen and Paris Kitsos. Submission to ECRYPT call for stream ciphers: the self-synchronizing stream cipher MOSQUITO. ECRYPT Stream Cipher Project Report 2005/018, 2005. http://www.ecrypt.eu.org/stream.
5. Antoine Joux and Frederic Muller. Loosening the KNOT. In Thomas Johansson, editor, *Fast Software Encryption, FSE 2003*, number 2887 in LNCS, pages 87–99. Springer, 2003.
6. ECRYPT Network of Excellence in Cryptology. ECRYPT stream cipher project, 2005. http://www.ecrypt.eu.org/stream/.
7. Bart Preneel, Marnix Nuttin, Vincent Rijmen, and Johan Buelens:. Cryptanalysis of the CFB mode of the DES with a reduced number of rounds. In D.R. Stinson, editor, *Advances in Cryptology - CRYPTO 1993*, number 773 in Lecture Notes in Computer Science, pages 212–223. Springer-Verlag, 1994.
8. Gregory Rose, Philip Hawkes, Michael Paddon, and Miriam Wiggers de Vries. Primitive specification for SSS. ECRYPT Stream Cipher Project Report 2005/028, 2005. http://www.ecrypt.eu.org/stream.

Improved cryptanalysis of Py

Paul Crowley, paul@ciphergoth.org

LShift Ltd, www.lshift.net

Abstract We improve on the best known cryptanalysis of the stream cipher Py by using a hidden Markov model for the carry bits in addition operations where a certain distinguishing event takes place, and constructing from it an "optimal distinguisher" for the bias in the output bits which makes more use of the information available. We provide a general means to efficiently measure the efficacy of such a hidden Markov model based distinguisher, and show that our attack improves on the previous distinguisher by a factor of 2^{16} in the number of samples needed. Given 2^{72} bytes of output we can distinguish Py from random with advantage greater than $\frac{1}{2}$, or given only a single stream of 2^{64} bytes we have advantage 0.03.
Keywords: Py, symmetric cryptanalysis, hidden Markov model

1 Introduction

Py [2] is a candidate in the eSTREAM project to identify new stream ciphers that might be suitable for widespread adoption. It is a synchronous stream cipher with a 1300-byte internal state, and at each step produces eight bytes of output, organised as two four-byte words. Py is one of the fastest eSTREAM candidates in software.

[6] presents a distinguisher against this cipher. It defines an event L in the internal state of the cipher which occurs with probability roughly $2^{-41.91}$. When this event occurs, two output values can be guaranteed to be equal. This results in a very small linear bias in the output of Py, which can be detected with on the order of $\Pr[L]^{-2}$ samples.

Specifically, when the event occurs, two output words $O_{1,1}$ and $O_{2,3}$ are generated from three words of the internal state S, A, and B as follows:

$$O_{1,1} = (S \oplus B) + A$$
$$O_{2,3} = (S \oplus A) + B$$

This implies that the least significant bits of $O_{1,1}$ and $O_{2,3}$ are equal. [6] goes on to observe that there will also be biases in the more significant bits of $O_{1,1} \oplus O_{2,3}$.

In this paper, we show that a more effective distinguisher can be built using the same model of the cipher as the above by making use of all of the bits of $O_{1,1}$ and $O_{2,3}$ in concert rather than considering them separately. We use a hidden Markov model to trace the propogation of the unknown carry bits from least to most significant bit to calculate the exact probability that a given $O_{1,1}, O_{2,3}$ pair will be seen given that the event L takes place, and from this construct a distinguisher optimal for this model with the method described in [1]. We show that this results in a reduction in the number of samples needed by a factor of approximately 60552.

2 Description of Py

An understanding of the exact workings of Py is not needed to follow how our work builds on the work of [6], but we describe the round function here for completeness. Py operates on 32-bit words (treated as members of $\mathbb{Z}/2^{32}\mathbb{Z}$) and (8-bit) bytes. Its internal state in round i comprises

Algorithm 1 Py round function

$$
\begin{aligned}
O_{1,i} &= (\text{ROTL32}(s_i, 25) \oplus Y_i[256]) + Y_i[P_i[26]] \\
O_{2,i} &= (s_i \oplus Y_i[-1]) + Y_i[P_i[208]] \\
Y_{i+1} &= Y_i[-2 \ldots 256] \parallel ((\text{ROTL32}(s_i, 14) \oplus Y_i[-3]) + Y_i[P_i[153]]) \\
P_{i+1} &= \begin{cases} P_i[1 \ldots k-1] \parallel P_i[0] \parallel P_i[k+1 \ldots 255] \parallel P_i[k] & k \neq 0 \\ P_i[1 \ldots 255] \parallel P_i[0] & k = 0 \end{cases} \\
&\quad \text{where} \quad k = Y_{i+1}[185] \bmod 256 \\
s_{i+1} &= \text{ROTL32}(s_i + Y_{i+1}[P_{i+1}[72]] - Y_{i+1}[P_{i+1}[239]], (P_{i+1}[116] + 18) \bmod 32)
\end{aligned}
$$

"$\parallel$" represents array concatenation.

- a 260-word array Y_i, indexed from -3 to 256
- a 256-byte array P_i indexed from 0 to 255 which always contains a permutation, and
- a word s_i.

The specification of Py in [2] describes the round function as two state-update functions with an output function inbetween. To simplify cryptanalysis, we mark the boundaries between rounds differently, so that we can consider the round function to be an output function followed by a state-update function combining both parts. This is consistent with the conventions of [6]. Algorithm 1 defines the output and state update functions; it produces two 32-bit output words $O_{1,i}, O_{2,i}$ in round i.

We do not specify the key/IV setup; like [6], for all of our results we model P_1, Y_1 and s_1 as independent and uniformly distributed, with P_1 uniformly distributed over permutations of bytes.

3 Sekar et al attack

[6] presents a distinguisher against Py that requires 8 bytes of output from each of $2^{83.82}$ distinct keystreams. The authors define an event L which is the combination of the following six conditions:

- $P_2[116] \equiv -18 \pmod{32}$
- $P_3[116] \equiv 7 \pmod{32}$
- $P_2[72] = P_3[239] + 1$
- $P_2[239] = P_3[72] + 1$
- $P_1[26] = 1$
- $P_3[208] = 254$

They show that $\Pr[L] \approx 2^{-41.91}$ (with the initial state is modelled as random as always). Defining

$$
\begin{aligned}
A &= Y_1[1] \\
B &= Y_1[256] \\
S &= \text{ROTL32}(s_1, 25)
\end{aligned}
$$

they show that where L occurs,

$$
\begin{aligned}
O_{1,1} &= (S \oplus B) + A \\
O_{2,3} &= (S \oplus A) + B
\end{aligned}
$$

In particular, where $[A]_0$ is the low bit of A, this implies that $[O_{1,1} \oplus O_{2,3}]_0 = ([S]_0 \oplus [B]_0 \oplus [A]_0) \oplus ([S]_0 \oplus [A]_0 \oplus [B]_0) = 0$. The authors show that $\Pr[[O_{1,1} \oplus O_{2,3}]_0 = 0 | \neg L] = \frac{1}{2}$, and thus that

$$\begin{aligned}\Pr[[O_{1,1} \oplus O_{2,3}]_0 = 0] &= \Pr[[O_{1,1} \oplus O_{2,3}]_0 = 0 | L] \Pr[L] + \\ &\quad \Pr[[O_{1,1} \oplus O_{2,3}]_0 = 0 | \neg L] \Pr[\neg L] \\ &= \Pr[L] + \frac{1}{2}(1 - \Pr[L]) \\ &= \frac{1}{2}(1 + \Pr[L])\end{aligned}$$

The authors go on to estimate that this bias can be used to construct an effective distinguisher given roughly $\Pr[L]^{-2} \approx 2^{83.82}$ samples.

[6] defines a second event with the same probability which we term L', which is identical to L except that $P_2[72] = P_3[72] + 1$ and $P_2[239] = P_3[239] + 1$. The authors assert [4] that where L' occurs, $O_{1,1} = (\text{ROTL32}(S, 25) \oplus B) + A$ and $O_{2,3} = (\text{ROTL32}(S + 2K, 25) \oplus A) + B$ where S, K, A and B are all independent and uniformly random under the assumption of independent uniform randomness in the initial state. Where neither L nor L' occur, $O_{1,1}$ and $O_{2,3}$ are independent and uniformly random.

We now measure the exact efficacy of this distinguisher using [1] and show how to improve on it with a hidden Markov model.

4 Optimal distinguishers

[1] describes a general means to construct an efficient distinguisher between distributions D_0 and D_1 over a shared alphabet $\mathcal{Z}$, given n independent and identically distributed samples drawn from the unknown distribution D. $\Pr_{\mathsf{D}_j}[X]$ is shorthand for $\Pr[X | \mathsf{D} = \mathsf{D}_j]$ and $P_j(z) = \Pr_{\mathsf{D}_j}[D = z]$ where D is a random variable drawn from D. We consider only the case where $P_0(z) > 0$ and $P_1(z) > 0$ for all $z \in \mathcal{Z}$. Where $Z = z_1 \ldots z_n$ is the vector of samples, the efficacy of a distinguisher $\mathcal{A}$ is measured by its "advantage":

$$\text{Adv}(\mathcal{A}) = \Pr_{\mathsf{D}_1}[\mathcal{A}(Z) = 1] - \Pr_{\mathsf{D}_0}[\mathcal{A}(Z) = 1]$$

and [1] shows that the distinguisher $\mathcal{A}_{opt}$ defined here maximizes advantage given the information available:

$$\mathcal{A}_{opt}(Z) = \begin{cases} 1 & \text{where } P_1(Z) > P_0(Z) \\ 0 & \text{otherwise} \end{cases}$$

If we define the *log-likelihood ratio* function LLR below then (since each z_i is independent) $\mathcal{A}_{opt}$ can be expressed in a different way:

$$\begin{aligned}\text{LLR}(z) &= \log\left(\frac{P_1(z)}{P_0(z)}\right) \\ \mathcal{A}_{opt}(Z) &= \begin{cases} 1 & \text{where } \sum_i \text{LLR}(z_i) > 0 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Appealing to the central limit theorem, the authors show that where n is large, $\Pr_{\mathsf{D}_j}[\mathcal{A}_{opt}(Z) = 1] \approx \Phi\left(\frac{\sqrt{n}\mu_j}{\sigma_j}\right)$ where $\mu_j = \text{E}[\text{LLR}(\mathsf{D}_j)]$ and $\sigma_j^2 = \text{Var}[\text{LLR}(\mathsf{D}_j)]$. Next they define for every $z \in \mathcal{Z}$:

$$\epsilon_z = P_1(z) - P_0(z)$$

Where D_0 and D_1 are close (ie where $\epsilon_z \ll P_0(z)$ for all $z \in \mathcal{Z}$), they state that

$$-\mu_0 \approx \mu_1 \approx \frac{\beta}{2}, \quad \sigma_0^2 \approx \sigma_1^2 \approx \beta \quad \text{where} \quad \beta = \sum_{z \in \mathcal{Z}} \frac{\epsilon_z^2}{P_0(z)}$$

and thus that

$$\begin{aligned}
\mathrm{Adv}(\mathcal{A}_{opt}) &= \Pr_{\mathsf{D}_1}[\mathcal{A}_{opt}(Z) = 1] - \Pr_{\mathsf{D}_0}[\mathcal{A}_{opt}(Z) = 1] \\
&\approx \Phi\left(\frac{\sqrt{n}\mu_1}{\sigma_1}\right) - \Phi\left(\frac{\sqrt{n}\mu_0}{\sigma_0}\right) \\
&\approx \Phi\left(\frac{\sqrt{n}\beta}{2\sqrt{\beta}}\right) - \Phi\left(-\frac{\sqrt{n}\beta}{2\sqrt{\beta}}\right) \\
&= 1 - 2\Phi\left(-\frac{\sqrt{n\beta}}{2}\right)
\end{aligned}$$

For the distinguisher of [6] we have that

$P_j(z)$	$j=0$	$j=1$	ϵ_z
$z=0$	$\frac{1}{2}$	$\frac{1}{2}(1+\Pr[L])$	$\frac{1}{2}\Pr[L]$
$z=1$	$\frac{1}{2}$	$\frac{1}{2}(1-\Pr[L])$	$-\frac{1}{2}\Pr[L]$

where D_j is the distribution of $[O_{1,1}]_0 \oplus [O_{2,3}]_0$, from which we can deduce that in this instance $\beta = \Pr[L]^2$. Thus where $n = \Pr[L]^{-2}$ the advantage is approximately $1 - 2\Phi\left(-\frac{1}{2}\right) \approx 0.3829$; for an advantage greater than $\frac{1}{2}$, around 2^{85} samples (or 2^{88} bytes) are required. The presence of event L' makes no difference to the efficacy of this distinguisher.

5 Hidden Markov models

We can construct a more efficient distinguisher for Py by using a hidden Markov model [7,5]. We briefly reprise the theory of hidden Markov models here.

Consider a sequence of $n+1$ random variables $Q_0 \dots Q_n$ drawn from an alphabet of states $\Psi = \{S_1 \dots S_N\}$. We say this sequence is generated by a first-order Markov process if the probability that Q_{i+1} is in state S_k depends only on the previous state Q_i, or in other words, if for all $0 \le i < n$ and for all $q_0 \dots q_{i+1} \in \Phi^{i+1}$

$$\Pr[Q_{i+1} = q_{i+1} | Q_0 \dots Q_i = q_0 \dots q_i] = \Pr[Q_{i+1} = q_{i+1} | Q_i = q_i]$$

We define the initial state vector π such that $\pi_i = \Pr[Q_0 = S_i]$, and the transition matrix M_i such that $(M_i)_{jk} = \Pr[Q_{i+1} = S_k | Q_i = S_j]$. The entries of π must sum to 1, as must each column of each M_i. For all the processes we consider here, each M_i will be the same and we drop the subscript i. π and M completely characterize the Markov process.

In a hidden Markov model, we also consider each transition[1] to also generate an output Y_i from an output alphabet $\mathcal{Y}$. We therefore define a transition matrix M_y for each possible output symbol $y \in \mathcal{Y}$ such that $(M_y)_{jk} = \Pr[Y_i = y \wedge Q_{i+1} = S_k | Q_i = S_j]$. For each state the probabilities of each output/next-state pair must sum to 1 as before, so each column of $\sum_{y \in \mathcal{Y}} M_y$ must sum to 1.

Given this matrix representation, if we define the vector $\mathbf{x} = M_{y_{n-1}} \dots M_{y_0} \pi$ then $\mathbf{x}_i = \Pr[(Y_0 \dots Y_{n-1}) = (y_0 \dots y_{n-1}) \wedge Q_n = S_i]$ and thus the sum of the elements of $\mathbf{x}$ gives the probability of the output sequence $y_0 \dots y_{n-1}$. This is known as the "forward algorithm".

[1] Following the practice described in section IV.C of [5], we specify outputs as produced on transitions, not from states

6 Applying the hidden Markov model to Py

In order to build a more efficient distinguisher using this method, we now consider the problem of calculating $\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L]$. A naive algorithm for this, based on the observation that $\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L] = \Pr[(S \oplus B) + A = o_1 \wedge (S \oplus A) + B = o_2] = |\{a, b, s \in (\mathbb{Z}/2^{32}\mathbb{Z})^3 | (s \oplus b) + a = o_1 \wedge (s \oplus a) + b = o_3\}|/2^{96}$, will take approximately 2^{96} operations. We use a hidden Markov model to calculate this exactly and efficiently.

Define

$$\text{carry}(x, y) = (x + y) \oplus x \oplus y$$

it is well known (see eg [3]) that if $z = \text{carry}(x, y)$ then $[z]_0 = 0$ and $[z]_{i+1} = \text{maj}([x]_i, [y]_i, [z]_i)$ for $i \in 0 \ldots 30$ where maj is the binary majority function.

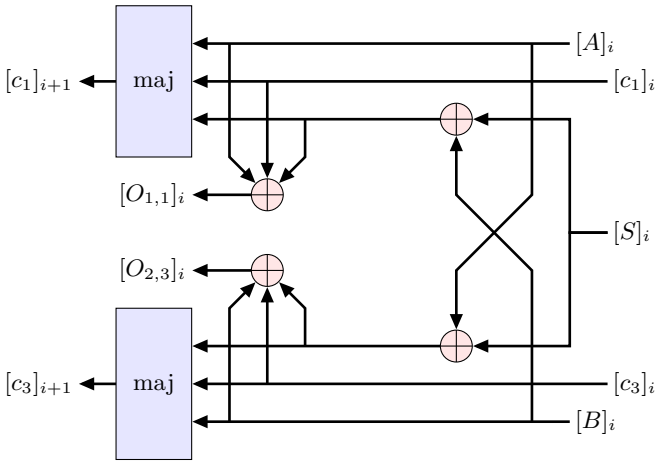


Figure 1. Calculating $[O_{1,1}]_i, [O_{2,3}]_i$

Following [6] we define $c_1 = \text{carry}(S \oplus B, A)$ and $c_3 = \text{carry}(S \oplus A, B)$. Our sequence of hidden states is the sequence of pairs of carry bits $([c_1]_i, [c_3]_i)$ for each bit i; the initial state is $([c_1]_0, [c_3]_0) = (0, 0)$ with probability 1, and the hidden Markov model tracks the propogation of these carry bits from least to most signficant bit in parallel across the two addition operations. Our outputs are pairs of bits $[O_{1,1}]_i, [O_{2,3}]_i$. Both the states and the outputs are drawn from the alphabet $\Psi = \mathcal{Y} = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. A transition is represented in figure 1.

Each transition depends on the three independent uniform random bits $[A]_i$, $[B]_i$ and $[S]_i$. This gives us enough information to exactly specify the probability that a particular output and next state will result from a given state; it is determined by the number of $([A]_i, [B]_i, [S]_i)$ triples of bits that can produce this output/next state from that state. Given this model, the forward algorithm [5,7] can straightforwardly be used to exactly calculate $\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L]$ for any (o_1, o_3) pair.

We determine the transition matrices below.

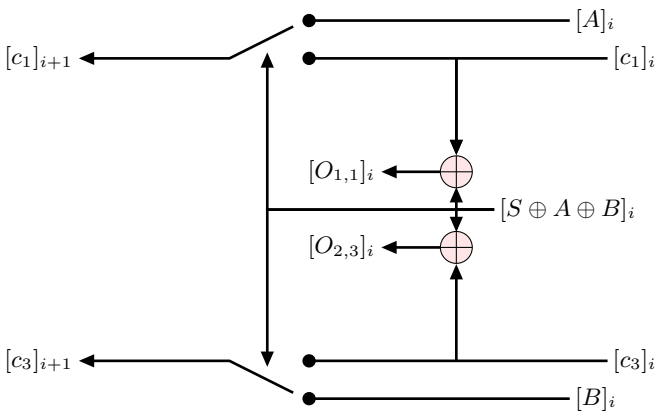


Figure 2. Simplification of figure 1

$$
\begin{aligned}
&\Pr\left[\begin{array}{c}[O_{1,1}]_i, [O_{2,3}]_i = w_1, w_3 \\ \wedge\ [c_1]_{i+1}, [c_3]_{i+1} = v_1, v_3\end{array}\,\middle|\, L \wedge [c_1]_i, [c_3]_i = u_1, u_3\right] \\
&= \frac{\left|\left\{a, b, s \in \{0,1\}^3 \,\middle|\, \begin{array}{c} w_1 = s \oplus b \oplus a \oplus u_1 \wedge w_3 = s \oplus a \oplus b \oplus u_3 \\ \wedge\ v_1 = \mathrm{maj}(s \oplus b, a, u_1) \wedge v_1 = \mathrm{maj}(s \oplus a, b, u_3)\end{array}\right\}\right|}{8} \\
&= \frac{\left|\left\{a, b, s' \in \{0,1\}^3 \,\middle|\, \begin{array}{c} w_1 = s' \oplus u_1 \wedge w_3 = s' \oplus u_3 \\ \wedge\ v_1 = \mathrm{maj}(s' \oplus a, a, u_1) \wedge v_1 = \mathrm{maj}(s' \oplus b, b, u_3)\end{array}\right\}\right|}{8} \\
&= \frac{\left|\left\{a, b, s' \in \{0,1\}^3 \,\middle|\, \begin{array}{c} w_1 = s' \oplus u_1 \wedge w_3 = s' \oplus u_3 \\ \wedge\ v_1 = \mathrm{IF}(s', a, u_1) \wedge v_1 = \mathrm{IF}(s', b, u_3)\end{array}\right\}\right|}{8} \\
&= \begin{cases} \frac{1}{2} & \text{if } (u_1, u_3) = (v_1, v_3) = (\neg w_1, \neg w_3) \\ \frac{1}{8} & \text{if } (u_1, u_3) = (w_1, w_3) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

where

$$\mathrm{IF}(a, b, c) = \begin{cases} b & \text{if } a = 0 \\ c & \text{if } a = 1 \end{cases}$$

This simplification (illustrated in figure 2) is achieved by defining $s' = a \oplus b \oplus s$. This yields the following transition matrices:

$$M_{(0,0)} = \begin{pmatrix} \frac{1}{8} & 0 & 0 & 0 \\ \frac{1}{8} & 0 & 0 & 0 \\ \frac{1}{8} & 0 & 0 & 0 \\ \frac{1}{8} & 0 & 0 & \frac{1}{2} \end{pmatrix}, \; M_{(0,1)} = \begin{pmatrix} 0 & \frac{1}{8} & 0 & 0 \\ 0 & \frac{1}{8} & 0 & 0 \\ 0 & \frac{1}{8} & \frac{1}{2} & 0 \\ 0 & \frac{1}{8} & 0 & 0 \end{pmatrix},$$

$$M_{(1,0)} = \begin{pmatrix} 0 & 0 & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{8} & 0 \\ 0 & 0 & \frac{1}{8} & 0 \\ 0 & 0 & \frac{1}{8} & 0 \end{pmatrix}, \; M_{(1,1)} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{8} \\ 0 & 0 & 0 & \frac{1}{8} \\ 0 & 0 & 0 & \frac{1}{8} \\ 0 & 0 & 0 & \frac{1}{8} \end{pmatrix}$$

Finally, we apply the forward algorithm described above, to yield the formula

$$\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L] = (\,1\;1\;1\;1\,) M_{([o_1]_{31},[o_3]_{31})} \dots M_{([o_1]_0,[o_3]_0)} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This more sophisticated model of $O_{1,1}, O_{2,3}$ yields some surprising results. For example if $O_{1,1}$ ends with the suffix 01^k for any k, then $O_{2,3}$ must end with the same suffix.

7 The Markov distinguisher

Now that we can efficiently calculate $\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L]$, we can use the techniques from [1] presented in section 4 to directly construct a distinguisher from the probability model.

We examine n streams from n distinct key/IV pairs, and from each stream i we take a sample $z_i = O_{1,1}, O_{2,3}$, so our alphabet $\mathcal{Z}$ consists of all pairs of 32-bit words[2]. As above, we define $\mathrm{LLR}(z) = \log\left(\frac{P_1(z)}{P_0(z)}\right)$ and our distinguisher returns 1 iff $\sum_i \mathrm{LLR}(z_i) > 0$.

We do not yet take account of event L'; where L does not occur, we model $O_{1,2}, O_{2,3}$ as independent and uniformly random. This introduces a small error; we believe that the distinguisher will nevertheless be roughly as effective as advertised, but it is likely that a very slightly more effective distinguisher could be built by taking L' into account. Instead, we approximate $P_1(z)$ as $\Pr[(O_{1,1}, O_{2,3}) = z|L] \Pr[L] + P_0(z) \Pr[\neg L]$.

To find β for this distinguisher and thus discover the number of samples required for a given advantage, we proceed as follows:

$$\begin{aligned}
\beta &= \sum_{z \in \mathcal{Z}} \frac{(P_1(z) - P_0(z))^2}{P_0(z)} \\
&= |\mathcal{Z}| \sum_{z \in \mathcal{Z}} (P_1(z) - \frac{1}{|\mathcal{Z}|})^2 \\
&= |\mathcal{Z}| \sum_{z \in \mathcal{Z}} \left(\begin{array}{l} \Pr_{\mathsf{D}_1}[(O_{1,1}, O_{2,3}) = z|L] \;\Pr[L] \\ + \Pr_{\mathsf{D}_1}[(O_{1,1}, O_{2,3}) = z|\neg L] \;\Pr[\neg L] \\ - \frac{1}{|\mathcal{Z}|} \end{array} \right)^2 \\
&= |\mathcal{Z}| \sum_{z \in \mathcal{Z}} \left(\begin{array}{l} \Pr_{\mathsf{D}_1}[(O_{1,1}, O_{2,3}) = z|L] \;\Pr[L] \\ + \frac{1}{|\mathcal{Z}|}(1 - \Pr[L]) \\ - \frac{1}{|\mathcal{Z}|} \end{array} \right)^2 \\
&= |\mathcal{Z}| \Pr[L]^2 \sum_{z \in \mathcal{Z}} \left(\Pr_{\mathsf{D}_1}[(O_{1,1}, O_{2,3}) = z|L] - \frac{1}{|\mathcal{Z}|} \right)^2
\end{aligned}$$

[2] Two alphabets are at work in this distinguisher. The hidden Markov model works over an alphabet of pairs of bits $\mathcal{Y} = \{0,1\}^2$ to find the probability of a given pair of words; the optimal distinguisher constructed from it works on an alphabet of pairs of words $\mathcal{Z} = (\mathbb{Z}/2^{32}\mathbb{Z})^2$. Note that $|\mathcal{Z}| = |\mathcal{Y}|^{32}$.

We cannot directly compute this sum in reasonable time because $\mathcal{Z}$ has 2^{64} elements. However, we can define the following function family:

$$f_k(\mathbf{x}) = \sum_{\mathbf{y} \in \mathcal{Y}^k} ((1\ 1\ 1\ 1) M_{y_0} M_{y_1} \dots M_{y_{k-1}} \mathbf{x} - \frac{1}{|\mathcal{Z}|})^2$$

and from our formula for $\Pr[(O_{1,1}, O_{2,3}) = z|L]$ we see that

$$\beta = |\mathcal{Z}| \Pr[L]^2 f_{32} \left(\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right)$$

Furthermore, we can define f recursively:

$$f_0(\mathbf{x}) = ((1\ 1\ 1\ 1)\mathbf{x} - \frac{1}{|\mathcal{Z}|})^2$$
$$f_{k+1}(\mathbf{x}) = \sum_{y \in \mathcal{Y}} f_k(M_y \mathbf{x})$$

This by itself does not yield an efficient algorithm for finding β, since each evaluation of f_{k+1} requires four evaluations of f_k. However, we now show by induction that there exists a series of matrices $A_0 \dots A_{32}$ such that $f_k(\mathbf{x}) = \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}^T A_k \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}$. A_0 is simply a 5x5 matrix whose every entry is 1, and

$$\begin{aligned}
f_{k+1}(\mathbf{x}) &= \sum_{y \in \mathcal{Y}} f_k(M_y \mathbf{x}) \\
&= \sum_{y \in \mathcal{Y}} \begin{pmatrix} M_y \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}^T A_k \begin{pmatrix} M_y \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix} \\
&= \sum_{y \in \mathcal{Y}} \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}^T \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix}^T A_k \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}^T \left(\sum_{y \in \mathcal{Y}} \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix}^T A_k \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix} \right) \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}^T A_{k+1} \begin{pmatrix} \mathbf{x} \\ -\frac{1}{|\mathcal{Z}|} \end{pmatrix}
\end{aligned}$$

where

$$A_{k+1} = \sum_{y \in \mathcal{Y}} \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix}^T A_k \begin{pmatrix} M_y & 0 \\ 0 & 1 \end{pmatrix}$$

We can therefore use this algorithm to find A_{32} recursively, from which we find that $\beta \approx 60552 \Pr[L]^2$. For a distinguisher with the same advantage as that of [6], we therefore need only $n = \left\lceil \frac{1}{\beta} \right\rceil \approx \frac{1}{60552} \Pr[L]^{-2}$ samples.

8 Conclusions and further work

We have shown that Py can be distinguished from a random function given roughly a factor of 2^{16} fewer samples than the previous best attack in [6]. We prefer to state the

number of samples needed to gain advantage greater than $\frac{1}{2}$; with 2^{69} samples—ie 2^{72} bytes—the attack has an advantage of around 0.53. Like that attack, this attack is not restricted to using words at the start of the stream to build the distinguisher; it may use nearly the entire stream. This means that there will be correlations between samples, but those correlations are unlikely to affect the efficacy of the attack. Py is limited to producing 2^{64} bytes from a single key/IV pair, which is equivalent to just under 2^{61} samples, so we gain advantage greater than $\frac{1}{2}$ once the complete streams from roughly 2^8 different key/IV pairs are used. Surprisingly, this attack is disallowed by the security goals set out in [2], which limit the attacker to at most 2^{64} bytes of keystream total. Against a single complete stream, our attack offers advantage 0.03, which is low but perhaps not negligible.

We did not take account of event L' defined in section 3. We anticipate that if we did so, we would need fewer samples still. Extending the hidden Markov model to find $\Pr[(O_{1,1}, O_{2,3}) = (o_1, o_3)|L \vee L']$ is not hard—a single bit may be added to the state indicating which of L or L' took place—but we have not yet done the work of estimating β for this extended model.

9 Acknowledgements

Thanks to Souradyuti Paul for invaluable clarification of [6], to Shahram Khazaei for the suggestion of considering $O_{1,1}$ and $O_{2,3}$ separately, and to Matthias Radestock for useful commentary on my drafts.

References

1. Thomas Baignères, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 432–450. Springer, 2004.
2. Eli Biham and Jennifer Seberry. Py (Roo) : A fast and secure stream cipher using rolling arrays. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/023, 2005.
3. Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In Mitsuru Matsui, editor, *Fast Software Encryption 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.
4. Souradyuti Paul. Re: Improved cryptanalysis of Py. Personal emails, December 2006.
5. Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. pages 267–296, 1990.
6. Gautham Sekar, Souradyuti Paul, and Bart Preneel. Distinguishing attacks on the stream cipher Py. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/081, 2005.
7. Wikipedia. Hidden Markov model — Wikipedia, the free encyclopedia, 2006. [Online; accessed 19-January-2006].

http://www.ciphergoth.org/crypto/py/

Practical Attacks on one Version of DICING

Gilles Piret

Ecole Normale Supérieure, Département d'Informatique,
45, Rue d'Ulm, 75230 Paris cedex 05, France
http://www.di.ens.fr/~piret/
Gilles.Piret@ens.fr

Abstract

DICING is a synchronous stream cipher submitted to the eSTREAM project. Two versions of the cipher actually exist: the first one can be found in the proceedings of the SKEW conference, while the second is available from the web site. In this paper we describe practical distinguishing and key recovery attacks against the first version. These attacks do not apply as such to the web site version of DICING.

Keywords stream cipher cryptanalysis, eSTREAM, DICING, irregular clocking.

1 Introduction

The eSTREAM project [1] aims at identifying new stream ciphers that might become suitable for widespread adoption. For this purpose, a public call for primitives has been made in November 2004. In May 2005, it resulted in 34 stream cipher submissions.

DICING is one of them. It is based on four Galois-style LFSRs, two of which are used to clock the other two. While such irregular clocking is a good way to obtain non-linearity at a low cost, the security of primitives based on this principle is often difficult to analyze.

It happens to be two versions of DICING. The first one [2] can be found in the proceedings of the SKEW conference, that took place in Åarhus, Denmark, on May 26-27, 2005. The second [3] is available from the eSTREAM web site; it differs from [2] by several changes to the output function. In this paper, we are concerned with the security of the first version. We show that the way variable clocking is applied in it leads to very serious weaknesses.

2 Notations

Throughout this paper, we use the following notations:

- $\mathbb{F}_{2^n}$ is the Galois field with 2^n elements.
- $\oplus$ denotes exclusive or, that is bitwise addition.
- & denotes bitwise AND.
- $\sim X$ denotes the bit by bit complement of X.
- $X \gg a$ denotes the right shift of X by a bits.
- $X[a, b]$ denotes the substring of binary string X, going from bit position a to bit position b (bit positions are numbered from 0). $X[i]_{\text{byte}}$ denotes the ith byte of X, starting from 0.

3 Description of DICING

3.1 State Update Function

The DICING stream cipher is based on four Galois-style LFSRs Γ_1, Γ_2, Γ_3, Γ_4. Let $\alpha_t \in \mathbb{F}_{2^{127}}, \beta_t \in \mathbb{F}_{2^{126}}, \omega_t \in \mathbb{F}_{2^{128}}, \tau_t \in \mathbb{F}_{2^{128}}$ denote the state of LFSR $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ respectively. We can represent the elements of a Galois field of characteristic 2 as polynomials in $\mathbb{F}_2[x]/p(x)$, where $p(x)$ is an irreducible polynomial over $\mathbb{F}_2$. As an example, α_t will be denoted as $\alpha_{t,126} \cdot x^{126} \oplus \alpha_{t,125} \cdot x^{125} \oplus ... \oplus \alpha_{t,0}$. If the polynomial chosen corresponds to the feedback polynomial of the LFSR, then shifting the LFSR is equivalent to multiplication by x in $\mathbb{F}_2[x]/p(x)$. We do not give the feedback polynomials $p_i(x)$ of LFSRs $\Gamma_i (i = 1...4)$ here as they are not relevant for our attacks. Moreover, we often omit the modulo in our equations, as it is obvious from the context.

LFSRs Γ_1 and Γ_2 are shifted 8 bits per clock cycle, and are used to clock the other two LFSRs. More precisely, the state update process is the following:

1. The last eight bits of α_t and β_t are stored in *dices* D'_t and D''_t:

$$\begin{aligned} D'_t &= (\alpha_{t,126}, ..., \alpha_{t,119}) \in \mathbb{F}_2^8 \\ D''_t &= (\beta_{t,125}, ..., \beta_{t,118}) \in \mathbb{F}_2^8 \end{aligned} \tag{1}$$

   Then Γ_1 and Γ_2 are updated:

$$\begin{aligned} \alpha_{t+1} &= x^8 \cdot \alpha_t \mod p_1(x) \\ \beta_{t+1} &= x^8 \cdot \beta_t \mod p_2(x) \end{aligned} \tag{2}$$

2.

$$D_t = D'_t \oplus D''_t, \quad a_t = D_t \& 15 \in \mathbb{F}_2^4, \quad b_t = D_t \gg 4 \in \mathbb{F}_2^4 \tag{3}$$

3. Two memories $u_t, v_t \in \mathbb{F}_{2^{128}}$ are updated by XORing the states ω_t and τ_t to them:

$$\begin{aligned} u_t &= u_{t-1} \oplus \omega_t \\ v_t &= v_{t-1} \oplus \tau_t \end{aligned} \tag{4}$$

4. Γ_3 and Γ_4 are updated by shifting them from 0 to 15 bits depending on the value of a_t and b_t:

$$\begin{aligned} \omega_{t+1} &= x^{a_t} \cdot \omega_t \mod p_3(x) \\ \tau_{t+1} &= x^{b_t} \cdot \tau_t \mod p_4(x) \end{aligned} \tag{5}$$

3.2 Output Function

At each clock cycle, the output function produces a 128-bit value z_t, depending on values u_t, v_t, D'_t, D''_t. The function used depends on how D'_t and D''_t compares:

$$z_t = \begin{cases} C_0(u_t) \oplus v_t & \text{if } D'_t > D''_t \\ C_0(v_t) \oplus u_t & \text{if } D'_t < D''_t \\ u_t \oplus v_t & \text{if } D'_t = D''_t \end{cases} \tag{6}$$

where C_0 is a non-linear and key-dependent function.

3.3 Initialization

The initialization of the generator is done in four phases:

1. The key and IV material are used to compute the initial states $\alpha_{-64}, \beta_{-64}, \omega_{-64}, \tau_{-64}$ of the four LFSRs.
2. The state update function is applied to them 32 times without any output.
3. The resulting state is used to construct the function C_0 used in the output function (see [2] for more details).
4. The state update function is applied another 32 times. We obtain $\alpha_0, \beta_0, \omega_0, \tau_0$, the initial states of $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ before keystream generation.

The key and IV loading proceeds as follows:

1. $K_I = K \oplus IV$
2. $K' = \begin{cases} K_I & \text{if } K \text{ has length 256} \\ K_I | (\sim K_I) & \text{if } K \text{ has length 128} \end{cases}$
3. $K_{ICS} = S_0(K' \oplus c)$, where S_0 denotes the parallel application of a fixed S-box $S_0 : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ and c is a constant.
4. $\alpha_{-64} = K_{ICS}[0, 126] \qquad \beta_{-64} = K_{ICS}[128, 253]$
5. $s = \bigoplus_{0 \le i < 32} K_{ICS}[i]_{\text{byte}} \in \mathbb{F}_2^8 \qquad \sigma = (s, s, ..., s) \in \mathbb{F}_2^{256}$
6. $K_{II} = S_0(K_{ICS} \oplus \sigma \oplus (\sim c))$
7. $\omega_{-64} = K_{II}[0, 127] \qquad \tau_{-64} = K_{II}[128, 255]$

It is remarkable that the knowledge of ω_{-64} and τ_{-64} is enough to retrieve K_I.

4 A Practical Distinguisher

Assume that during cycle t both dices D'_t and D''_t have the same value. Due to statistical properties of the LFSRs this event happens exactly with probability 1/256. Then $(a_t, b_t) = D_t = D'_t \oplus D''_t = 0$. Therefore states ω_t and τ_t do not change during this cycle: $\omega_{t+1} = \omega_t$ and $\tau_{t+1} = \tau_t$. It implies $u_{t+1} = u_t \oplus \omega_{t+1} = u_{t-1} \oplus \omega_t \oplus \omega_{t+1} = u_{t-1}$ and similarly $v_{t+1} = v_{t-1}$. Finally if the output function used is the same for cycles $t-1$ and $t+1$, we have $z_{t+1} = z_{t-1}$. It will happen whenever ($D'_{t-1} < D''_{t-1}$ and $D'_{t+1} < D''_{t+1}$), or ($D'_{t-1} = D''_{t-1}$ and $D'_{t+1} = D''_{t+1}$), or ($D'_{t-1} > D''_{t-1}$ and $D'_{t+1} > D''_{t+1}$), thus with probability $2 \cdot \left(\frac{2^8-1}{2^9}\right)^2 + \frac{1}{2^{16}} \simeq \frac{1}{2}$.
The conclusion is that two 128-bit output words produced at cycles $t-1$ and $t+1$ are equal with probability $\simeq \frac{1}{512}$ (instead of 2^{-128} for a truly random sequence). So the amount of keystream necessary for our distinguisher to work is about $512 \cdot 128$ bits = 64 Ko. The processing time is negligible.

5 A Key Recovery Attack

Instead of assuming $D'_t = D''_t$, suppose that D'_t and D''_t agree on their 4 right-most (or left-most) bits only. Then (assuming $a_t = 0$, the other case $b_t = 0$ is similar)

$$\begin{cases} \omega_{t+1} = \omega_t \\ \tau_{t+1} = x^{b_t} \cdot \tau_t \neq \tau_t \end{cases}, \tag{7}$$

which implies

$$\begin{cases} u_{t+1} = u_{t-1} \\ v_{t+1} \neq v_{t-1} \end{cases}. \tag{8}$$

If $D'_{t-1} > D''_{t-1}$ (resp. $D'_{t+1} > D''_{t+1}$) then $z_{t-1} = C_0(u_{t-1}) \oplus v_{t-1}$ (resp. $z_{t+1} = C_0(u_{t+1}) \oplus v_{t+1}$). As both events occur with probability $\simeq 1/2$, and $a_t = 0$ with probability 1/16, we conclude that

$$z_{t-1} \oplus z_{t+1} = v_{t-1} \oplus v_{t+1} = \tau_t \oplus \tau_{t+1} = \tau_t \cdot (1 \oplus x^{b_t}) \tag{9}$$

is satisfied with probability $\simeq 1/64$. Similarly (considering the case $b_t = 0$ instead of $a_t = 0$),

$$z_{t-1} \oplus z_{t+1} = u_{t-1} \oplus u_{t+1} = \omega_t \oplus \omega_{t+1} = \omega_t \cdot (1 \oplus x^{a_t}) \tag{10}$$

is satisfied with probability $\simeq 1/64$ as well.

Assume the attacker has got a long enough keystream sequence $(z_t)_{t \geq 0}$. The idea of the attack is the following: each time $a_t = 0$ (resp. $b_t = 0$) and the conditions on $D'_{t-1}, D''_{t-1}, D'_{t+1}, D''_{t+1}$ are satisfied, by guessing correctly the value of b_t (resp. a_t), we can obtain the actual value of τ_t (resp. ω_t) by using equation (9) (resp. equation (10)). From this value we can compute the sequence of the past states

of the LFSR (with two consecutive elements of the sequence differing by one bit shift of the LFSR). As equations (9) and (10) are satisfied relatively often, by considering enough positions t we will observe several similar sequences (provided we "align" them correctly). On the other hand, when equation (9) (resp. (10)) is not satisfied, or we do not guess correctly, the value τ_t (resp. ω_t) we deduce can be considered as random (see Appendix A). So the observed similar sequences highly probably correspond to the right one. The best way to identify them is to use two hash tables (or sorted lists) Ω and $\mathbf{T}$.

Once the actual past history of Γ_3 and Γ_4 has been identified, it remains to identify the actual initial states ω_{-64} and τ_{-64} of these registers. Knowing the state of one LFSR at cycle t, the number of bit shifts separating it from the initial state could roughly go from 0 to $15 \cdot (t + 64)$ depending on how the LFSR has been clocked, with an expectancy of $7.5 \cdot (t + 64)$. So we guess this distance for both LFSRs, beginning with values close to the expectancy (which are the most probable ones, as the distance is a random variable with roughly a normal distribution). The computed initial states allow us to retrieve the key from which we can compute a keystream. Comparison with the actual keystream is used to accept or reject the guess on the distances.

More precisely, the attack goes as follows:

1. $\Omega, \mathbf{T} = \varnothing$. For $t = 1, 2, ...$:
   (a) Compute $z_{t-1} \oplus z_{t+1}$.
   (b) Assume $a_t = 0$. For $b_t = 1, 2, ..., 15$:
   Deduce the supposed value $\tau_t^{(b_t)}$ using (9). Use it to compute the history of Γ_4: more precisely, we compute a sequence of values $(\tau_{t,s}^{(b_t)})_{-15t+15 \leq s \leq 0}$, such that $\tau_{t,0}^{(b_t)} = \tau_t^{(b_t)}$ and $\tau_{t,s}^{(b_t)} = x \cdot \tau_{t,s-1}^{(b_t)}$. The bound on s is chosen such that, assuming the guesses on a_t and b_t were right, all values in the actual sequence $(\tau_t)_{t \geq 1}$ are also in $(\tau_{t,s}^{(b_t)})_{-15t+15 \leq s \leq 0}$. The difference between both sequences is that the latter is regularly clocked (shifts of one bit at a time), while the former results from variable clocking. For $-15t + 15 \leq s \leq 0$, check whether $\tau_{t,s}^{(b_t)}$ is in the hash table $\mathbf{T}$.
      - If yes, let $(\tau_{t^*}^{(b_{t^*})}, t^*) \in \mathbf{T}$ be this element. It is probably part of the true history of Γ_4. From $(\tau_{t^*}^{(b_{t^*})}, t^*)$ reconstruct the history $(\tau_{t^*,s}^{(b_{t^*})})_{-15 \cdot (t^*+64) \leq s \leq 0}$.
      - Else store $(\tau_t^{(b_t)}, t)$ in $\mathbf{T}$.
   (c) Similarly, we can assume $b_t = 0$ and compute a supposed value $\omega_t^{(a_t)}$ for each $a_t \in \{1, 2, ..., 15\}$ using (10). We deduce candidate sequences $(\omega_{t,s}^{(a_t)})_{-15t+15 \leq s \leq 0}$, and use another hash table Ω to identify similar sequences. The only difference with step (b) is that computations are performed modulo $p_3(x)$, instead of modulo $p_4(x)$.
   (d) Stop the loop as soon as the true history of both Γ_3 and Γ_4 has been found.

2. Let $(\tau_s^*)_s$ and $(\omega_s^*)_s$ be the two sequences we computed at step 1, and t', t'' be the respective corresponding indexes of the clock cycle corresponding to $s = 0$. Let $\overline{s}' := \lfloor -7.5 \cdot (t' + 64) \rfloor$ and $\overline{s}'' := \lfloor -7.5 \cdot (t'' + 64) \rfloor$. Let us denote by $\texttt{Try}(s_1, s_2)$ the following computation:
   - Assume that s_1 and s_2 are the indexes of the initial state of Γ_4 and Γ_3 in $(\tau_s^*)_s$ and $(\omega_s^*)_s$ respectively.
   - Deduce the initial key.
   - Check it by generating a keystream from it and comparing it to the actual keystream.

   First we perform $\texttt{Try}(\overline{s}', \overline{s}'')$. Then we set $i = 1$ and repeat:
   (a) For $j = \overline{s}'' - i$ to $j = \overline{s}'' + i$, $\texttt{Try}(\overline{s}' - i, j)$ and $\texttt{Try}(\overline{s}' + i, j)$.
   (b) For $j = \overline{s}' - i + 1$ to $j = \overline{s}' + i - 1$,
   $\texttt{Try}(j, \overline{s}'' - i)$ and $\texttt{Try}(j, \overline{s}'' + i)$.
   (c) i++

   We stop as soon as `Try` gives a positive answer.

Remark that step 1 is another distinguisher for DICING: as a matter of fact, performing this computation on a truly random sequence is very unlikely to lead to discovery of a collision in Ω or $\mathbf{T}$. As we will see, this distinguisher requires less data than the previous one, but more computation.

We now look at the complexity of the attack. For it to succeed, we need equations (9) and (10) to be satisfied in two distinct positions t. As these equations are satisfied with probability $1/64$, a keystream of about 128 words= 16 Ko is necessary.

Regarding the time complexity of the first phase of the attack (and hence of the distinguisher), about $128 \cdot 15$ sequences $(\tau_{t,s}^{(b_t)})_{-15 \cdot t \leq s \leq 0}$ and $128 \cdot 15$ sequences $(\omega_{t,s}^{(a_t)})_{-15 \cdot t \leq s \leq 0}$ need to be computed. Their average length is $15 \cdot 64$, so the total time complexity of this phase is about $2^7 \cdot 15 \cdot 2 \cdot 15 \cdot 64 \simeq 2^{22}$ LFSR shifts and 2^{22} hash table lookups (which are assumed to be feasible in constant time).

As for the second phase, assuming the first occurrence of the actual history roughly took place for $t = 64$, the number of pairs of initial states we have to test is at most $(15 \cdot 128)^2 \simeq 2^{22}$, which is still practical (each test requires computation of the initialization of the stream cipher, and of a few words of keystream). Note that most of the time less than 2^{16} pairs will be tested before finding the right combination of indexes, and hence the key (as the right index is a random normal variable).

6 The Other Version of DICING

The second version of DICING, available from the ECRYPT web site [3], differs from the description made in section 3 in its output function,

which becomes

$$z_t = \begin{cases} C(u_t, v_t) & \text{if } D'_{t-1} > D''_{t-1} \\ C(v_t, u_t) & \text{if } D'_{t-1} < D''_{t-1} \\ \varnothing & \text{if } D'_{t-1} = D''_{t-1} \end{cases} \tag{11}$$

We remark three changes:

- The choice of the output function no longer depends on the comparison of the current dices, but rather of the previous ones. Note that although it does not formally change the state update function, its computation order is modified. As a matter of fact, this change amounts to updating LFSRs Γ_3 and Γ_4 *before* updating memories u_t and v_t.
- If both dices are equal, the output function outputs nothing (instead of $u_t \oplus v_t$).
- The new output function C used whenever both dices are different is no longer linear in any of its component; it is still key-dependent.

The first two changes prevent use of the distinguisher described in section 4. As a matter of fact, we still have $D'_t = D''_t \Rightarrow (u_{t-1} = u_{t+1}$ and $v_{t-1} = v_{t+1})$. But as $D'_t = D''_t$, there is no output corresponding to u_{t+1} and v_{t+1}. Otherwise said, one of the two repeating values does not appear anymore.

Our second attack (in section 5) obviously exploits partial linearity in the output function. As this linearity has been removed, it no longer works.

7 Conclusion

In this paper we have shown that one of the two versions of DICING is so weak that practical attacks can be mounted against it. The second version obviously appears more secure, and is not vulnerable to these attacks as such. However it is not clear whether the new output function C is strong enough to prevent a distinguishing attack derived from our attack of Section 5 (but probably much less efficient than this last).

A major characteristic of our attacks is that they exploit the fact that one of the LFSRs can stay unchanged for two consecutive cycles. This property of the cipher is easy to prevent; for example, we can replace equation (3) by

$$D_t = D'_t \oplus D''_t, \quad a_t = 1 + (D_t \& 15), \quad b_t = 1 + (D_t \gg 4)$$

This change has been suggested by the author of DICING herself, but has not been integrated into a new specification of the cipher.

References

[1] ECRYPT Stream Cipher Project. Part of the ECRYPT Network of Excellence in Cryptology, European Commission project IST-2002-507932. `http://www.ecrypt.eu.org/stream/`.

[2] Li An-Ping. A New Stream Cipher: DICING. In *Proceedings of the Symmetric Key Encryption Workshop*. Åarhus, Denmark, May 2005.

[3] Li An-Ping. A New Stream Cipher: DICING. Available at `http://www.ecrypt.eu.org/stream/dicing.html`.

A About Possible False Alarms

In this appendix, we consider the case where the attacker falsely assumes $a_t = 0$ (resp. $b_t = 0$), or falsely guesses the value of b_t (resp. a_t) when the first assumption is correct.

In the first case, i.e. if neither a_t nor b_t equals 0, $z_{t-1} \oplus z_{t+1}$ is nonlinear in either ω_t or τ_t, which makes very unlikely that the computed candidates for τ_t and ω_t have anything to do with their actual values. They can be considered as random.

The case where the assumption $a_t = 0$ (resp. $b_t = 0$) is correct and equation (9) (resp. (10)) is satisfied, but the guess on b_t (resp. a_t) is wrong, is more interesting. Consider two cycles t and t' such that (9) is satisfied. For the actual values b_t and $b_{t'}$ we have

$$\begin{cases} \tau_t = (z_{t-1} \oplus z_{t+1}) \cdot (1 \oplus x^{b_t})^{-1} \\ \tau_{t'} = (z_{t'-1} \oplus z_{t'+1}) \cdot (1 \oplus x^{b_{t'}})^{-1} \end{cases} \tag{12}$$

with

$$\tau_t = x^n \cdot \tau_{t'} \tag{13}$$

for some n. For false guesses b_t^* and $b_{t'}^*$, the attacker computes wrong values τ_t^* and $\tau_{t'}^*$:

$$\begin{cases} \tau_t^* = (z_{t-1} \oplus z_{t+1}) \cdot (1 \oplus x^{b_t^*})^{-1} \\ \tau_{t'}^* = (z_{t'-1} \oplus z_{t'+1}) \cdot (1 \oplus x^{b_{t'}^*})^{-1} \end{cases} \tag{14}$$

Putting equations (12), (13), (14) together, we obtain:

$$\tau_t^* \cdot \frac{1 \oplus x^{b_t^*}}{1 \oplus x^{b_t}} = x^n \cdot \tau_{t'}^* \cdot \frac{1 \oplus x^{b_{t'}^*}}{1 \oplus x^{b_{t'}}} \tag{15}$$

So when $b_t = b_{t'}$, there are 14 wrong guesses on the history of Γ_4: those corresponding to $b_t^* = b_{t'}^* \neq b_t$. However it happens with probability $1/15$ only, and this problem can be solved by neglecting cycle t', and finding another clock cycle t'' such that (9) is satisfied, with $b_t \neq b_{t''}$.

The eSTREAM Software Performance Testing

Christophe De Cannière

IAIK Krypto Group, Graz University of Technology
Inffeldgasse 16A, A–8010 Graz, Austria
christophe.decanniere@iaik.tugraz.at

Abstract. In this talk, we review the software performance testing efforts conducted by eSTREAM during the first phase of the project. We give an overview of the testing framework that was developed to streamline the evaluation, and briefly comment on the timing results observed on different software platforms for the various stream cipher candidates.

More information can be found online at http://www.ecrypt.eu.org/stream/perf/.

Comparison of 256-bit stream ciphers at the beginning of 2006

Daniel J. Bernstein *

djb@cr.yp.to

Abstract. This paper evaluates and compares several stream ciphers that use 256-bit keys: counter-mode AES, CryptMT, DICING, Dragon, FUBUKI, HC-256, Phelix, Py, Py6, Salsa20, SOSEMANUK, VEST, and YAMB.

1 Introduction

ECRYPT, a consortium of European research organizations, issued a Call for Stream Cipher Primitives in November 2004. A remarkable variety of ciphers were proposed in response by a total of 97 authors spread among Australia, Belgium, Canada, China, Denmark, England, France, Germany, Greece, Israel, Japan, Korea, Macedonia, Norway, Russia, Singapore, Sweden, Switzerland, and the United States.

Evaluating a huge pool of stream ciphers, to understand the merits of each cipher, is not an easy task. This paper simplifies the task by focusing on the relatively small pool of ciphers that allow 256-bit keys. Ciphers limited to 128-bit keys (or 80-bit keys) are ignored. See Section 2 to understand my interest in 256-bit keys.

The ciphers allowing 256-bit keys are CryptMT, DICING, Dragon, FUBUKI, HC-256, Phelix, Py, Py6, Salsa20, SOSEMANUK, VEST, and YAMB. I included 256-bit AES in counter mode as a basis for comparison. Beware that there are unresolved claims of attacks against Py (see [4] and [3]), SOSEMANUK (see [1]), and YAMB (see [5]).

ECRYPT, using measurement tools written by Christophe De Cannière, has published timings for each cipher on several common general-purpose CPUs. The original tools and timings used reference implementations (from the cipher authors) but were subsequently updated for faster implementations (also from the cipher authors). I extended the list of CPUs and then wrote a few extra tools, now available from `http://cr.yp.to/streamciphers.html#timings`, to convert ECRYPT's timings into the tables and graphs shown in Section 3.

Section 4 discusses several other interesting cipher features. For example, some ciphers have "free" built-in message authentication, so users can avoid the cost of computing a separate authenticator. One can and should quantify this benefit by making a separate table of timings for authenticated encryption; I plan to do this in subsequent comparison papers.

* Permanent ID of this document: `eff0eb8eebacda58462948ab97ca48a0`. Date of this document: 2006.01.23. This document is final and may be freely cited.

2 Why use 256-bit keys?

Some readers may wonder why I am not satisfied with 128-bit keys. Haven't I heard that—without massive advances in computer technology—a brute-force attack will never find a 128-bit key? After all, if checking about 2^{20} keys per second requires a CPU costing about 2^6 dollars, then searching 2^{128} keys in a year will cost an inconceivable 2^{89} dollars.

Answer: Even without advances in computer technology, the attacker does not need to spend 2^{89} dollars. Here are three reasons that lower-cost attacks are a threat:

- The attacker can succeed in far fewer than 2^{128} computations. He reaches success probability p after just $2^{128}p$ computations.
- More importantly, each key-checking circuit costs far less than 2^6 dollars, at least in bulk: 2^{10} or more key-checking circuits can fit into a single chip, effectively reducing the attacker's costs by a factor of 2^{10}.
- Even more importantly, if the attacker simultaneously attacks (say) 2^{40} keys, he can effectively reduce his costs by a factor of 2^{40}.

One can counter the third reduction by putting extra randomness into nonces, but putting the same extra randomness into keys is less expensive.

See [2] for a much more detailed discussion of these issues.

3 Speed

Ciphers in the tables in this section are sorted by a low-level feature, namely the number of bytes of state recorded between blocks. At one extreme is HC-256, which expands a key and nonce into a pair of 4096-byte arrays, making several array modifications for each block. At the other extreme is Salsa20, which simply records a key, nonce, and block counter in a 64-byte array, performing computations anew for each block. Most ciphers lie somewhere in the middle.

This ordering is not meant to imply that one extreme is better than the other. A large state has both advantages and disadvantages: it is expensive to set up and maintain, but it is also expensive for the attacker to analyze.

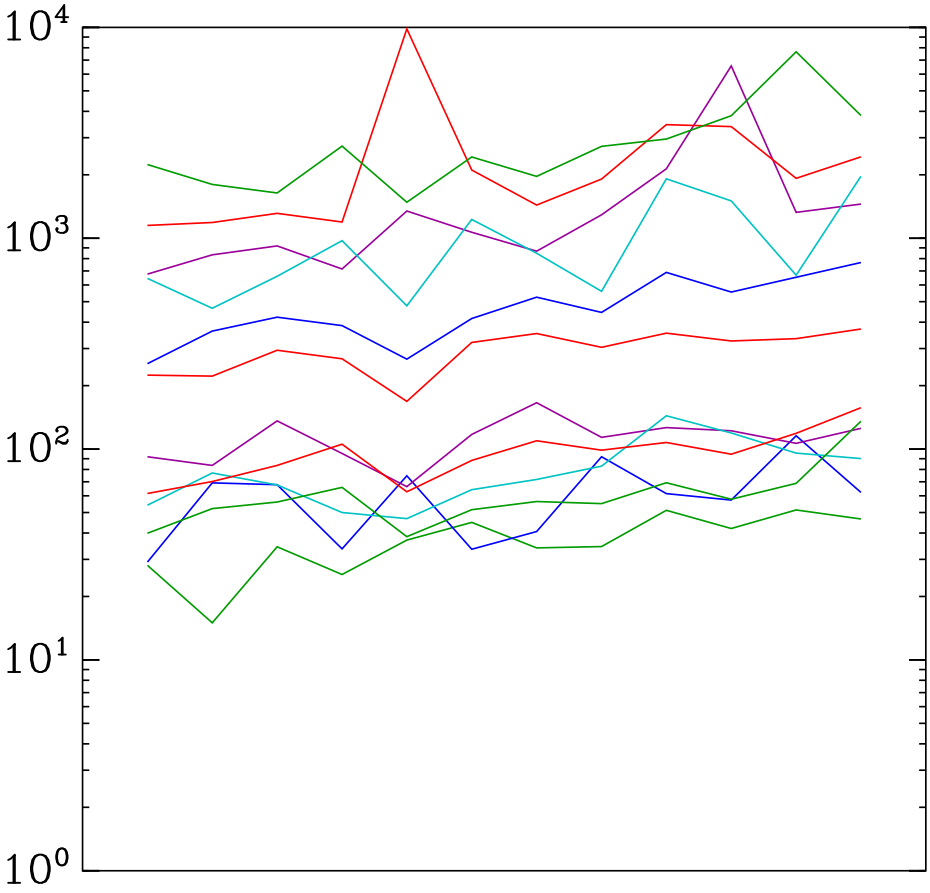
Table entries measure times for key setup, nonce setup, and encryption. All times are expressed as the number of cycles per encrypted byte. Smaller numbers are better here. Lines vary in how much setup they include, how many bytes are encrypted, and which CPU is measured. Bonus for readers using color displays: red means slower than AES; blue means faster than AES; lighter blue means twice as fast as AES; green means three times faster than AES.

FUBUKI has been omitted from the tables in this section. VEST has been omitted from the tables and graphs in this section. The cycle counts for FUBUKI and VEST are too large to be interesting.

	Salsa 20	Phelix	AES	Dragon	YAMB	SOSEMANUK	Py6	Crypt MT	Py	DICING	HC-256
Bytes	64	132	260	284	424	452	1124	3020	4196	4396	8396
Set up key, set up nonce, and encrypt 40-byte packet:											
A64	28.1	29.1	39.9	61.6	644.7	54.2	91.9	675.6	224.3	254.3	2236.5
PPC G4	15.0	69.1	52.2	70.2	465.1	77.0	83.7	834.4	221.9	362.6	1800.6
PM 695	34.4	67.8	56.1	83.7	659.4	67.6	136.0	919.0	294.1	422.1	1638.4
Athlon	25.4	33.6	65.8	105.5	974.3	50.0	95.3	714.5	268.1	385.0	2733.0
HP	37.0	74.7	38.4	62.7	478.0	46.8	66.3	1345.9	168.4	266.9	1481.0
P4 f41	44.9	33.5	51.6	88.4	1227.6	64.2	117.3	1066.9	320.6	416.2	2429.0
P3 68a	34.0	40.6	56.4	109.5	849.0	71.8	166.0	868.8	353.4	525.5	1964.3
SPARC	34.5	92.0	55.1	98.8	560.0	83.0	113.7	1292.1	303.7	444.7	2728.8
P4 f29	51.2	61.5	69.2	107.4	1914.6	143.9	126.4	2134.2	354.8	688.6	2953.2
P4 f12	42.0	57.3	57.8	94.5	1504.0	119.2	122.2	6560.6	325.6	555.3	3811.1
Alpha	51.4	115.7	68.8	118.7	667.8	95.7	106.5	1327.2	334.1		7660.2
P1 52c	46.6	62.3	135.5	157.2	1967.1	90.1	125.4	1452.1	371.0	766.7	3822.1
Set up nonce and encrypt 40-byte packet:											
A64	26.6	20.9	32.2	58.6	639.7	24.3	62.5	673.5	155.0	252.6	2234.7
PPC G4	13.6	52.3	44.6	66.9	459.8	31.9	62.4	832.3	169.2	361.2	1798.4
PM 695	32.5	53.8	47.3	80.1	656.0	36.0	94.0	917.2	168.8	420.0	1636.8
Athlon	23.4	24.9	56.7	99.9	970.2	27.9	65.7	712.2	196.6	382.6	2730.6
HP	35.0	59.9	31.9	58.1	473.5	29.7	42.4	1343.3	111.0	265.4	1478.8
P4 f41	42.1	23.6	43.6	83.0	1221.5	39.3	83.3	1064.7	230.3	414.7	2427.0
P3 68a	32.6	30.0	48.0	103.1	845.2	38.3	97.2	867.0	164.7	524.0	1963.1
SPARC	33.0	67.4	40.9	91.4	554.0	43.4	76.4	1288.7	227.4	442.9	2726.0
P4 f29	48.5	43.9	55.9	98.8	1902.8	51.2	84.3	2131.6	245.6	686.0	2950.6
P4 f12	39.6	39.6	46.0	86.3	1497.3	46.1	90.1	6556.6	256.4	552.6	3808.6
Alpha	49.7	83.6	57.7	109.6	661.7	50.7	70.3	1322.3	237.2		7647.3
P1 52c	42.5	46.0	113.2	148.6	1959.9	54.2	76.0	1449.3	252.8	763.6	3818.9
Set up nonce and encrypt 576-byte packet:											
A64	9.2	6.1	25.4	24.0	62.0	8.3	10.0	60.1	16.5	27.4	159.3
PPC G4	4.4	17.1	35.0	28.9	44.7	10.3	9.2	74.6	16.6	38.9	130.5
PM 695	12.1	14.9	35.1	27.8	64.9	9.6	9.1	74.8	14.1	41.5	117.5
Athlon	10.7	7.3	44.7	37.3	90.0	8.8	10.4	64.6	19.5	39.5	194.7
HP	11.6	16.4	22.5	26.0	47.5	8.8	6.6	113.7	11.3	28.4	107.2
P4 f41	14.3	7.0	33.5	32.6	106.7	12.4	9.3	94.6	19.0	42.1	171.6
P3 68a	14.5	9.0	37.7	35.4	81.7	12.0	9.8	73.4	14.5	50.7	142.0
SPARC	14.5	21.0	31.8	46.2	54.7	14.0	11.4	110.8	22.6	45.4	197.4
P4 f29	19.8	12.6	40.2	34.6	165.7	13.5	9.0	164.3	20.0	72.5	206.2
P4 f12	17.3	12.0	37.2	31.0	143.4	12.8	11.7	471.5	24.0	66.8	270.0
Alpha	22.6	28.3	43.2	52.3	64.4	16.9	11.0	128.0	23.2		549.5
P1 52c	19.8	14.2	85.7	60.3	181.5	17.3	17.4	136.2	28.3	82.4	275.4

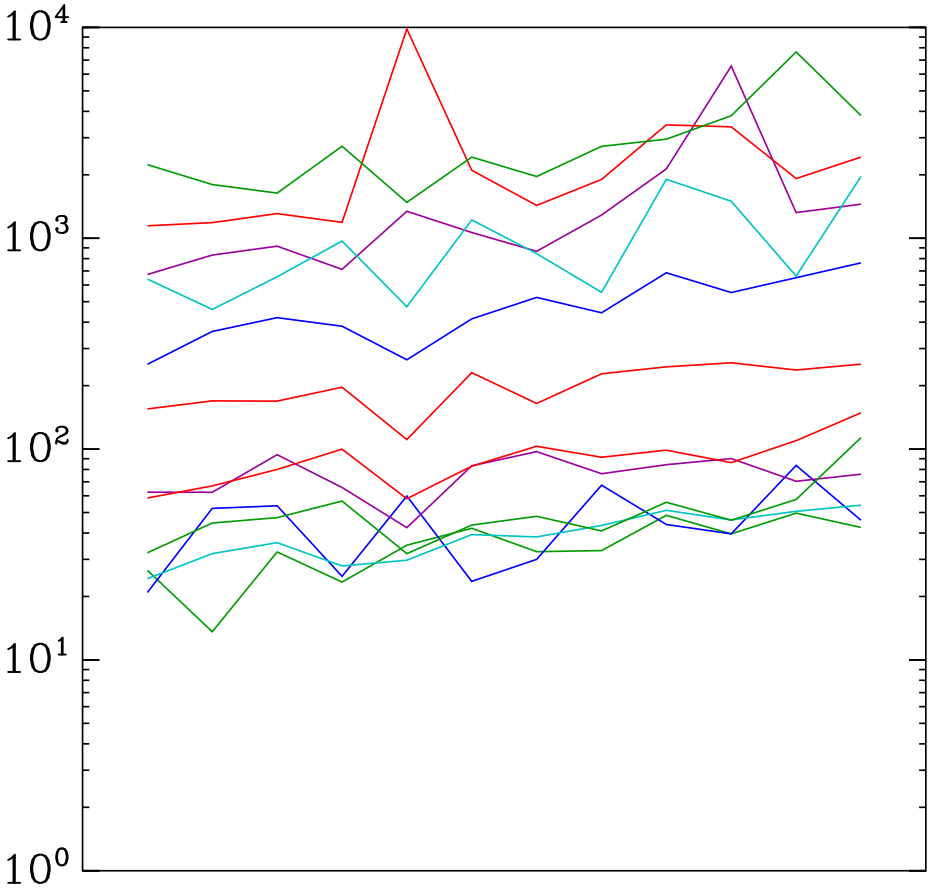
	Salsa 20	Phelix	AES	Dragon	YAMB	SOSEMANUK	Py6	Crypt MT	Py	DICING	HC-256
Bytes	64	132	260	284	424	452	1124	3020	4196	4396	8396
Set up nonce and encrypt 1500-byte packet:											
A64	9.4	5.4	25.4	22.3	35.5	7.3	7.7	28.0	10.1	17.2	64.2
PPC G4	4.5	15.5	35.0	27.1	25.6	8.9	6.8	38.4	9.7	24.4	54.2
PM 695	12.3	13.1	35.0	25.4	37.8	8.1	5.2	34.6	7.0	24.4	48.0
Athlon	10.9	6.5	44.7	34.2	49.5	7.5	7.8	32.1	11.2	24.1	78.5
HP	12.0	14.4	22.5	24.6	28.0	7.4	5.0	59.7	6.7	17.7	44.6
P4 f41	14.7	6.0	33.2	30.3	49.4	10.7	5.9	44.2	9.8	25.6	68.9
P3 68a	14.8	8.0	37.6	32.3	46.7	10.2	5.8	35.6	7.6	29.3	58.6
SPARC	14.9	18.9	31.8	44.0	31.8	12.2	8.5	54.7	13.2	27.4	81.6
P4 f29	20.0	11.0	39.5	32.1	79.2	10.8	5.5	72.4	10.3	41.7	82.7
P4 f12	20.1	10.9	37.2	28.9	80.8	10.6	8.6	200.5	13.6	36.8	106.7
Alpha	23.2	26.0	43.2	49.6	36.9	15.0	8.4	70.7	13.1		222.7
P1 52c	20.1	12.7	89.4	51.2	95.6	15.2	15.7	65.3	20.3	51.3	113.1
Encrypt one long stream:											
A64	8.9	4.9	25.2	8.1	18.9	4.4	3.9	9.3	4.0	10.8	4.4
PPC G4	4.2	9.6	34.8	8.4	13.7	6.2	5.3	16.4	5.4	15.2	6.2
PM 695	11.8	12.1	34.7	12.9	20.8	5.2	2.9	10.2	2.7	13.6	4.4
Athlon	10.5	6.0	44.4	13.4	24.3	5.6	4.4	13.1	5.0	14.3	5.7
HP	11.4	23.0	22.3	6.2	15.3	6.1	4.3	24.6	4.2	10.9	5.3
P4 f41	13.9	5.6	33.1	12.3	16.5	5.7	3.8	16.1	3.7	14.7	5.0
P3 68a	14.3	7.5	37.4	14.3	24.9	6.2	3.3	12.6	3.2	15.7	6.5
SPARC	14.3	16.9	31.6	8.8	17.6	8.3	6.5	20.7	6.7	16.2	9.0
P4 f29	17.0	10.1	39.3	12.9	29.2	6.5	3.5	15.3	3.8	23.5	4.8
P4 f12	17.0	10.1	36.8	12.9	37.9	6.2	4.5	16.1	4.8	21.7	5.0
Alpha	22.5	19.9	42.9	12.7	19.7	13.9	6.7	38.0	6.9		18.6
P1 52c	20.8	12.1	88.4	26.0	43.1	11.0	9.4	25.0	10.8	30.5	11.6
Encrypt many parallel streams in 256-byte blocks:											
A64	10.2	7.2	27.6	10.4	23.6	5.7	12.0	12.7	25.0	24.5	18.2
PPC G4	4.9	12.3	37.7	10.1	17.2	7.2	13.4	23.7	31.3	35.6	27.6
PM 695	12.8	14.5	37.7	15.1	25.5	6.3	10.7	17.1	26.7	31.1	21.3
Athlon	12.4	9.5	48.6	16.8	31.2	7.4	16.8	26.5	41.2	41.9	34.7
HP	12.1	24.7	24.9	8.1	18.4	7.3	8.3	28.8	14.7	23.1	17.8
P4 f41	16.4	9.3	37.1	16.2	24.0	7.5	12.8	23.8	26.4	38.4	28.6
P3 68a	15.8	11.6	43.3	19.9	37.6	7.8	25.3	41.1	77.3	58.4	55.2
SPARC	15.4	20.0	36.1	12.1	23.0	10.2	14.6	32.9	21.6	66.6	57.2
P4 f29	19.4	14.6	44.2	18.4	42.8	8.8	12.3	25.0	27.2	48.2	29.8
P4 f12	19.2	14.2	42.0	17.6	45.6	8.1	14.8	24.4	28.2	43.8	27.1
Alpha	23.4	22.4	49.2	15.5	24.9	15.0	15.1	38.4	36.0		50.0
P1 52c	21.3	14.7	85.8	27.2	47.1	12.1	18.0	29.3	39.5	50.3	33.5

Set up key, set up nonce, and encrypt 40-byte packet



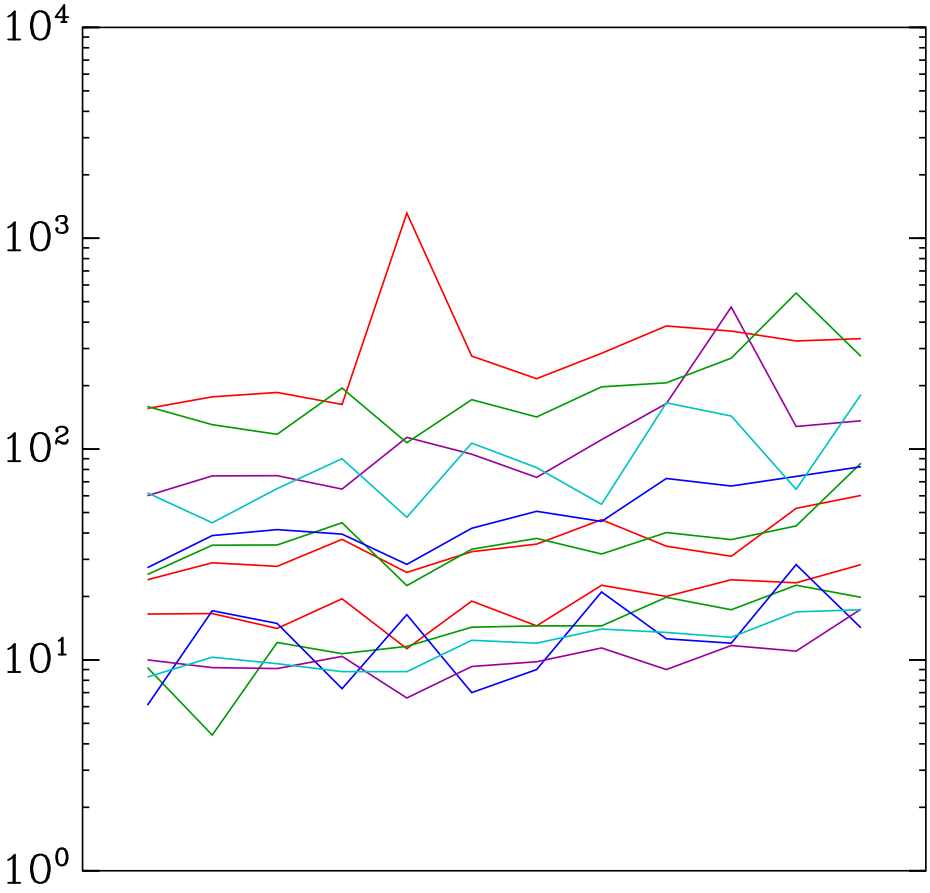
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c
HC	HC	HC	HC	FUB	HC	HC	HC	FUB	Cry		HC
FUB	FUB	FUB	FUB	HC	FUB	FUB	FUB	HC	HC	HC	FUB
Cry	Cry	Cry	YA	Cry	YA	Cry	Cry	Cry	FUB	FUB	YA
YA	YA	YA	Cry	YA	Cry	YA	YA	YA	YA	Cry	Cry
DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	YA	DIC
Py	Py	Py	Py	Py	Py	Py	Py	Py	Py	Py	Py
Py6	Py6	Py6	Dra	Phe	Py6	Py6	Py6	SOS	Py6	Dra	Dra
Dra	SOS	Dra	Py6	Py6	Dra	Dra	Dra	Py6	SOS	Phe	AES
SOS	Dra	Phe	AES	Dra	SOS	SOS	Phe	Dra	Dra	Py6	Py6
AES	Phe	SOS	SOS	SOS	AES	AES	SOS	AES	AES	SOS	SOS
Phe	AES	AES	Phe	AES	Sal	Phe	AES	Phe	Phe	AES	Phe
Sal	Sal	Sal	Sal	Sal	Phe	Sal	Sal	Sal	Sal	Sal	Sal

Set up nonce and encrypt 40-byte packet



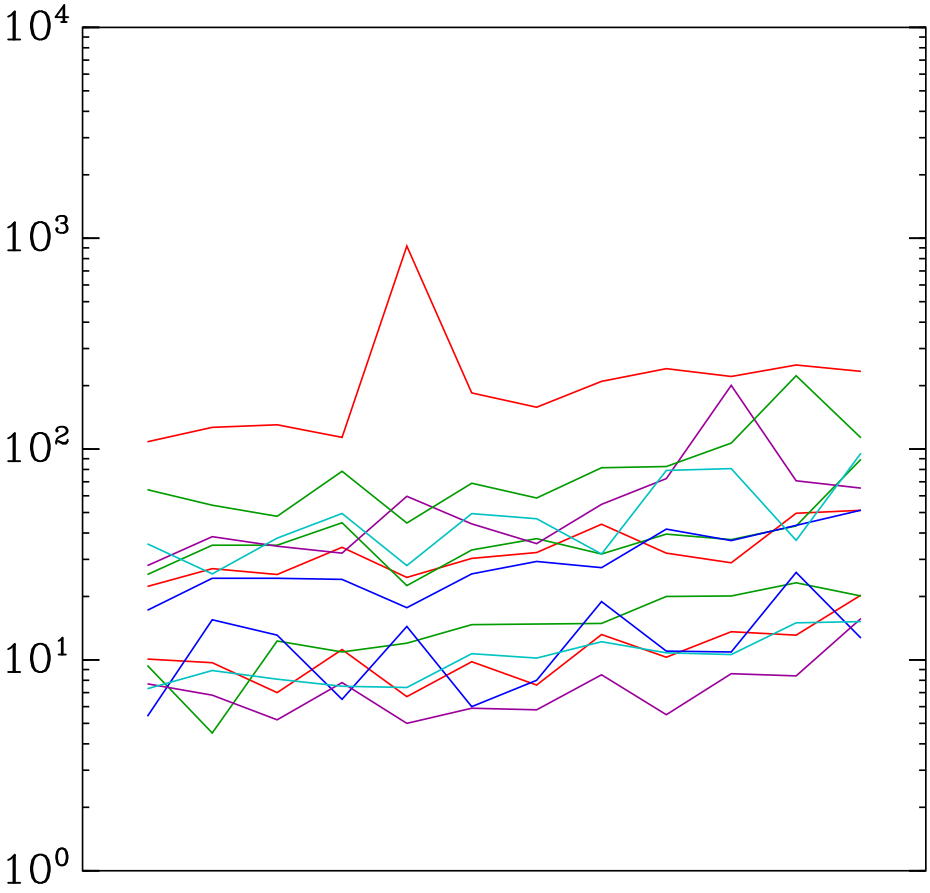
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c
HC	HC	HC	HC	FUB	HC	HC	HC	FUB	Cry		HC
FUB	FUB	FUB	FUB	HC	FUB	FUB	FUB	HC	HC	HC	FUB
Cry	Cry	Cry	YA	Cry	YA	Cry	Cry	Cry	FUB	FUB	YA
YA	YA	YA	Cry	YA	Cry	YA	YA	YA	YA	Cry	Cry
DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	YA	DIC
Py	Py	Py	Py	Py	Py	Py	Py	Py	Py	Py	Py
Py6	Dra	Py6	Dra	Phe	Py6	Dra	Dra	Dra	Py6	Dra	Dra
Dra	Py6	Dra	Py6	Dra	Dra	Py6	Py6	Py6	Dra	Phe	AES
AES	Phe	Phe	AES	Py6	AES	AES	Phe	AES	SOS	Py6	Py6
Sal	AES	AES	SOS	Sal	Sal	SOS	SOS	SOS	AES	AES	SOS
SOS	SOS	SOS	Phe	AES	SOS	Sal	AES	Sal	Sal	SOS	Phe
Phe	Sal	Sal	Sal	SOS	Phe	Phe	Sal	Phe	Phe	Sal	Sal

Set up nonce and encrypt 576-byte packet



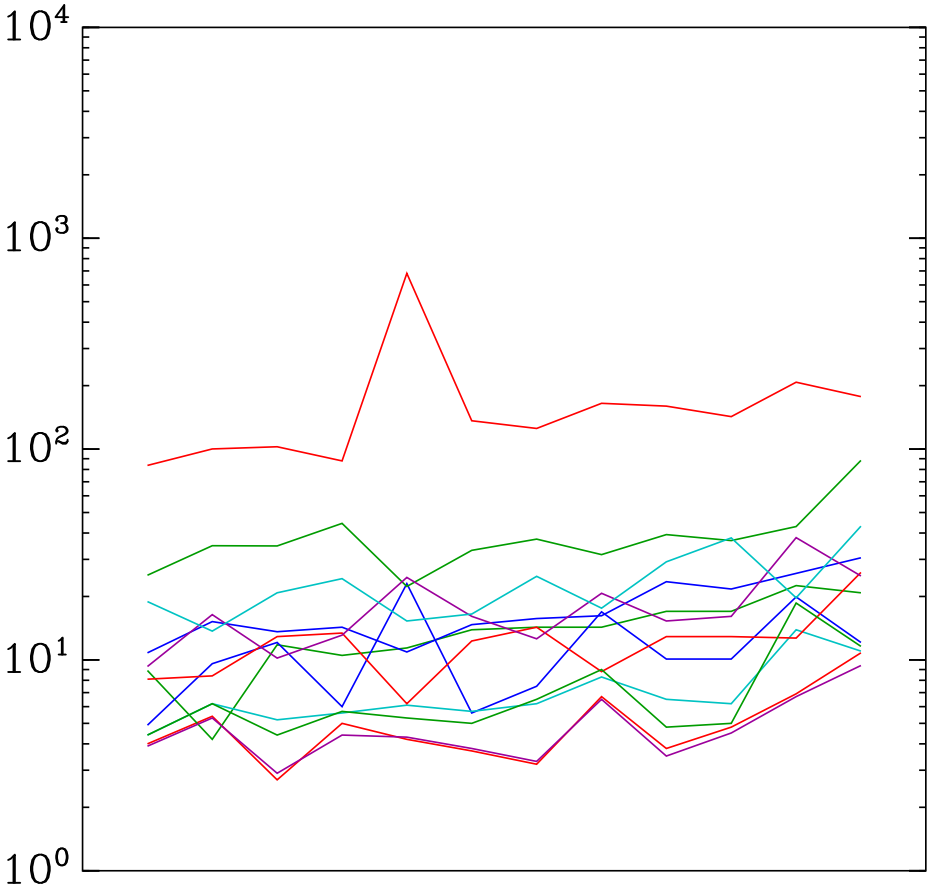
HC	FUB	FUB	HC	FUB	FUB	FUB	FUB	FUB	Cry		FUB
FUB	HC	HC	FUB	Cry	HC	HC	HC	HC	FUB	HC	HC
YA	Cry	Cry	YA	HC	YA	YA	Cry	YA	HC	FUB	YA
Cry	YA	YA	Cry	YA	Cry	Cry	YA	Cry	YA	Cry	Cry
DIC	DIC	DIC	AES	DIC	DIC	DIC	Dra	DIC	DIC	YA	AES
AES	AES	AES	DIC	Dra	AES	AES	DIC	AES	AES	Dra	DIC
Dra	Dra	Dra	Dra	AES	Dra	Dra	AES	Dra	Dra	AES	Dra
Py	Phe	Phe	Py	Phe	Py	Sal	Py	Py	Py	Phe	Py
Py6	Py	Py	Sal	Sal	Sal	Py	Phe	Sal	Sal	Py	Sal
Sal	SOS	Sal	Py6	Py	SOS	SOS	Sal	SOS	SOS	Sal	Py6
SOS	Py6	SOS	SOS	SOS	Py6	Py6	SOS	Phe	Phe	SOS	SOS
Phe	Sal	Py6	Phe	Py6	Phe	Phe	Py6	Py6	Py6	Py6	Phe
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c

Set up nonce and encrypt 1500-byte packet



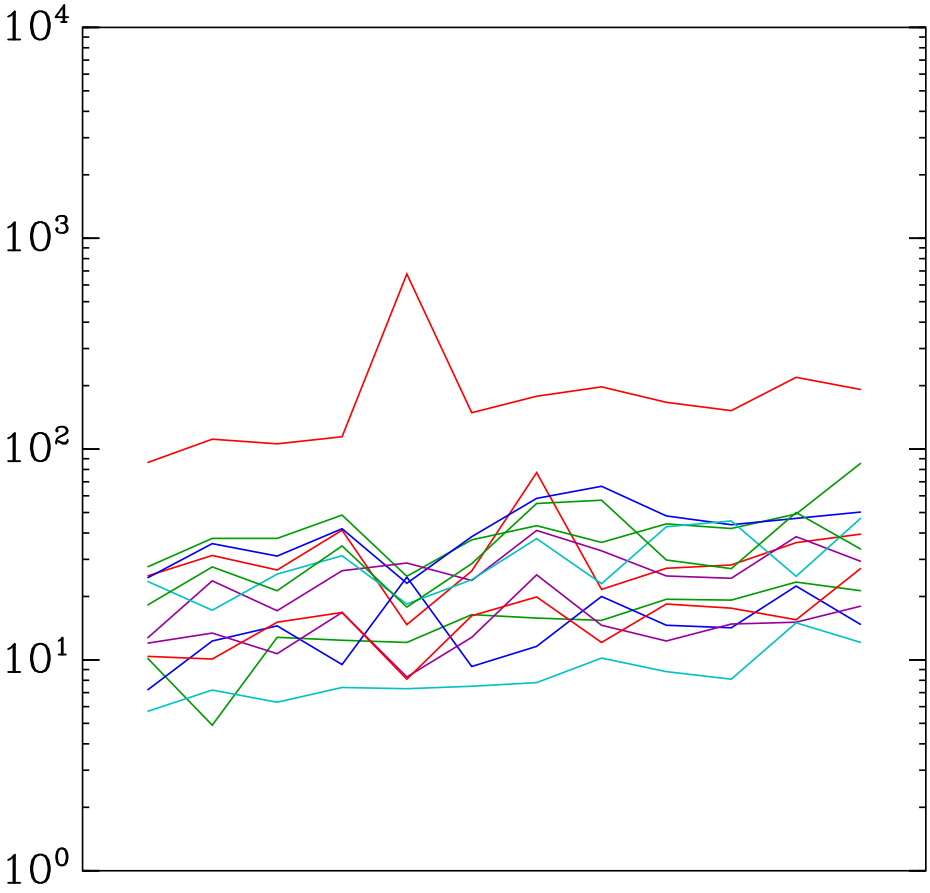
FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB		FUB
HC	HC	HC	HC	Cry	HC	HC	HC	HC	Cry	FUB	HC
YA	Cry	YA	YA	HC	YA	YA	Cry	YA	HC	HC	YA
Cry	AES	AES	AES	YA	Cry	AES	Dra	Cry	YA	Cry	AES
AES	Dra	Cry	Dra	Dra	AES	Cry	YA	DIC	AES	Dra	Cry
Dra	YA	Dra	Cry	AES	Dra	Dra	AES	AES	DIC	AES	DIC
DIC	DIC	DIC	DIC	DIC	DIC	DIC	DIC	Dra	Dra	YA	Dra
Py	Phe	Phe	Py	Phe	Sal	Sal	Phe	Sal	Sal	Phe	Py
Sal	Py	Sal	Sal	Sal	SOS	SOS	Sal	Phe	Py	Sal	Sal
Py6	SOS	SOS	Py6	SOS	Py	Phe	Py	SOS	Phe	SOS	Py6
SOS	Py6	Py	SOS	Py	Phe	Py	SOS	Py	SOS	Py	SOS
Phe	Sal	Py6	Phe	Py6	Py6	Py6	Py6	Py6	Py6	Py6	Phe
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c

Encrypt one long stream



FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB		FUB
AES	AES	AES	AES	Cry	AES	AES	AES	AES	YA	FUB	AES
YA	Cry	YA	YA	Phe	YA	YA	Cry	YA	AES	AES	YA
DIC	DIC	DIC	DIC	AES	Cry	DIC	YA	DIC	DIC	Cry	DIC
Cry	YA	Dra	Dra	YA	DIC	Sal	Phe	Sal	Sal	Sal	Dra
Sal	Phe	Phe	Cry	Sal	Sal	Dra	DIC	Cry	Cry	Phe	Cry
Dra	Dra	Sal	Sal	DIC	Dra	Cry	Sal	Dra	Dra	YA	Sal
Phe	SOS	Cry	Phe	Dra	SOS	Phe	HC	Phe	Phe	HC	Phe
SOS	HC	SOS	HC	SOS	Phe	HC	Dra	SOS	SOS	SOS	HC
HC	Py	HC	SOS	HC	HC	SOS	SOS	HC	HC	Dra	SOS
Py	Py6	Py6	Py	Py6	Py6	Py6	Py	Py	Py	Py	Py
Py6	Sal	Py	Py6	Py	Py	Py	Py6	Py6	Py6	Py6	Py6
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c

Encrypt many parallel streams in 256-byte blocks



FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB	FUB		FUB
AES	AES	AES	AES	Cry	DIC	Py	DIC	DIC	YA	FUB	AES
Py	DIC	DIC	DIC	AES	AES	DIC	HC	AES	DIC	HC	DIC
DIC	Py	Py	Py	Phe	HC	HC	AES	YA	AES	AES	YA
YA	HC	YA	HC	DIC	Py	AES	Cry	HC	Py	Cry	Py
HC	Cry	HC	YA	YA	YA	Cry	YA	Py	HC	Py	HC
Cry	YA	Cry	Cry	HC	Cry	YA	Py	Cry	Cry	YA	Cry
Py6	Py6	Dra	Py6	Py	Sal	Py6	Phe	Sal	Sal	Sal	Dra
Dra	Phe	Phe	Dra	Sal	Dra	Dra	Sal	Dra	Dra	Phe	Sal
Sal	Dra	Sal	Sal	Py6	Py6	Sal	Py6	Phe	Py6	Dra	Py6
Phe	SOS	Py6	Phe	Dra	Phe	Phe	Dra	Py6	Phe	Py6	Phe
SOS	Sal	SOS	SOS	SOS	SOS	SOS	SOS	SOS	SOS	SOS	SOS
A64	PPC G4	PM 695	Athl	HP	P4 f41	P3 68a	SP	P4 f29	P4 f12	Alpha	P1 52c

Notes on the timings

The tables and graphs use the following representative set of 12 machines, all with version 156 (2006.01.16) of ECRYPT's timing suite except where otherwise noted:

- A64: 2000MHz (one of two CPU cores) AMD Athlon 64 X2 (CPU identifier 15/43/1) named cph (gcc 4.0.2, Ubuntu 5.10).
- PPC G4: 533MHz (one of two CPUs) Motorola PowerPC G4 7410 named gggg (gcc 4.0.2, Ubuntu 5.10).
- PM 695: 1300MHz Intel Pentium M (695) named whisper (Fedora).
- Athlon: 900MHz AMD Athlon (622) named thoth (gcc 4.0.2, Ubuntu 5.10).
- HP PA: 440MHz (one of two CPUs) HP 9000/785 J5000 named hp400 (HP/UX).
- P4 f41: 3000MHz Intel Pentium 4 (f41) named pentium4b, timings collected by Christophe De Cannière.
- P3 68a: 1000MHz (one of two CPUs) Intel Pentium III (68a) named neumann (gcc 2.95.4 and gcc 3.0.4, Debian).
- SPARC: 900MHz Sun UltraSPARC III named wessel (SunOS 5.9).
- P4 f29: 2800MHz (one of two CPUs) Intel Pentium 4 (f29) named rzitsc (gcc 3.2.3, Red Hat).
- P4 f12: 1900MHz Intel Pentium 4 (f12) named fireball (gcc 4.0.2, Ubuntu 5.10).
- Alpha: 400MHz DEC Alpha EV5.6 21164A named alpha, using version 140 (2005.12.21), timings collected by Christophe De Cannière.
- P1 52c: 133MHz Intel Pentium (52c) named cruncher (gcc 4.0.2, Ubuntu 5.10).

The machines are sorted by the geometric average of all cipher cycle counts. This sorting accounts for the overall left-to-right upward trend in the graphs on previous pages.

See my web page `http://cr.yp.to/streamciphers.html#timings` for more comprehensive data. The web page includes speed reports for 24 machines; I'd also like to include timings for 8-bit CPUs and for ASICs. I will continue to update the web page as I receive newer information.

The graphs use cycles per byte, with a logarithmic scale, for the vertical axis. The labels below the graphs list ciphers in speed order. Consider, for example, the first graph: "Set up key, set up nonce, and encrypt 40-byte packet." The first column of the graph is labelled, from top to bottom, HC FUB Cry YA DIC Py Py6 Dra SOS AES Phe Sal A64. This column shows that, for setup and 40-byte encryption on an Athlon 64 (A64), HC-256 (HC) takes the most cycles per byte, and Salsa20 (Sal) takes the fewest cycles per byte. The graph shows that HC-256 takes about $2 \cdot 10^3$ cycles per byte while Salsa20 takes about $3 \cdot 10^1$ cycles per byte. The earlier table shows that HC-256 takes 2236.5 cycles per byte (i.e., 89460 cycles for 40 bytes) while Salsa20 takes 28.1 cycles per byte (i.e., 1124 cycles for 40 bytes).

4 Additional features

Bonus for readers using color displays: in this section, blue means an advantage compared to AES, and red means a disadvantage compared to AES.

AES in counter mode

Encryption. Unpatented. Variable time. 256-bit security conjecture. Security margin: has faster reduced-round versions; Ferguson et al. reported an attack on 7 out of 14 rounds; as far as I know, all claimed attacks on 8 rounds actually have worse price-performance ratio than brute-force search; there are no public claims of attacks on 9 rounds.

CryptMT

Encryption. Patented. Constant time. 256-bit security conjecture. No explicit security margin.

DICING

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin.

Dragon

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin.

FUBUKI

Encryption. Patented. Variable time. 256-bit security conjecture. No explicit security margin.

HC-256

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin.

Phelix

Authenticated encryption. Unpatented. Constant time. 128-bit security conjecture. No explicit security margin.

Py

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin. Attacks: Sekar, Paul, and Preneel in [4] reported an attack on Py using 2^{88} output bytes and comparable time. Crowley in [3] reduced 2^{88} to 2^{72}. The authors have not yet responded.

Py6

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin. Attacks: The attacks on Py by Sekar et al. can, presumably, be extended to Py6.

Salsa20

Encryption. Unpatented. Constant time. 256-bit security conjecture. Security margin: has faster reduced-round versions; Crowley reported an attack on 5 out of 20 rounds; there are no public claims of attacks on 6 rounds.

SOSEMANUK

Encryption. Unpatented. Variable time. 128-bit security conjecture. No explicit security margin. Attacks: Ahmadi, Eghlidos, and Khazaei in [1] reported an attack on SOSEMANUK using 2^{226} simple operations—but this doesn't disprove the original 128-bit security conjecture for SOSEMANUK. The authors have not yet responded.

VEST

Authenticated encryption. Patented. Variable time. 256-bit security conjecture. No explicit security margin.

YAMB

Encryption. Unpatented. Variable time. 256-bit security conjecture. No explicit security margin. Attacks: Wu and Preneel in [5] reported an attack on YAMB requiring 2^{58} output blocks and comparable time. There has been no response from the authors after six months.

5 Recommendations

Py, Py6, SOSEMANUK, and YAMB don't appear to provide 256-bit security. Unless there's a dispute regarding the attacks on these ciphers, they should be eliminated from consideration, at least as competition for 256-bit AES.

FUBUKI has no apparent advantages over AES and is several times slower. Unless there are dramatic speedups in the FUBUKI software, FUBUKI should be eliminated from consideration.

VEST is painfully slow in software but is claimed to provide considerably better performance in hardware. I haven't seen a careful evaluation of hardware performance, so I won't make any recommendations now regarding VEST.

The remaining 256-bit stream ciphers are CryptMT, DICING, Dragon, HC-256, Phelix, and Salsa20. Each of these ciphers provides better performance than AES for long streams, and some of them provide better performance than AES in other situations.

I recommend keeping all six ciphers—CryptMT, DICING, Dragon, HC-256, Phelix, and Salsa20—under consideration. One might be tempted to say, e.g., "CryptMT is practically always slower than Phelix and should be eliminated," but this will sound quite silly in retrospect if Phelix turns out to be breakable. The initial stream-cipher submission deadline was only eight months ago; the Py and SOSEMANUK attacks were published only a month ago; obviously we need more time for cryptanalysis.

References

1. Hadi Ahmadi, Taraneh Eghlidos, Shahram Khazaei, *Improved guess and determine attack on SOSEMANUK*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/085 (2005). URL: `http://www.ecrypt.eu.org/stream`. Citations in this paper: §1, §4.
2. Daniel J. Bernstein, *Understanding brute force* (2005). URL: `http://cr.yp.to/papers.html#bruteforce`. ID `73e92f5b71793b498288efe81fe55dee`. Citations in this paper: §2.
3. Paul Crowley, *Improved cryptanalysis of Py* (2006). URL: `http://www.ciphergoth.org/crypto/py/`. Citations in this paper: §1, §4.
4. Gautham Sekar, Souradyuti Paul, Bart Preneel, *Distinguishing attacks on the stream cipher Py*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/081 (2005). URL: `http://www.ecrypt.eu.org/stream`. Citations in this paper: §1, §4.
5. Hongjun Wu, Bart Preneel, *Distinguishing attack on stream cipher Yamb*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/043 (2005). URL: `http://www.ecrypt.eu.org/stream`. Citations in this paper: §1, §4.

Statistical Analysis of Synchronous Stream Ciphers

Meltem Sönmez Turan, Ali Doğanaksoy, Çağdaş Çalık

Institute of Applied Mathematics,
Middle East Technical University, Ankara, Turkey,
{msonmez, aldoks, e110870}@metu.edu.tr

Abstract. Synchronous stream ciphers produce long keystreams to be XORed with plaintext. The output keystreams should be indistinguishable from truly random sequences and should not leak any information about the secret key and the internal state of the cipher. In this study, we propose six new statistical tests to evaluate the randomness properties of synchronous stream ciphers. We applied four of these tests to the ciphers presented to ECRYPT and tabulated the results.
Keywords: Synchronous stream ciphers, statistical randomness testing

1 Introduction

Synchronous stream ciphers are an important class of symmetric encryption algorithms. Their basic design philosophy is inspired by the perfectly secure One Time Pad cipher in which the plaintext is encrypted with a random keystream using the XOR operation. It is the only cipher known to be unconditionally secure provided that keystream is truly random. Truly random keystreams remove the statistical weaknesses of the plaintext. The requirement of a keystream not shorter than the plaintext, distributing it securely in advance and not recycling the key make the cipher impractical. The motivation of generating a long pseudo-random keystream using a short random key is employed to overcome these disadvantages. Stream ciphers are used as an approximation to the action of One Time Pad. They do not provide the theoretical security of One Time Pad, but they are very practical. Therefore, the design goal of a synchronous stream cipher is to efficiently generate pseudo-random bits which are practically indistinguishable from truly random bits.

For a truly random generator, the number of ones and zeros in the output are equal. It is possible to formulate many other statistical properties that describe the keystream generated by a random source. Golomb [1] proposed three postulates for the structure of periodic pseudo-random sequences. It is clear that these three postulates are not sufficient to describe random looking sequences. A variety of different statistical tests can be applied to a keystream to evaluate the statement that the stream is generated by a truly random source.

Various test suites [2–5] are available in the literature. Knuth [2] presented several empirical tests including; frequency, serial, gap, poker, coupon collector's, permutation, run, maximum-of-t, collision, birthday spacings and serial

correlation. DIEHARD Battery of Tests consists of 18 independent statistical randomness tests including; birthday spacings, overlapping 5-permutations, binary rank, bitstream test, monkey tests on 20-bit Words, monkey tests OPSO, OQSO, DNA, count the 1's in a stream of bytes, count the 1's in specific bytes, parking lot, minimum distance, 3D spheres, squeeze, overlapping sums, runs and craps [3]. Also, Crypt-X [4] suite which was developed in the Information Security Research Centre at Queensland University of Technology consists of frequency, binary derivative, change point, runs, sequence complexity and linear complexity tests. Lastly, NIST [5] Statistical Test Suite consists of 16 tests namely; frequency, block frequency, runs, longest run, matrix rank, spectral, non-overlapping template matchings, overlapping template matchings, universal test, Lempel-Ziv complexity, linear complexity, serial cumulative sums, runs, approximate entropy, random excursions and variants.

These statistical tests are designed to evaluate the randomness properties of a finite sequence. For the evaluation of block ciphers presented for AES, Soto [6] proposed nine different ways to generate large number of data streams from a block cipher and tested these streams using the statistical tests available in NIST test suite.

While testing the randomness properties of stream ciphers, the general approach is to generate a large amount of keystream and apply certain statistical tests. The keystream itself is directly used in the tests. Failing from these tests do not usually lead to key or internal state recovery, but can be used for distinguishing the keystream from a truly random one. This kind of testing can be considered as a black box approach, since the internal structure, key or Initialization Vector (IV) loading phases of the cipher is not taken into account.

In this study, we propose six new statistical tests to analyze the randomness properties of synchronous stream ciphers. These tests, rather than examining the randomness properties of the keystream solely, concentrate on the correlations between key, IV, internal state and keystream.

In 2005, a call for stream cipher primitives has been announced by European Network of Excellence for Cryptology (ECRYPT). From the synchronous stream ciphers presented for ECRYPT, we analyzed the followings; ABC [7], Achterbahn [8], CryptMT/Fubuki [9], Decim [10], Dicing [11], Dragon [12], Edon80 [13], F-FCSR [14], Frogbit [15], Grain [16], HC-256 [17], Hermes8 [18], Lex [19], Mag [20], Mickey [21], Mickey-128 [22], Mir-1 [23], NLS [24], Phelix [25], Polar Bear [26], Pomaranch [27], Py [28], Rabbit [29], Salsa20 [30], Sfinks [31], Sosemanuk [32], Trivium [33], TSC-3 [34], Vest [35], WG [36], Yamb [37] and ZK-Crypt [38].

In the next section, different statistical randomness testing approaches used by the candidates of ECRYPT are summarized. In Section 3, our approach is given in detail. The experimental results are presented in Section 4. Finally, the conclusion is given in Section 5.

2 Randomness Testing

A summary of the randomness testing approaches performed by the authors of the ciphers presented for ECRYPT is given in this section.

Anashin et al. [7] proved that the distribution of 32-bit words in the keystream of ABC is uniform. The empirical statistical test results given in NIST suite did not indicate any deviation from a random sequence. Also, the authors applied some statistical randomness tests to evaluate the propagation property of the cipher using both Hamming Distance and naive correlation. For a fixed key and for a key varying with each IV pair, 384 such sequences of length 10^6 were obtained and empirically evaluated. Results of NIST Statistical Test Suite and DIEHARD Battery of Tests did not show any deviation from random behavior. As a result of these tests, ABC shows strong propagation properties [7].

In [9], Fubuki and AES have been tested according to their bit diffusion property with small number of rounds. Using 4-round AES and 2-round Fubuki, the diffusion bias is eliminated.

In [12], the keystream of Dragon is tested by the statistical randomness tests given in Crypt-X. Authors applied the frequency, binary derivative, change point, subblock and runs tests to 30 keystreams of length 8 megabits. Additionally, the sequence and linear complexity tests were applied to 30 streams with 200 kilobits each. Dragon showed no deviation from randomness according to these results. Also, the output of the F-FCSR generator is tested using the NIST Statistical Test Suite [14].

Theoretical validation for diffusion criteria in the initialization state has been done for Grain to defeat statistical chosen-IV attacks [16].

Wu [17] concentrated on distinguishing attacks while analyzing the randomness of HC-256. Keystreams with no linear masking and weakened feedback function are analyzed and it is concluded that distinguishing 2^{128} bits of keystream of HC-256 from a truly random sequence is computationally infeasible.

In [18], it is reported that the output of Hermes8 is tested using the FIBS 140-2 and DIEHARD Battery of Tests and no deviation from randomness is observed.

Vuckovac [20] reported that the output of Mag is tested for patterns in every stage of development by using statistical randomness tests available in ENT, DIEHARD and Crypt-X test suites. According to the results, no deviation from randomness is observed. The cipher Py had also been tested using statistical randomness tests [28]. It is claimed that the output keystream is uncorrelated and statistical tests should not succeed even when more extensive tests are made.

For the cipher Rabbit, the statistical tests from NIST, DIEHARD and ENT suites was applied. The tests were done for both the internal state and the keystream [29]. Also, various statistical tests were applied to the key setup function and also to the reduced version of Rabbit where each state variable has been given in 8 bits. Authors did not find any statistical weakness in any of these cases.

Hong et al. [34] reported that they had applied statistical randomness tests similar to the ones in NIST suite and had not found any weaknesses.

Bigeard et al. [35] tested the output of each component of Vest and claimed that individual streams of any of the outputs of Vest accumulators, combined Vest counters and complete Vest ciphers were indistinguishable from truly random sequences.

The randomness property of WG is given in terms of high period, balance, two-level autocorrelation, t-tuple distribution and linear complexity [36].

The keystream generated using ZK-Crypt passed from the statistical randomness tests of NIST and DIEHARD [38].

No statistical analyses are reported for the ciphers Achterbahn, Decim, Dicing, Edon80, Lex, Mickey, Mickey-128, Mir-1, NLS, Phelix, Polar Bear, Pomaranch, Salsa20, Sfinks, Sosemanuk, TRBDK3/YAEA, Trivium and Yamb in algorithm specification documents.

As summarized above, different testing approaches have been applied to the ciphers. While statistical analyzing, it is important to consider the relationship between key, IV, internal state and the keystream, since availability of the keystream and IV should not leak any information about the internal state or secret key.

In this study, we propose six new statistical tests for analyzing synchronous stream ciphers. The first test, *Key/Keystream Correlation Test*, considers the correlation between key and the corresponding keystream using a fixed IV. Similarly, the second test, *IV/Keystream Correlation Test*, considers the correlation between IV and the corresponding keystream using a fixed key. The third test, *Frame Correlation Test*, considers the correlation between keystreams using different IV values. The fourth test, *Diffusion Test*, examines the diffusion property of each bit of key and IV.

These four tests take key and IV as inputs and do not consider the internal state of the cipher. The following two tests, *Internal State Correlation Test* and *Internal State/Keystream Correlation Test*, consider the internal structure of the ciphers after key and IV loading phases. *Internal State Correlation Test* concentrates on the effect of similar IV values on the internal state using a fixed key and *Internal State/Keystream Correlation Test* examines the effect of internal state with low/high weight on the keystream weight.

3 Proposed Tests

Let S be a stream cipher with k-bit key, v-bit IV and n-bit internal state and let z_i and $(s_1, \ldots, s_n)$ represent the keystream and internal state, respectively.

Key/Keystream Correlation Test: The purpose of this test is to evaluate the correlation between the key and the first k bits of keystream. Firstly, m random keys are generated and IV is fixed. Next, a keystream of length k, $z_1, \ldots, z_k$ is produced for each key. Then, to evaluate their correlation, key and its corresponding keystream are XORed and weight of the resulting sequence is calculated. For a secure cipher, distribution of the weights is Binomial with parameters k and $1/2$. Low and high weight values indicate a correlation between i^{th} bit of key and i^{th} bit of keystream for $i = 1, \ldots, k$. However, the test does not consider

the correlations between i^{th} bit of key and j^{th} bit of keystream when $i \neq j$. The Chi-Square Goodness of Fit test is applied to evaluate this correlation. If the cipher fails from this test, key loading part of the initialization phase should be revised.

IV/Keystream Correlation Test: The purpose of this test is to evaluate the correlation between IV and the first v bits of keystream. Firstly, m random IVs are generated and key is fixed. Then, the keystream of length v is produced using each IV value and the fixed key. To evaluate the correlation, IV and its corresponding keystream are XORed and its weight is calculated. For a secure cipher, distribution of the weights is Binomial with parameters v and $1/2$. High correlation between IV and keystream may lead to generation of keystream without knowing the value of secret key. The Chi-Square Goodness of Fit test is applied to evaluate the correlation between IV and keystream. If the cipher fails from this test, IV loading part of the initialization phase should be revised.

Frame Correlation Test: In synchronous stream ciphers, after generating a fixed length keystream called *frame*, IV values are updated. Since IVs are commonly used as counters, two consecutive IV values are similar. The purpose of this test is to analyze the correlation between frames generated with similar IVs. In this test, first a random key and an IV value are chosen, then a keystream of length L is produced. This procedure is repeated N times with incremented values of IV. Using these keystreams, a matrix of size $N \times L$ is generated and the column weights of the matrix are calculated. Distribution of the weights is approximately normal with mean $N/2$ and variance $N/4$, when N is large. Columns with very high/low weight indicate weaknesses due to insecure resynchronization. The Chi-Square Goodness of Fit test is applied to evaluate the correlation between frames. If the cipher fails from this test, IV loading part of initialization phase should be revised.

Diffusion Test: This test examines the diffusion property of each bit of key and IV on the keystream. To satisfy diffusion, each bit of IV and key should affect the keystream. Minor changes in the IV or key should result in random looking changes in the keystream. In the *Diffusion Test*, firstly, a random vector $(u_1, \ldots, u_k, u_{k+1}, \ldots, u_{k+v})$ is chosen, where the first k bits represent the key, and the remaining v bits represent the IV. Using this key and IV, a keystream of length L is generated. Then, $k+v$ new vectors are generated by the operation $(u_1, \ldots, u_{k+v}) \oplus e_i$, where e_i is the vector having 1 in the entry i and zero elsewhere. For each vector, keystream of length L is generated. Then, these keystreams are XORed with the original keystream. Using these vectors, a matrix of size $(k+v) \times L$ is obtained. This procedure is repeated N times and obtained matrices are added in $\Re$. For a secure cipher, the entries of the matrix follow a normal distribution with mean $N/2$ and variance $N/4$, when N is large. Entries with high/low value indicate poor diffusion properties of corresponding cells. The Chi-Square Goodness of Fit test is applied to the entries of the matrix to evaluate diffusion property. If the cipher fails from this test, initialization phase of the algorithm should be revised.

Internal State Correlation Test: The purpose of this test is to analyze the effect of similar IVs on the internal state of the cipher. The idea of the test is very similar to *Frame Correlation Test*. Firstly, key and IV values are chosen randomly, then the internal state $(s_1, \ldots, s_n)$ after key/IV loading is stored. This procedure is repeated $M-1$ times with incremented values of IV. A total of M internal state vectors are stored in a matrix of size $M \times n$. To evaluate the correlation between internal states, the column weights of the matrix are calculated. Distribution of the weights is approximately normal with mean $M/2$ and variance $M/4$, when M is large. The Chi-Square Goodness of Fit test is applied to evaluate the correlation of internal states. If the cipher fails from this test, initialization phase of the algorithm should be revised.

Internal State/Keystream Correlation Test: Attacks to stream ciphers try to obtain the secret key or the internal state of the cipher when a part of the keystream is given. If the attacker recovers the internal state of the cipher at time t, he can easily generate the remaining part of the keystream without knowing the secret key. So, availability of keystream should not leak any information about the internal state of the cipher. The main idea of this test is that at any time, if the internal state has a distinguishing property such as low/high weight, the following keystream part should behave randomly in terms of its weight. Firstly, M initial state vectors of length n with low/high weight are chosen randomly. Then, these random initial states are directly assigned to the internal state of the cipher, in other words, the key and IV loading phase of the cipher is totally omitted. For each initial state, keystream of length n is generated from the cipher and its weight is calculated. The weights should follow the normal distribution with mean $n/2$ and variance $n/4$, when M is large enough. Using the Chi-Square Goodness of Fit tests, these weights are evaluated. If the cipher fails from this test, keystream generation phase from internal state should be revised. Special care must be taken while assigning states. The initial states should be chosen among the possible internal states. Forbidden states such as assigning a zero vector to a linear feedback shift register should be avoided.

In the next section, experimental results of the first four tests are presented. Analyzing ciphers using *Internal State Correlation* and *Internal State/Keystream Correlation Test* are left as a future study.

4 Experimental Results

For the *Key/Keystream Correlation Test*, $m = 2^{20}$ keys are generated randomly. For each key, keystream of length k (80 or 128 bits) is generated using a zero vector as IV. Keys and their corresponding keystreams are XORed and their weights are calculated. The weight probabilities are computed using the Binomial distribution. Then, the weights are categorized into 5 groups with approximately equal probabilities and the correlation between key and keystream bits is evaluated using Chi-Square Goodness of Fit tests.

For the *IV/Keystream Correlation Test*, $m = 2^{20}$ IVs and a fixed key are generated randomly. For each IV, keystream of length v (64, 80 or 128 bits) is

generated. IVs and their corresponding keystreams are XORed and their weights are calculated. The probability of each weight is computed using the Binomial distribution. The weight values are categorized into 5 groups with approximately equal probabilities and the correlation between IV and keystream bits is evaluated using Chi-Square Goodness of Fit tests.

For the *Frame Correlation Test*, starting with the IV 0x00000001 and incrementing until the IV 0x00100000, 2^{20} keystreams of length 256 bits are generated with a fixed random key. Using these keystreams, a matrix of size $2^{20} \times 256$ is formed and column weights are calculated. The distribution of these weights is approximately normal with mean 2^{19} and variance 2^{18}). Weights are categorized into 5 groups with approximately equal probabilities and evaluated using Chi-Square Goodness of Fit tests.

Finally, for the *Diffusion Test*, a matrix of size $(k+v)\times 256$ is generated using 2^{10} random key and IV pairs. Using the Binomial distribution, the entries of the matrix are categorized into 5 groups with approximately equal probabilities and diffusion of key and IV bits are evaluated using Chi-Square Goodness of Fit tests.

These four tests are applied to the synchronous stream ciphers presented in ECRYPT and the results are given in Table 1. Most of the ciphers support various key and IV sizes. The selected alternatives are listed in the table. For further analysis, other key and IV sizes should be considered.

The table shows the p-values obtained from each test. P-values less than 0.01 indicate a possible weakness. Low p-values have been obtained from the ciphers Decim, F-FCRS-8, Frogbit, Mag and Zk-Crypt. For Decim, it is observed that key and the first k bit of keystream are positively correlated. Similar correlation between IV and keystream is also available for the cipher. As the result of *Frame Correlation Test*, deviation from expected distribution is observed. However, the cipher statistically satisfies the diffusion property. For F-FCRS-8, correlation between frames is observed. Moreover, lack of diffusion property of IV bits between 66 and 101 causes the cipher to fail from the *Diffusion Test*. According to our results, the cipher Frogbit does not satisfy the necessary diffusion property and the frames generated using different IVs are correlated. Due to the small IV size of Mag, the *IV/Key Correlation Test* is not applied. For Mag, the desired diffusion property is not satisfied by the IV values. Therefore, it fails from the last two tests. For Zk-Crypt, the 29^{th} and 30^{th} bits of IV and key do not satisfy the desired diffusion.

5 Conclusion

In this study, six new statistical randomness tests are proposed, four of them are applied to the synchronous ciphers presented for ECRYPT. Some deviations from expected values are observed due to some possible weaknesses in key/IV loading phases of the ciphers. Analyzing ciphers using *Internal State Correlation Test* and *Internal State/Keystream Correlation Test* is left as a future study.

Cipher	*Key Size*	*IV Size*	*Key/Keystream Correlation*	*IV/Keystream Correlation*	*Frame Correlation*	*Diffusion*
ABC v.2	128	128	0.601073	0.610270	0.032804	0.466065
Achterbahn	80	64	0.417178	0.117759	0.048505	0.993111
CryptMT	128	128	0.897359	0.957659	0.740576	0.511523
Decim	80	64	0.000000	0.000000	0.000000	0.696777
Dicing	128	64	0.159261	0.203056	0.583911	0.730663
Dragon	128	128	0.613571	0.640181	0.213892	0.146159
Edon80	80	64	0.994770	0.672348	0.854742	0.345438
F-FCRS-8	128	128	0.331626	0.185941	0.000000	0.000000
Frogbit	128	128	0.525744	0.416107	0.000000	0.000000
Fubuki	128	128	0.428248	0.295603	0.113781	0.810933
Grain	80	64	0.559919	0.192504	0.670431	0.714399
HC-256	128	64	0.367689	0.142642	0.128726	0.470896
Hermes8	128	128	0.691878	0.156081	0.054161	0.806776
LEX	128	128	0.466709	0.874932	0.791357	0.85092
Mag	128	32	0.909934	-	0.000000	0.000000
Mickey	80	64	0.588080	0.037922	0.777025	0.734788
Mickey-128	128	128	0.660162	0.903834	0.395561	0.530875
Mir-1	128	64	0.805644	0.859696	0.827476	0.990484
NLS	128	128	0.560680	0.520917	0.725241	0.328536
Phelix	128	128	0.771726	0.664038	0.254927	0.863882
Polar Bear	128	128	0.216437	0.321427	0.762572	0.342001
Pomaranch	128	64	0.784698	0.978887	0.572945	0.825298
Py	128	64	0.656513	0.594916	0.242581	0.049459
Rabbit	128	64	0.791524	0.444611	0.033308	0.292981
Salsa20	128	64	0.110543	0.968776	0.512680	0.595137
SFINKS	80	80	0.476098	0.033331	0.351140	0.724150
Sosemanuk	128	64	0.583909	0.369988	0.333554	0.448504
Trivium	80	64	0.097261	0.479771	0.968566	0.937681
TSC-3	128	64	0.660202	0.508506	0.571159	0.596460
Vest	128	64	0.611495	0.013717	0.747299	0.333582
WG	128	128	0.563085	0.162022	0.847017	0.880886
Yamb	128	64	0.416602	0.187911	0.477731	0.447853
Zk-Crypt	128	128	0.482789	0.113247	0.000000	0.000000

Table 1. Test Results

References

1. S. W. Golomb. *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, USA, 1981.
2. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1981.
3. G. Marsaglia. DIEHARD Statistical Tests. http://stat.fsu.edu/ geo/diehard.html.
4. Information Security Institute. Crypt-X, 1998. http://www.isi.qut.edu.au/resources/cryptx/.
5. A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. 2001. http://www.nist.gov.
6. J. Soto. Randomness testing of the AES candidate algorithms, 1999.
7. V. Anashin, Bogdanov A., Kizhvatov I., and Kumar S. ABC: A New Fast Flexible Stream Cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
8. B. Gammel, R. Göttfert, and O. Kniffler. The Achterbahn Stream Cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
9. M. Matsumoto, H. Mariko, T. Nishimura, and M. Saito. Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
10. C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sib. Decim, A New Stream Cipher for Hardware Applications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
11. L. An-Ping. A New Stream Cipher: Dicing. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
12. E. Dawson, K. Chen, M. Henricksen, W. Millan, L. Simpson, and S. Moon H. Lee. Dragon: A Fast Word Based Stream Cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
13. D. Gligoroski, S. Markovski, L. Kocarev, and M. Gusev. Edon80. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
14. T. Berger, F. Arnault, and C. Lauradoux. F-FCSR. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
15. T. Moreau. The Frogbit cipher, A data integrity algorithm. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
16. M. Hell, T. Johansson, and Willi Meier. Grain - A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
17. H. Wu. Stream Cipher HC-256. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
18. U. Kaiser. Hermes stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
19. A. Biryukov. A New 128-bit Key Stream Cipher LEX. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
20. R. Vuckovac. MAG My Array Generator (A New Strategy for Random Number Generation). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.

21. S. Babbage and M. Dodd. The Stream Cipher MICKEY (version 1). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
22. S. Babbage and M. Dodd. The Stream Cipher MICKEY-128 (version 1). eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
23. A. Maximov. A new stream cipher Mir-1. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
24. G. Rose, P. Hawkes, M. Paddon, and M. W. de Vries. Primitive specification for NLS. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
25. D. Whiting, B. Schneier, S. Lucks, and F. Muller. Phelix, fast encryption and authentication in a single cryptographic primitive. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
26. J. Hastad and M. Naslund. The stream cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
27. C. Jansen and A. Kolosha. Cascade jump controlled sequence generator. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
28. E. Biham and J. Seberry. Py: A fast secure stream cipher using rolling arrays. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
29. M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner. The stream cipher rabbit. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
30. D. J. Bernstein. Salsa20 design. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
31. A. Braeken, J. Lano, N. Mentens, B. Preneel, and I. Verbauwhede. SFINKS: A synchronous stream cipher for restricted hardware environments. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
32. C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M Minier, T. Pornin, and H. Sibert. Sosemanuk, a fast software-oriented stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
33. C. De Cannire and B. Preneel. Trivium specifications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
34. J. Hong, D. H. Lee, Y. Yeom, D. Han, and S. Chee. T-function based stream cipher TSC-3. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
35. C. Bigeard, S. O'Neil, B. Gittins, and H. Landman. VEST hardware dedicated stream ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
36. G. Gong and Y. Nawaz. The WG stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
37. LAN Crypto. Primitive specifications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.
38. C. Gressel, R. Granot, and G. Vago. Zk-crypt - a compact stream cipher and more. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream.

d-Monomial Tests are Effective Against Stream Ciphers

Markku-Juhani O. Saarinen

Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK.
m.saarinen@rhul.ac.uk

Abstract. d-Monomial tests are statistical randomness tests based on Algebraic Normal Form representation of a Boolean function, and were first introduced by Filiol in 2002. We show that there are strong indications that the Gate Complexity of a Boolean function is related to a bias detectable in a d-Monomial test. We then discuss how to effectively apply d-Monomial tests in chosen-IV attacks against stream ciphers. Finally we present results of tests performed on eSTREAM proposals, and show that six of these new ciphers can be broken using the d-Monomial test in a chosen-IV attack. Many ciphers even fail a trivial (ANF) bit-flipping test.

Keywords: Stream Ciphers, eSTREAM, Algebraic Normal Form, Möbius test, d-monomial test.

1 Introduction

Statistical testing has traditionally been a part of evaluation of stream ciphers. However, most cryptographers agree that generic tests such as the NIST 800-22 suite are appropriate mainly for catching implementation errors rather than determining the cryptographic strength of an algorithm [4, 5].

Usually these tests have been performed in a passive setting; a sequence of bits is generated under a (random) key, and these bits are then subjected to a generic statistical test. What is ignored in this approach is that stream ciphers equipped with an Initialization Vector (IV) should also be able to withstand chosen-IV attacks, where a sequence of data is generated by varying the IV value rather than the "counter" value (see Figure 1).

Stream ciphers are optimized for security, but also for speed and cost. Cost in many applications equates to the number of logical gates in a hardware implementation of the cipher, and hence designers usually attempt to minimize their gate complexity.

Most stream ciphers can be specified as a relatively simple iterated function. As a result of this, it has been observed that some keystream bits can be expressed as simple Boolean functions of the key and IV bits. In a chosen-IV attack, the key bits remain constant and the stream cipher can be viewed as a "black box" Boolean function of the IV alone.

In a chosen-IV distinguishing attack, an attacker would wish to be able to determine whether or not a keystream bit (say, the first one after IV setup) is a simple Boolean function of some IV bits simply by making queries to this black box.

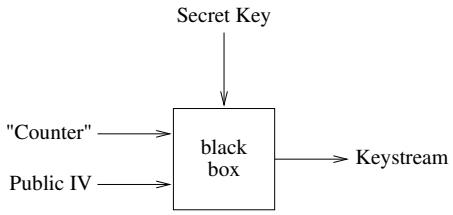


Fig. 1. A stream cipher can be seen as a black box Boolean function that takes in a secret key, a public IV, and a public "counter" to produce a single bit of keystream.

How would one automatically distinguish such a Boolean function of n bits from a random one? One solution is to examine its Algebraic Normal Form (ANF) representation for anomalies such as redundancy or bias. A test that utilizes this approach was first proposed by Eric Filiol in 2002 [2]. In this paper we will give further theoretical and experimental evidence of the applicability of ANF-based tests on stream ciphers.

The structure of this paper is as follows. In Section 2 we recall the Algebraic Normal Form and its basic properties. Section 3 contains an exposition of a variant of Filiol's d-monomial statistical test. Section 4 gives new, clear evidence of the relationship between Boolean gate complexity and the d-monomial test. Section 5 discusses a simple statistical attack based on flipping input bits that was found to be surprisingly effective against eSTREAM ciphers [3]. Section 6 contains new results on statistical tests on the 34 eSTREAM cipher proposals, followed by conclusions in Section 7.

2 Preliminaries

Let $\mathbb{F}_2^n$ be the vector space defined by n-vectors $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in \mathbb{F}_2$, i.e. each of the n elements has either value 0 or 1 and computations are defined modulo 2. A Boolean function f of n variables is simply a mapping $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$. There are exactly 2^{2^n} distinct Boolean functions of n variables, each uniquely defined by its truth table.

There are many alternative representations for Boolean functions, such as Conjunctive and Disjunctive Normal Forms (CNF and DNF), which are widely used in automated theorem proving and other fields of theoretical computer science. We will focus on Algebraic Normal Form (ANF, also known as Ring Sum Expansion, or RSE [6]). [1]

Definition 1. *A function* $\hat{f} : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ *satisfying*

$$\hat{f}(\mathbf{x}) = \sum_{\mathbf{a} \in \mathbb{F}_2^n} f(\mathbf{a}) \prod_{i=1}^{n} x_i^{a_i}$$

is an Algebraic Normal Form representation of a Boolean function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$.

[1] This transform is sometimes confusingly called the Möbius transform [2], hence the name, "Möbius test" in Filiol's original paper.

Using transformed function $\hat{f}$, a multivariate polynomial representation of f can be obtained as can be seen from the following example (or directly from the definition).

Example 1. Consider the Boolean function $f : \mathbb{F}_2^3 \mapsto \mathbb{F}_2$ defined by the following table:

$$f(0,0,0) = 1,\ f(1,0,0) = 0,\ f(0,1,0) = 1,\ f(1,1,0) = 0,$$
$$f(0,0,1) = 1,\ f(1,0,1) = 1,\ f(0,1,1) = 0,\ f(1,1,1) = 1.$$

As indicated by Definition 1, we wish to find a $\hat{f}$ that for all $\mathbf{x}$ satisfies

$$f(x_1,x_2,x_3) = \hat{f}(0,0,0) + \hat{f}(1,0,0)x_1 + \hat{f}(0,1,0)x_2 + \hat{f}(1,1,0)x_1x_2 +$$
$$\hat{f}(0,0,1)x_3 + \hat{f}(1,0,1)x_1x_3 + \hat{f}(0,1,1)x_2x_3 + \hat{f}(1,1,1)x_1x_2x_3.$$

this corresponds to solving the following system of linear equations in $\mathbb{F}_2$:

$$\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 1&1&0&0&0&0&0&0\\ 1&0&1&0&0&0&0&0\\ 1&1&1&1&0&0&0&0\\ 1&0&0&0&1&0&0&0\\ 1&1&0&0&1&1&0&0\\ 1&0&1&0&1&0&1&0\\ 1&1&1&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} \hat{f}(0,0,0)\\ \hat{f}(1,0,0)\\ \hat{f}(0,1,0)\\ \hat{f}(1,1,0)\\ \hat{f}(0,0,1)\\ \hat{f}(1,0,1)\\ \hat{f}(0,1,1)\\ \hat{f}(1,1,1) \end{pmatrix} = \begin{pmatrix} f(0,0,0)=1\\ f(1,0,0)=0\\ f(0,1,0)=1\\ f(1,1,0)=0\\ f(0,0,1)=1\\ f(1,0,1)=1\\ f(0,1,1)=0\\ f(1,1,1)=1 \end{pmatrix}.$$

The solution to this matrix equation is obtained easily with Gaussian elimination:

$$\hat{f}(0,0,0) = 1,\ \hat{f}(1,0,0) = 1,\ \hat{f}(0,1,0) = 0,\ \hat{f}(1,1,0) = 0,$$
$$\hat{f}(0,0,1) = 0,\ \hat{f}(1,0,1) = 1,\ \hat{f}(0,1,1) = 1,\ \hat{f}(1,1,1) = 1.$$

The ones in $\hat{f}$ directly give the five monomials in the polynomial expression for f:

$$f(x_1,x_2,x_3) = 1 + x_1 + x_1x_3 + x_2x_3 + x_1x_2x_3.$$

2.1 Properties of the Algebraic Normal Form

We briefly summarize some of the most important properties and concepts (facts) of ANF that are relevant to the present discussion:

F.1 A unique $\hat{f}$ exists for all Boolean functions f.
F.2 The ANF transform is its own inverse, an involution; iff $g = \hat{f}$, then $\hat{g} = f$.
F.3 We define a *partial order* for vectors $\mathbf{x}$ as follows: $\mathbf{x} \leq \mathbf{y}$ iff $x_i \leq y_i$ for all i. Using the partial order, Definition 1 can be written as $\hat{f}(\mathbf{x}) = \sum_{\mathbf{a} \leq \mathbf{x}} f(\mathbf{a})$.
F.4 The *Hamming distance* $\mathtt{d}(\mathbf{x},\mathbf{y})$ between $\mathbf{x}$ and $\mathbf{y}$ is the number of positions where $x_i \neq y_i$.
F.5 A norm, called the *Hamming weight*, $\mathtt{wt}(\mathbf{x}) = \mathtt{d}(\mathbf{0},\mathbf{x})$, is equivalent to number of positions in $\mathbf{x}$ where $x_i = 1$.

F.6 The *algebraic degree* $\deg(f)$ is the maximum Hamming weight $\mathbf{x}$ that satisfies $\hat{f}(\mathbf{x}) = 1$; this is equivalent to the length of the longest monomial (most variables) in the polynomial representation of f.

F.7 Functions of degree one are *affine functions*. If the constant term $\hat{f}(0, 0, \ldots, 0) = 0$, an affine function is simply a sum of some of its input bits and called a *linear function*.

F.8 A *d-Truncated Algebraic Normal Form* of Boolean function f, denoted $\hat{f}_d(\mathbf{x})$, is equal to $\hat{f}(\mathbf{x})$ when $\mathtt{wt}(\mathbf{x}) \leq d$, and zero otherwise. In essence, monomials of degree greater than d have been removed from the corresponding polynomial of the truncated ANF.

F.9 Since $\hat{f}(\mathbf{x})$ is the sum of f at all positions with smaller or equal partial order (and hence degree) than $\mathbf{x}$ (F.3), it can be seen that if we have tabulated $f(\mathbf{y})$ at all positions $\mathbf{y}$ with $\mathtt{wt}(\mathbf{y}) \leq d$, the d-truncated ANF can be completely determined.

2.2 Computing the ANF

Networks and algorithms for computing the complete ANF do not require more than $n2^{n-1}$ additions in $\mathbb{F}_2$.

Let $z : \mathbb{F}_2^n \mapsto \mathbb{Z}$ be the standard mapping from binary vectors to integers; $z(\mathbf{x}) = \sum_{i=1}^{n} 2^{i-1} x_i$. Let v be a binary-valued vector of length 2^n that contains the truth table of f; $v_{z(\mathbf{x})+1} = f(\mathbf{x})$ for all $\mathbf{x}$. Algorithm 1 gives a fast method for computing $\hat{f}$.

Algorithm 1 Compute the Algebraic Normal Form in vector v of length 2^n using two auxiliary vectors t and u of length 2^{n-1}.

```
for j = 1, 2, 3, ..., n do
  for i = 1, 2, ..., 2^(n-1) do
    t_i ← v_(2i-1)
    u_i ← v_(2i-1) ⊕ v_(2i)
  end for
  v ← t || u
end for
```

The complexity of Algorithm 1 is clearly $O(n \lg n)$. Variants of this algorithm can be implemented very efficiently using shifts and bit-manipulation operations.

3 The *d*-Monomial Tests

In [2] Filiol introduced "Möbius tests", which examine whether or not an ANF expression of a Boolean function has the expected number of d-degree monomials. With $d = 0$ the test is called the *Affine test* and for $d > 0$ a *d-Monomial test*.

Please note that the following exposition of the test / distinguisher is significantly simpler and less formal than that originally proposed by Filiol. Details have been modified for the purposes of this paper. The reader is encouraged to use [2] as a reference for Filiol's version of the test.

In practical terms the d-Monomial test involves counting the number of ones $\hat{f}(\mathbf{x}) = 1$ of an ANF transformed function f at positions $\mathbf{x}$ with Hamming weight d. A d-truncated ANF is is sufficient for this purpose. A χ^2 statistical test is then applied to this count to see if the count is exceptionally high or low.

Theorem 1. *For a randomly chosen n-bit Boolean function f, $\mathsf{Pr}[\hat{f}(\mathbf{x}) = 1] = 1/2$ for all $\mathbf{x}$.*

Proof. Trivial. Since the ANF transformation is bijective on the truth table of f, $\hat{f}$ will be random if f is.

Consider an n - bit Boolean function f. Our null hypothesis is that the expected bitcount $\sum_{\mathsf{wt}(\mathbf{x})=d} \hat{f}(\mathbf{x})$ is $\frac{1}{2}\binom{n}{d}$ and the bitcount is binomially distributed. The alternative hypothesis is that there is a bias in this sum, up or down.

We can use Pearson's classic χ^2 test in this case. Suppose that we sample $\hat{f}$ at N distinct points (in this case with $\mathsf{wt}(\mathbf{x}) = d$) and in M of those $\hat{f}(\mathbf{x}) = 1$. Then we set

$$\chi^2 = \frac{1}{N}\left(2M - N\right)^2.$$

Since "0" and "1" cases in bitcount are mutually exclusive, there is one degree of freedom in the test. Using the cumulative degree-one distribution function of χ^2, we can determine a confidence level for f being distinguishable from random in our test. We call this the P value and its intuitive interpretation is the "probability that the null hypothesis is true". For example, if P is 0.01, there's still a 1% probability that the null hypothesis is true (and the function is, in this sense, "random").

Some "upper critical" values for χ^2 and the corresponding P values are given in the following table:

χ^2	P
6.635	0.01
10.83	0.001
18.70	2^{-16}
40.17	2^{-32}
24.02	2^{-40}
83.82	2^{-64}
105.8	2^{-80}

This type of test is dependent upon the sample size; even a very slightly biased function will yield a high χ^2 value by the test if the sample size is allowed to be arbitrarily large. The sample sizes are bound by computational restrictions, however. A distinguishing attack is not relevant unless its total expected computational complexity is smaller than the claimed security level of the cipher (typically equivalent to 2^{k-1} key trials, where k is the size of the secret key).

4 Gate Complexity and the d-Monomial Test

In this section we will give a formal definition for *gate complexity* and investigate its relationship with the d-Monomial test. Gate complexity is essentially equivalent to circuit complexity with realistic limitations [1, 6].

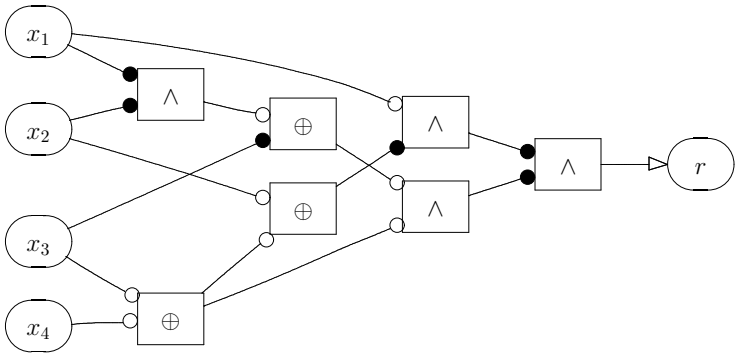


Fig. 2. An automatically generated picture of a Boolean function with gate complexity 7. In this picture a filled circle indicates that the given input is inverted. This function can not be implemented with, say, six gates (regardless of the choice of gates).

Definition 2. *Gate complexity of a Boolean function $f(x_1, x_2, \ldots, x_n)$ is the minimum number of gates required to implement it in an acyclic circuit network. A gate is a Boolean function with two inputs. The constant functions 0 and 1, together with trivial functions $x_1, x_2, \ldots$ have gate complexity 0.*

Note that all $2^{2^2} = 16$ two-bit functions count as a single gate, not just the standard ones ($\vee$, $\wedge$, $\neg$, $\oplus$).

We have determined the gate complexity of all $2^{2^4} = 65536$ four-bit Boolean functions. This was done by performing an exhaustive search over all circuits with one gate, two gates, etc, until circuits for all functions had been found. The task was computationally nontrivial, even though we optimized the code to take various symmetries and isometries into account. The maximum gate complexity turned out to be 7 (see Figure 2).

Table 1 gives the distribution of functions by gate complexity. In it, G_i is the number of functions of gate complexity i. These sum to $\sum_i G_i = 65536$. Here $g_{i,d}$ is the number of monomials of degree d and gate complexity i. These sum to $\sum_d g_{i,d} = G_i$. The maximum possible value for $g_{i,d}$ is $G_i\binom{4}{d}$. The expected number in a d-monomial test is half of this value. The table contains the "bias" fraction $q_{i,d} = g_{i,d}/(G_i\binom{4}{d})$.

Note how in Table 1 the d-Monomial "bias" $q_{i,d}$ tends to be strongly increasing as the gate complexity i grows (apart for anomaly at $q_{6,4}$). This is clear evidence of a correlation between the complexity of a Boolean circuit and the d-monomial test. It is plausible to expect that a similar phenomenon is exhibited by Boolean functions with $5, 6, \ldots$ inputs. However, the exact degree of this bias is currently an open problem for $n > 4$. We can expect simple functions to be distinguishable in a d-monomial test even when n is large.

It is interesting to note that it is even possible to test the opposite; to distinguish a *complex* function from a randomly chosen one, as the following example illustrates.

		$d=0$		$d=1$		$d=2$		$d=3$		$d=4$	
i	G_i	$g_{i,0}$	$q_{i,0}$	$g_{i,1}$	$q_{i,1}$	$g_{i,2}$	$q_{i,2}$	$g_{i,3}$	$q_{i,3}$	$g_{i,4}$	$q_{i,4}$
0	6	1	0.167	4	0.167	0	0.000	0	0.000	0	0.000
1	64	34	0.531	76	0.297	48	0.125	0	0.000	0	0.000
2	456	228	0.500	648	0.355	672	0.246	256	0.140	0	0.000
3	2474	1237	0.500	3912	0.395	5136	0.346	3264	0.330	832	0.336
4	10624	5312	0.500	18960	0.446	26976	0.423	17536	0.413	4608	0.434
5	24184	12092	0.500	47888	0.495	71328	0.492	47616	0.492	13216	0.546
6	25008	12504	0.500	52992	0.530	83232	0.555	55744	0.557	12576	0.503
7	2720	1360	0.500	6592	0.606	9216	0.565	6656	0.612	1536	0.565

Table 1. Distribution of the 65536 four-bit Boolean functions by gate complexity and the results of d-monomial tests on Boolean functions of given gate complexity.

Example 2. With the 2720 functions of gate complexity 7, all d-Monomial counts appear to be biased *upwards*; $q_{7,d} \geq 0.5$. We will use a d-Monomial test to create a distinguisher based on this fact, particularly that $q_{7,1} = 0.606$.

Consider the following game. There is a list L containing binary vectors of length 5. Entries in L are may have been generated with one of the following two methods:

1. Choose a random 4-bit Boolean function of gate complexity 7 for each entry, and add the following vector to the list

$$(f(0,0,0,0), f(1,0,0,0), f(0,1,0,0), f(0,0,1,0), f(0,0,0,1)).$$

2. Choose a completely random Boolean function (one of the 65536 possibilities) and create a vector in similar fashion.

We pose the following question: How long does L need to be for us to see which type of list it is ?

We first note that the vectors contain sufficient information for computation of 1-Monomial test (e.g. $\hat{f}(1,0,0,0) = f(0,0,0,0) + f(1,0,0,0)$). Each 1-Monomial test is simply the sum of 4 bits in the ANF result. The expected sum after n list entries is $2n$ for a random function and based on our exhaustive search, $g_{7,1}n/G_7 = 6592/2720n \approx 2.424n$ for a gate complexity 7 function. Our distinguisher will simply return “a” if the sum is greater than $2n$ and “b” otherwise.

In the second, fully random case, the distinguisher has no advantage as the bits in the vector are random too; “a” and “b” will both be returned with probability $1/2$ regadless of the length of L.

In case 1, after $n = 34$ steps, the sum can be expected to reach $2.424{*}34 = 82.4$. "a" will be returned by the distinguisher with probability 99%. Hence we can distinguish the list of (partially computed and randomly chosen) “complex” functions with significant certainty with a list of only 34 entries! Note that the probability here was computed exactly using binomial sums, rather than using the χ^2 test.

5 The (ANF) Bit-Flip Test

The bit-flip test is a simple statistical test that measures the effect of flipping one of the input bits on a Boolean function. The test can be performed either on the function f itself or its ANF counterpart $\hat{f}$.

The same "bit-counting" χ^2 test with one degree of freedom can be applied as in d-Monomial test (Section 3).

Given a vector with $\mathbf{b}$ with $\mathsf{wt}(\mathbf{b}) = 1$, we sample $f(\mathbf{x})$ (or $\hat{f}(\mathbf{x})$) at N distinct points with $x_i = 0$ and count the number of occurrences M where $f(\mathbf{x}) = f(\mathbf{x} + \mathbf{b})$ (or, respectively, $\hat{f}(\mathbf{x}) = \hat{f}(\mathbf{x} + \mathbf{b})$). The statistic is again

$$\chi^2 = \frac{1}{N} (2M - N)^2$$

and the confidence level P is computed in the same fashion as with d-monomial test.

This simple test is useful for measuring the basic mixing properties of the function and was therefore employed in our tests of eSTREAM proposals as discussed in the following section.

6 Chosen-IV Tests on eSTREAM Proposals

As there were as many as 34 proposals for eSTREAM [3], some with poor documentation, we decided to make certain assumptions about their structure in order to facilitate "automatic" d-Monomial and bit-flipping testing.

1. We wish to find a subset of input bits that is likely to receive less mixing during the IV setup process than other bits. This is likely to be either at the beginning or the end of the IV bit-vector.
2. After the bits for a d-Monomial test have been chosen, the remaining constant IV bits also greatly affect the probability that the keystream will exhibit bias. We chose to run the tests with these bits set as 0 and also when they are set to 1.
3. Rather than running the test on some low-degree limit d (In [2] $d \leq 3$ and $d \leq 5$ are mentioned), we limit the number of bits n to some manageable number and compute all d-Monomial tests on those bits.

There are four d-Monomial tests in total; {bits in beginning, bits in the end} $\times$ {rest of bits set to 0, rest of bits set to 1}. In practice the black box function (IV setup) was run with increasing values of n until a time or memory limit was exceeded. An ANF was then computed and monomials of various degrees counted. The same data was also subjected to bit-flipping tests as described in Section 5.

The testing code was integrated into the "eSTREAM speed testing framework", which allowed the test to be easily run on most eSTREAM ciphers. The test code simply utilizes the eSTREAM API and treats each cipher as a black box function.

There appears to be bugs in some cipher implementations, that resulted in exceedingly high biases. Those cases are ignored in the discussion below. We only mention ciphers where definitive evidence of statistical anomaly was detected (positive results

are not reported). All tests were run at least 10 times with randomized keys. We only report anomalies that reoccurred in a consistent pattern in distinct tests. Note that when the same tests were run on reference ciphers such as AES-CTR, no anomalies were found.

All specifications of the ciphers are available from the eSTREAM web site [3]. The following list of results is not exhaustive, but just relates to the current status of the tests.

6.1 MAG, Frogbit, and F-FCSR

MAG is a stream cipher designed by Rade Vuckovac that uses a 128-bit key and a 32-bit IV. Frogbit is a "cipher, data integrity algorithm" designed by Thierry Moreau with 128-bit key and IV values. F-FCSR is a family of stream ciphers designed by Thierry Berger, François Arnault and Cédric Lauradoux.

These ciphers exhibited extreme biases. In some cases flipping a particular bit in IV did not affect the first keystream bits at all. The designers of these ciphers appear to have failed to consider the implications of chosen-IV attacks.

6.2 DECIM

Decim is a stream cipher with a 80-bit key and a 64-bit IV designed by Come Berbain et al. Decim is highly vulnerable to d-Monomial distinguishers. Biases that occur with $P < 2^{-96}$ (our implementation precision limit) were consistently found. Decim also appears to be susceptible to a bit-flipping attack, although to a lesser degree. In a typical run of 2^{18} IV setups, a bit-flipping bias with $P < 2^{-16}$ could be found.

6.3 ZK-Crypt

ZK-Crypt is a stream cipher designed by Carmi Gressel, Ran Granot and Gabi Vago. With a 128-bit key and a 128-bit IV it is highly vulnerable to both bit-flipping and d-Monomial distinguishers. Biases with $P < 2^{-96}$ were consistently found in bit-flipping attacks. In d-Monomial attacks the bias was in $P < 2^{-12}$ range, although in one case $P < 2^{-37}$ was observed. A typical test run would involve 2^{21} IV setups.

6.4 POMARANCH

POMARANCH is a stream cipher designed by Cees Jansen and Alexander Kolosha. With a 128-bit key and a 112-bit IV it is susceptible to bit-flipping tests when the flipping occurs at the end of the IV vector. Biases with $P < 2^{-96}$ were consistently observed in such attacks. Typical run would involve 2^{17} IV setups.

6.5 NLS and TSC-3

NLS is a stream cipher designed by Gregory Rose, Philip Hawkes, Michael Paddon and Miriam Wiggers de Vries. TSC-3 is a stream cipher proposed by Jin Hong, Dong Hoon Lee, Yongjin Yeom, Daewan Han and Seongtaek Chee.

These ciphers fall into "borderline category". Some strong biases were found, but not strong enough to indicate a clear design flaw. We suspect that improved attacks are possible by hand-crafting the test parameters to exploit particular features of the design of these ciphers.

In NLS with a 128-bit key and a 128-bit IV, a bias with $P < 2^{-20}$ was observed in one d-Monomial test run of 2^{24} IV setups. Multiple lesser d-Monomial biases occur in a consistent pattern.

In TSC-3 with a 160-bit key and a 128-bit IV, a bit flipping bias with $P < 2^{-18}$ was observed and lesser biases occur in a consistent pattern.

7 Conclusion

We have discussed the application of Algebraic Normal Form and d-Monomial tests to chosen-IV attacks against stream ciphers. It has been demonstrated that these tests appear to be highly effective in distinguishing "simple" Boolean functions as well as (rather surprisingly) complex functions from random ones.

In an experiment with eSTREAM stream ciphers, we found that the output of six of the 34 candidates could be distinguished from random with our methods, with additional few being borderline cases and requiring further investigation. Ciphers with poor mixing properties even fail a simple bit-flipping test (or its ANF variant).

8 Acknowledgments

The author wishes to thank Keith Martin for his valuable comments. This research was supported by a grant from *Helsingin Sanomain 100-Vuotissäätiö*.

References

1. Clote, P., Kranakis, E.: Boolean Functions and Computation Models. Springer-Verlag, 2002
2. Filiol, E.: A New Statistical Testing for Symmetric Ciphers and Hash Functions. Proc. ICICS 2002, LNCS 2513, Springer-Verlag 2002. pp. 342 – 353.
3. ECRYPT: The home page eSTREAM, the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`
4. Murphy, S.: The Power of NIST's Statistical Testing of AES Candidates. AES Comment to NIST, April 2000.
5. Rukhin, A. et al.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 (revised May 15, 2001)
6. Wegener, I.: The complexity of Boolean functions. Wiley-Teubner series in computer science. Wiley, Teubner, 1987

Testing Framework for eSTREAM Profile II Candidates*

L. Batina[1], S. Kumar[2], J. Lano[1], K. Lemke[2], N. Mentens[1], C. Paar[2], B. Preneel[1], K. Sakiyama[1] and I. Verbauwhede[1]

[1] Katholieke Universiteit Leuven, ESAT/COSIC, B-3001 Leuven, Belgium
[2] Horst Görtz Institute for IT Security Ruhr University Bochum, 44780 Bochum, Germany

Abstract. The aim of eSTREAM Profile II is to identify a small number of stream ciphers that are suitable for low resource circuitry based implementation. Besides algorithmic properties and security evaluation to theoretical attacks, performance evaluation is another important task of eSTREAM that is being considered. In this contribution we summarize and explain our testing framework for eSTREAM Profile II candidates regarding hardware implementations.

Keywords: stream ciphers, hardware implementations, implementation attacks

1 Introduction

The main motivation of the eSTREAM project is to identify stream ciphers that can be used as replacements for AES in both high throughput software based implementations (Profile I) and low resource hardware (circuitry) based implementations (Profile II).

Whereas the approach undertaken for performance testing of Profile I candidates is well known, detailed test plans for Profile II candidates have not been presented, yet. Our contribution encourages an open approach for this framework. This work is produced by the VAMPIRE lab as part of the ECRYPT project.

2 Performance Criteria for Profile II Candidates

* The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT, the European Network of Excellence in Cryptology. KUL researchers are also supported by FWO projects (G.0141.03, G.0450.04), GOA Mefisto 2000/06, GOA Ambiorix 2005/11.

The primary aim of eSTREAM Profile II is to find stream ciphers that require lower resources that an AES implementation in circuitry yielding at least the same throughput as an AES implementation. For evaluating the performance of Profile II candidates we consider the categories

1. Compactness (Area),
2. Performance (Throughput),
3. Power Consumption,
4. Flexibility/Scalability/Pipelining and
5. Simplicity/Completeness/Clarity

Each test category is explained in more detail below.

Our main approach is to consider the possible trade-offs between these categories. Among them, compactness and performance are the most important ones and a trade-off metric for compactness and performance is preferable. We mention also a firm requirement for a low power consumption, which is of crucial importance for wireless applications such as PDAs, mobile phones, RFIDs *etc.*

We especially compare with current AES implementation benchmarks (see Section 3). Candidates which are not able to outperform AES implementations in terms of compactness and performance can probably not be advanced further in the eSTREAM Profile II project. Secondary, we compare among eSTREAM candidates. An open question is whether the value of versatile algorithms that are proposed for both Profile I and Profile II is considered differently than pure Profile II submissions.

Note that the most important criterium for analysis of eSTREAM, *i.e.*, mathematical security of the algorithm, is not evaluated as part of this framework.

2.1 Compactness (Area)

For the hardware oriented stream ciphers, the silicon area determines the cost of the implementation. This feature is one of the first to be taken into consideration, because the main goal of stream ciphers is to be smaller than block ciphers. That is why the area of the proposed stream ciphers should be compared to the area of a compact AES implementation. The benchmarks that can be used for this comparison are described in Sect. 3.

2.2 Performance (Throughput)

The properties that are taken into account when evaluating the performance of the stream cipher implementation are frequency, bits per second (throughput) and bits per cycle. Performance is, together with area, one of the most important design criteria. In Sect. 3 performance benchmarks are given for area constrained AES implementations.

2.3 Power Consumption

As stream ciphers are used in small handheld devices, power consumption should be taken into account to estimate the battery's capabilities. However, estimating the power consumption of a design is not straightforward. Power estimation tools such as SPICE can help for this matter, but are not always reliable especially without back-annotating physical layout information.

2.4 Flexibility/Scalability/Pipelining

The flexibility of a stream cipher is determined by the variety of possible implementation options. A high flexibility usually results in a large design parameter space with area and performance as the two main dimensions: implementations can be optimized for speed or for area or the design criterium can be a trade-off of these two. By scalability we mean the ability to scale the design with respect to the width of the data path. This results again in a trade-off between area and speed. Inserting registers for pipelining allows to increase the frequency and the throughput of the implementation.

These criteria do not only consider the inherent flexibility/scalability/pipelining of the design stressed by the author, but also possibilities to realize these properties detected by the implementer.

2.5 Simplicity/Completeness/Clarity

Because the new stream cipher standard will be adopted in many applications, the description should be clear. More specific, all details needed for the implementation should be given in the describing document. To decrease the non-recurring engineering time, a simple description is preferred. Some stream ciphers are more simple by nature and therefore allow a more simple description. However, even the more complicated stream ciphers should be introduced in an illustrative manner. That is why the new stream ciphers should be evaluated on simplicity, completeness and clarity of the describing document.

3 AES Hardware Implementation

The Advanced Encryption Standard (AES) [11] was standardized by the National Institute of Standards and Technologies (NIST) in 2001. AES is a block cipher that operates on 128-bit blocks of data using a 128-bit, 192-bit or 256-bit key. The most common key size is 128-bit and is solely considered in this testing framework. For a complete specification of AES we refer to [11].

A recent report with a strong focus on AES hardware architectures can be found in [5]. For the purpose of this testing framework, the lightweight implementations of [5] are the most important ones.

Most of the previous work on compact AES implementations outlines benchmarks for either ASIC or FPGA implementations. Here, we aim to give both

benchmarks for ASIC and FPGA implementations as FPGAs have attracted more attention in the last years. Therefore, we selected two reference implementations for both ASIC and FPGA implementations.

For ASIC implementations, the reference implementations are from Feldhofer *et al.* [6] and Satoh *et al.* [14]. The former uses an 8-bit architecture and is currently the most compact AES ASIC implementation. On the other hand the work of Satoh *et al.* [14] gives results for different architectures ranging from 32-bit to 128-bit and therefore yields an increased throughput of data. In Table 1 we give the circuit benchmarks based on compactness. Both implementations use combinatorial logic for the S-Box implementation which is more suited for low-cost implementations than the use of a ROM table. There is also the work of Canright [2] that evaluates all options for basis, irreducible polynomial *etc.* to make the S-Box implementation even more compact in order to obtain further optimizations.

For low-cost FPGA benchmarks we select Good/Benaissa [7] and Chodowiec/Gaj [4] as references. The former is based on an 8-bit architecture, whereas [4] uses a 32-bit architecture. Benchmarks are summarized in Table 2.

	Feldhofer [6]	Satoh [14]	Satoh [14]	Satoh [14]
Architecture	8-bit	32-bit	64-bit	128-bit
No. S-boxes	1	4	8	20
Area [GEs]	3,400	5,398	7,998	12,454
Cycles per encryption [1]	1,032	54	32	11
Throughput [bits/cycle]	0.12	2.37	4.00	11.64
Technology [μm]	0.35	0.11	0.11	0.11
Clock frequency [MHz]	80	131	137	145
Throughput [Mbps]	9.9	311	548	1,691

Table 1. Benchmarks for AES-128 low-cost ASIC Implementations

	Good/Benaissa [7]	Chodowiec/Gaj [4]
Architecture	8-bit	32-bit
No. S-Boxes	1	4
FPGA	Xilinx Spartan-II XC2S15-6	Xilinx Spartan II XC2S30-6
Slices	124	222
No. of Block RAMs	2	3
Bits of Block RAM used	4,480	9,600 [7]
Total Equiv. Slices	264	522 [7]
Clock frequency [MHz]	67	60
Throughput [Mbps] [2]	2.2	69

Table 2. Benchmarks for AES-128 low-cost FPGA Implementations

[1] [6] includes the key schedule. For [14], add ten cycles for the key schedule.
[2] For comparison we use the definition of average throughput given by [7].

4 Performance Evaluation

The hardware performance measurements will be similar to Round 2 of AES where different AES candidates were implemented by NSA in an unbiased way. The design analysis consists of hardware designing (mostly based on the stream cipher designers' suggestions), coding in a hardware modeling language, simulation and synthesis for various hardware platforms. We would be concentrating on the low cost FPGAs and semi-custom ASIC with standard CMOS libraries. For a fair analysis, we provide an equivalent treatment for all the ciphers with basic optimizations that would be done during the normal hardware design phase. This would provide a meaningful comparison between the results of various designs and may be suitable only for this specific context of hardware performance measurement.

In Section 2, we mentioned the various performance parameters that will be considered. Since all performance parameters cannot be met in a single design, we would have to find possible trade-offs and possibly implement multiple designs. The flexibility of the algorithm would be the deciding factor for multiple designs. But compiler design constraint settings like delay and area are also another way to find various trade-off points. Our main approach will be to find designs that have low area and medium speed. An iterative kind of algorithm would be the standard choice for the designs.

We would be measuring the key-setup time, iv-setup time and the throughput performance of each of the designs. Our designs will be compared with efficient low-area implementations of AES mentioned in Section 3. Our aim would be to find designs that would be more compact than a low-area AES design but still faster in performance.

The different designs will be modeled using VHDL (VHSIC Hardware Description Language). The designs will be implemented following the standard methodology used by ASIC designers. This would include identifying various sub-blocks from the algorithm that would help to implement a small area iterative design. During this phase, a major deciding factor would be the algorithmic designer's suggestions mentioned in the specifications submitted to eSTREAM. A different approach would be taken only if the hardware designer feels a huge gain in performance than the one suggested. This will be followed by simulation and synthesis of the design model under different area/delay constraints to obtain the various performance measurements. The final physical layout and fabrication for ASIC designs would be beyond the scope of this testing.

For the unbiased approach we neglect the overhead for interfacing to the outside world by providing a standardized interfacing within each of the implementations. Though any input parameter needs that are constraining to a good hardware design would be noted in the final results. This user interface provides the algorithm with the key, initialization vector and the plaintext. It receives the key stream from the algorithm and XORs it to the plaintext, providing the ciphertext to the outside world. All other control signaling to the algorithm are also done from a common control block.

5 Evaluation of Other Implementation Properties

Besides performance criteria, we aim to evaluate also other implementation properties of stream ciphers in Phase II.

This task consists of the test categories

1. Design Analysis,
2. Side Channel Susceptibility,
3. Fault Analysis Susceptibility and
4. Probing Susceptibility.

The task "Design Analysis" deals with possible improvements and guidance for the final specification of the algorithms. The remaining three tasks evaluate the susceptibility of the implementations of eStream candidates towards implementation attacks. Counteracting implementation attacks typically requires additional implementation costs which are not considered in Section 2, yet.

Each task is explained in more detail below.

5.1 Design Analysis

The other main objective of the design analysis would be to find hardware efficient sub-blocks in the various algorithm. This will provide an easily identifiable list of functions that are good for hardware design and hence enable cryptographers to design a more hardware efficient stream cipher in the future.

5.2 Side Channel Susceptibility

Here we discuss vulnerabilities of hardware implementations of stream ciphers to side-channel attacks. It is very important to consider these already in the design phase as from the previous work some general recommendations for the design and countermeasures are known.

Implementation attacks in general exploit weaknesses in specific implementations of a cryptographic algorithm. Sensitive information, such as secret keys or a plaintext can be obtained by observing some side-channel information such as the power consumption, the electromagnetic radiation, *etc.*

In the 90's Kocher *et al.* performed successful attacks by measuring the power consumption while the cryptographic circuit is executing the implemented algorithm [9]. The most straightforward power analysis, called Simple Power Analysis (SPA), uses a single measurement to reveal the secret key by searching for patterns in the power trace. However, implementations that are resistant against SPA attacks, can still be broken by using a more advanced technique, namely Differential Power Analysis (DPA). In this case many power measurements are evaluated using statistical analysis. A similar terminology is used when the observed side-channel is electromagnetic radiation. In that case typical attacks are SEMA and DEMA.

Template attacks were invented by Chari *et al.* [3] and it was shown by Rechberger [12] that they can be also a serious threat to stream ciphers as well as all other ciphers.

From the power and electromagnetic analysis point of view there is not much previous work done on stream ciphers. However, the work of Lano *et al.* considers a DPA attack on synchronous stream ciphers with resynchronization mechanism [10]. Hence, their conclusion should be verified for the candidates in this class of stream ciphers. Also the work of Rechberger and Oswald [13] gives some recommendation for stream ciphers in order to avoid simple side-channel attacks.

5.3 Fault Analysis Susceptibility

Fault analysis is an active implementation attack that aims to disturb the computation of a cryptographic algorithm in such a way that an erroneous result is obtained. By applying mathematical cryptanalysis these erroneous results can be used to extract cryptographic key material. Reference [8] provides several general attacks that are applicable at LFSR based stream ciphers. For RC4, two different approaches have been presented in [1].

In this task, it is evaluated whether an eSTREAM candidate is vulnerable against one of the general techniques of [8]. If so, the complexity of a successful attack is estimated. Additionally, alternative approaches of fault analysis are checked.

5.4 Probing Susceptibility

Probing is an active implementation attack that directly connects to the circuit and allows monitoring of internal data flow.

In this task, the susceptibility of the implementation of eSTREAM candidates towards probing attacks is evaluated. Our approach first identifies critical connections within the implementation. The metric used for evaluation is the entropy loss (of the key, respectively, of the current state) at each critical connection as well as the maximum entropy loss by probing a few critical connections simultaneously.

6 Ongoing Test Activities

Due to the number of submissions, current test activities have started first by using the remaining candidates that are not 'broken' yet by mathematical analysis. After moving to Phase II it is assumed that also selected algorithms with a tweaked version are included in Profile II performance testing.

Actually, the submissions tested at the transition to Phase II are summarized in Table 3.

Profile I and II	Profile II
Hermes8	EDON-80
NLS (2A)	MICKEY / MICKEY-128
Phelix (2A)	MOSQUITO
Rabbit	Trivium
Salsa20	VEST (2A)

Table 3. Candidates under test for both Profile I and II candidates and Profile II candidates (in alphabetical order).

7 Conclusion

Currently, test specifications are still in a draft state. We encourage any third-party contributions and assessments!

References

1. Eli Biham, Louis Granboulan, and Phong Nguyen. Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In *The State of the Art of Stream Ciphers, Workshop Record*, pages 147–155. ECRYPT Network of Excellence in Cryptology, 2004.
2. Dan Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *LNCS*. Springer, 2005.
3. S. Chari, J.R. Rao, and P. Rohatgi. Template attacks. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2523 in Lecture Notes in Computer Science, pages 172–186, Redwood Shores, CA, USA, August 13-15 2002. Springer-Verlag.
4. Pawel Chodowiec and Kris Gaj. Very Compact FPGA Implementation of the AES Algorithm. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 2779 of *LNCS*, pages 319–333. Springer, 2003.
5. Martin Feldhofer, Kerstin Lemke, Elisabeth Oswald, François-Xavier Standaert, Thomas Wollinger, and Johannes Wolkerstorfer. State of the Art in Hardware Architectures. Technical report, ECRYPT Network of Excellence in Cryptology, 2005.
6. Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. AES Implementation on a Grain of Sand. *IEE Proceedings on Information Security*, 152:13–20, October 2005.
7. Tim Good and Mohammed Benaissa. AES FPGA from the Fastest to the Smallest. In Josyula R. Rao, editor, *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *LNCS*, pages 427–440. Springer, 2005.
8. Jonathan J. Hoch and Adi Shamir. Fault Analysis of Stream Ciphers. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *LNCS*, pages 240–253. Springer, 2004.

9. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology: Proceedings of CRYPTO'99*, number 1666 in Lecture Notes in Computer Science, pages 388–397. Springer-Verlag, 1999.
10. J. Lano, N. Mentens, B. Preneel, and I. Verbauwhede. Power analysis of synchronous stream ciphers with resynchronization mechanism. In *In ECRYPT Workshop, SASC - The State of the Art of Stream Ciphers*, pages 327–333, 2004.
11. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, 2001.
12. C. Rechberger. Side-channel analysis of stream ciphers. Master's thesis, TU Graz, Austria, 2004.
13. C. Rechberger and E. Oswald. Stream ciphers and side-channel analysis. In *In ECRYPT Workshop, SASC - The State of the Art of Stream Ciphers*, pages 320–326, 2004.
14. Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Colin Boyd, editor, *Advances in Cryptology — Asiacrypt 2001*, volume 2248 of *LNCS*, pages 239–254. Springer, 2001.

Hardware Evaluation of eSTREAM Candidates:

Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt

Frank K. Gürkaynak[1], Peter Luethi[1], Nico Bernold[2], René Blattmann[2],
Victoria Goode[2], Marcel Marghitola[2], Hubert Kaeslin[3],
Norbert Felber[1] and Wolfgang Fichtner[1]

([1]) Integrated Systems Laboratory, ETH Zurich, CH-8092 Zurich
([2]) Dept. of Information Technology and Electrical Engineering, ETH Zurich, CH-8092 Zurich
([3]) Microelectronics Design Center, ETH Zurich, CH-8092 Zurich

Abstract. One important requirement imposed on all eSTREAM stream cipher candidates was to show the potential to be superior to the AES in at least one significant aspect. We present hardware implementation results of eight different eSTREAM Profile-II candidates, all integrated in 0.25 µm 5-Metal CMOS technology. The goal of this work was to provide a fair base for comparision of different hardware crypto algorithms. Additionally, an AES core optimized for stream cipher output has been implemented and is listed as comparative reference.

1 Introduction

The European Network of Excellence for Cryptography (ECRYPT) has started a multi-year effort called eSTREAM to identify new stream ciphers that might become suitable for widespread adoption. A total of 34 algorithms have been submitted to eSTREAM. Nine candidates (ABC, CryptMT, DICING, Dragon, Frogbit, HC-256, Mir-1, Py, and SOSEMANUK) have been specified as pure software implementations, and a further 13 (F-FCSR, Hermes8, LEX, MAG, NLS, Phelix, Polar Bear, POMARANCH, Rabbit, Salsa20, SSS, TRBDK3 YAEA, and Yamb) have been specified to be suited for both software and hardware implementations. The remaining 12 algorithms (Achterbahn, DECIM, Edon80, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, TSC-3, VEST, WG, and ZK-Crypt) were designed with primarily hardware implementations in mind.

The *Integrated Systems Laboratory* (IIS), together with the *Microelectronics Design Center,* provides a series of lectures on VLSI design at the *Department of Information Technology and Electrical Engineering* (D-ITET) of the *ETH Zurich*. As part of this lecture, students are encouraged to work on projects where they design their own ASICs. Successful implementations are then sent to fabrication, and the manufactured chips are finally tested during a later semester. Since a lot of cryptographic algorithms are developed with hardware realizations in mind, they are very well suited for such semester theses. As a consequence, a number of successful projects were realized at our institute over the years [1,2,3].

For the winter semester 2005/2006, four students showed interest in a project targeting the implementation of cryptographic hardware. It was decided to design a subset of eSTREAM candidates and thereby to provide a fair comparison (of at least the implemented set) of candidate algorithms. Since the entire IC design had to be completed within one semester (14 weeks), not all 34 candidate algorithms could be realized with reasonable effort. According to the advice of Elisabeth Oswald, Thomas Johansson and Matt Robshaw [4], the following guidelines were adopted in order to reduce the number of algorithms suitable for integration within this project. Consider only:

1. Algorithms that were specifically intended for hardware realization (eSTREAM Profile-II candidates).
2. Algorithms that were not known to have any negative cryptological or technical issues.
3. Algorithms for which future development is more likely to be expected.

At the start of the project in October 2005, the decision for a subset of stream cipher algorithms had to be made, and eventually seven eSTREAM candidate algorithms were sorted out: Grain[5], MICKEY[6], MOSQUITO[7], SFINKS[8], Trivium[9], VEST[10], ZK-Crypt[11]. By the time when all of these seven algorithms were successfully implemented in hardware, little time was still left. At this stage, it was decided to briefly revise the remaining five Profile-II algorithms (Achterbahn, DECIM, Edon80, TSC-3, WG), and considered to implement additional algorithms if this could be achieved with reasonable effort. As a result of this procedure, Achterbahn[12] was added to the list of implemented algorithms.

To provide a comparative reference for the results, the well-known *Advanced Encryption Standard* (AES) [13] block-cipher was implemented in Output-Feedback (OFB) mode. In this configuration mode, the block-cipher is able to generate a continuous output stream that can be used as a stream-cipher. Since we have significant experience in implementing the AES algorithm at the IIS, we were able to efficiently customize an AES block for stream-cipher implementation. The customized AES core and the eight stream-cipher designs were then integrated in 0.25 µm CMOS technology.

The organization of the paper is as follows: Section 2 describes the methodology used in designing all circuits. The algorithms are briefly described in section 3. A brief discussion about *hardware efficiency* can be found in section 4 and the implementation results are presented in section 5. Finally, the conclusions are drawn in section 6.

2 Methodology

The comparision of hardware implementations of different algorithms is a difficult and challenging task. Most eSTREAM candidate submissions contain information regarding the hardware implementation of the algorithm. While the presented information is certainly valuable, it is difficult to use the data directly to compare different algorithms with each other. The reasons are as follows:

- The implementations may use different design styles, heavily depending on the type of target hardware: FPGA or ASIC. For ASIC design flows, we can usually take advantage of quite a fine-grained logic optimization enabled by dedicated synthesis tools. As a consequence, this allows for deep logic structures, or in other words, more logic functionality can be executed during a single clock cycle. On the opposite, FPGAs contain dedicated macro structures (e.g. logic slices, multipliers), which allow only for coarse-grained optimization. Most often, the interconnect delay significantly adds up to the overall timing of the final placed and routed FPGA design. Moreover, memory resources are in general quite costly on ASICs, while on FPGAs, large memories are rather common. In order to meet high throughput constraints, this leads naturally to different design styles: one relying on fine-grained logic optimization, the other on shallow logic depth and increased memory usage.
- For ASIC implementations, different manufacturing technologies may have been used, while for FPGAs different types of programmable devices may have been chosen. For instance, ASIC designs may differ in process technology or macro cell library (e.g. subset of fixed-size memories vs. RAM compiler), whilst FPGA architectures may employ specialized blocks such as multipliers or DSP slices. Therefore, it may not be obvious how algorithms realized on different hardware technologies can be compared and how they would fare on identical technology.
- The experience of the designer and the project schedule may play an important factor on how well an algorithm is mapped to hardware.

2.1 Design Flow

This project aims at providing a fair comparison between different algorithms, all of them implemented using a standard cell based ASIC design flow. The target technology for the chip integration is UMC 0.25 µm 5-Metal CMOS technology. Four seventh semester master students (Nico Bernold, Rene Blattmann, Victoria Goode, Marcel Marghitola) worked in two groups to implement the algorithms in VHDL. The students were supervised by two research assistants (Frank K. Gürkaynak, Peter Lüthi) with experience in the entire ASIC design flow. The VHDL source code was functionally verified using the *Mentor Graphics ModelSim* simulation environment. The C-code from the eSTREAM

submission package has been used as a golden model for verification. The circuit was synthesized using *Synopsys Design Vision* tools and the resulting netlist was placed and routed using *Cadence Design Systems SoC Encounter* software. The students had 14 weeks to complete the entire design flow in order to meet a strict tape-out deadline. The chips are due back from manufacturing mid-may 2006.

2.2 Interface

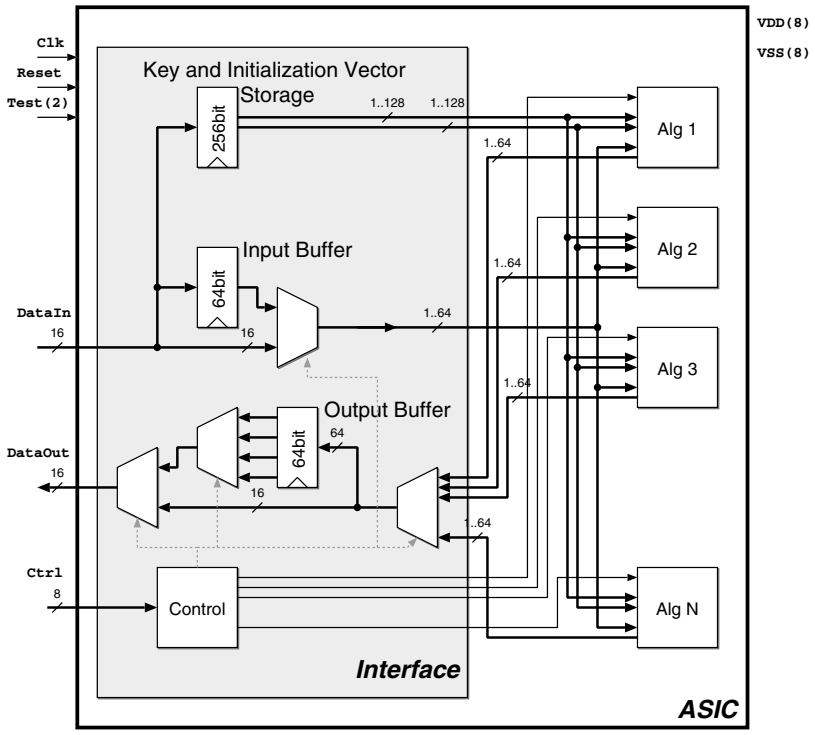


Fig. 1. Simplified block diagram showing the common interface used to access all algorithms.

The available resources for the physical implementation was limited by several constraints. Each group was assigned a die area of 5.92 mm^2. For the technology used, this area is sufficient for an ASIC with 84 pins and a core area of 3.56 mm^2. Since multiple algorithms had to be implemented on a single ASIC, a common interface as shown in figure 1 was developed. Due to a limited amount of I/O pads, the ASIC uses 16-bit input and output buses for data exchange, although some algorithms require more than 16 data bits per clock cycle. To satisfy the requirement for delivering more data, 64-bit buffers for both input and output have been added. Algorithms that require more than 16 bits of I/O data per clock cycle can be run using one of two options.

1. In the slow mode, the algorithm is halted until the input buffer has collected sufficient data. After accumulation of all data, the algorithm is run and the output is again collected at the output buffer. The algorithm is halted until the data is read out of the buffer. In this mode, all input data is used for en-/decryption.
2. In the fast mode, the algorithm is not paused. Instead, the missing input data is obtained by replication, and only a portion of the output is observed. This mode may be applied for speed testing.

The cipherkey and the initialization vector (IV) are stored in a common 256-bit register. This register is made available to all algorithms in parallel.

To provide an equal basis for comparison, the guidelines listed below were followed:

– Some submissions did not provide an associated authentication method. All algorithms were implemented without any authentication method add-on.
– All synthesized algorithms include scan-test structures for full-scan testing.
– No ROM macros were used for the look-up tables and/or complex functions.
– All algorithms were designed to accept plaintext and deliver ciphertext. Algorithms which only generate key streams were enhanced by adding XOR gates.

2.3 Cryptographic Security

We believe that our expertise resides mainly in the design of digital circuits. The discussion of security aspects of the implemented algorithms is therefore left to experts in cryptography. All algorithms have been assumed to be equally secure for performance comparison.

Once an otherwise secure cryptographic algorithm is implemented in hardware or software, it will acquire physical properties that can be observed. If it is possible to guess parts of the cipherkey by observing these physical properties, the hardware implementation is said to be vulnerable against side-channel attacks. The specific implementation of an algorithm may have a strong influence on how effective a given side-channel attack will be. However, there is no algorithm- and attack-independent methodology to rate the side-channel vulnerability of an implementation. Therefore, no remarks will be made on how vulnerable the implementations are against side-channel attacks. During the design phase, no special countermeasures against side-channel attacks have been considered and implemented. The side-channel security of the implemented algorithms will be determined by measurements on fabricated ASICs in a follow-up project.

2.4 Measuring Performance

The following performance metrics will be used in this paper:

– **Circuit area (A)**
A represents the total area that is required for the implementation, expressed in μm^2. For reference, in the technology used for this implementation: a 2-input NAND gate occupies an area of 23.76 μm^2, a 2-input XOR gate occupies 55.44 μm^2 and a scannable flip-flop with reset occupies 205.92 μm^2. The circuit area is obtained from synthesis results, and does not include buffers for clock distribution and additional overhead for placement and routing.
– **Maximum clock rate (f)**
The maximum clock rate, given in MHz, is determined by the critical path of the circuit. The number is once again taken from post-synthesis timing analysis. In the technology applied here, the fanout-of-four (FO4) delay [14] of a simple inverter is approximately 0.1 ns.
– **Processed bits per clock cycle (*radix*)**
Most of the submitted stream-cipher candidates have been specified with single bit output. For some algorithms, it is possible to modify the architecture in such a way that multiple output bits are calculated concurrently. Moreover, some algorithms like VEST have variations of the architecture for different output bit lengths. The number of bits simultanously generated by the algorithm is referred to as the *radix* of the implementation. Some algorithms will have multiple implementations with different radices.
– **Total throughput (T)**
One of the fundamental parameters of a cryptographic algorithm is the amount of data it can process within a given period. The total throughput of the algorithm is epxressed as Gbits/s and can be calculated from the previous parameters as:

$$T = f \times Radix$$

– **Throughput per unit area (TpA)**
Judging the performance purely by the throughput is not representative as this provides no indication about the area required for the implementation. For this purpose, the *throughput per unit area* measure will be used:

$$TpA = \frac{f \times Radix}{A}$$

3 Algorithms

In this section, specific comments about the eight implemented algorithms are given. These comments target mainly the *process of implementation* of the eight eSTREAM candidates, in other words, how straight-forward the actual implementation process was based on the provided documentation. Note that, for a hardware designer, the reference C-code is just as important as the written documentation. The implementation needs always to be verified against the reference C-code, and when in doubt, always the implementation in the C-code is assumed to be correct.

3.1 Achterbahn

As mentioned earlier, Achterbahn was not amongst the initial candidates for implementation. Once all intended algorithms were implemented, it was decided to briefly revise the remaining 5 algorithms to determine whether or not more could be implemented. Achterbahn was selected primarily because it is very well documented and has an excellent reference C-code, exactly the desired prerequisites for hardware designers.

Achterbahn can be configured to use initialization vectors (IV) of different bit-lengths. This flexibility comes at the expense of a more complex initialization sequence which also requires more hardware. Our implementation is therefore limited to support only a 64-bit IV.

While it is possible to implement higher *radix* versions of Achterbahn, doing so increases the critical path, hence reducing the efficiency of this approach. In principle the algorithm could be implemented employing any *radix* without major difficulties. Due to the initialization sequence, practical *radices* are limited to even dividers of 176.

3.2 Grain

Grain is an algorithm that is rather simple and straightforward to implement for *radices* up to 16. A *radix*-32 implementation is also possible, but would result in a longer critical path. At the start of the project (October 2005), there were two versions of Grain available. From a hardware performance point of view, there is no difference between the two versions. The submission package of Grain included good documentation and good reference C-code.

3.3 MICKEY

MICKEY is another compact algorithm that is very easy to implement. The documentation is written in a *'hardware designer friendly'* way and the reference C-code is also easy to follow. The only technical issue of this algorithm is the difficulty to increase the *radix*.

3.4 MOSQUITO

MOSQUITO is the only algorithm implemented that has separate encryption and decryption modes. There were several problems with the reference C-code. The initial submission was corrected in July 2005. However, this code still had some errors, which were finally corrected in December 2005. The accompanying documentation lacks precision for implementation.

MOSQUITO has a pipelined structure with very few gates in between registers. It is therefore difficult to modify the algorithm for higher *radix* implementations. Without the initialization sequence, a *radix*-9 implementation would theoretically be possible. However, the initialization sequence that uses 104 bits renders this impractical. We have implemented a *radix*-3 version of MOSQUITO.

The fifth-stage register is specified to be 53 bits wide. During synthesis of the algorithm, it was noticed that only 48 bits were used, the content of the 5 most significant bits was discarded.

3.5 SFINKS

SFINKS is mainly dominated by a multiplicative inverse function in GF(2^{16}). This is a relatively complex block that can be implemented by iteratively decomposing the function into operations in GF(2^2). While the documentation contains an appendix explaining this process, especially for hardware designers that are not well versed with Galois field implementations, the required transformation is not trivial. In essence, the inverse function is a 16-bit input, 16-bit output function. However, during normal operation, only 1-bit is used to calculate the key stream (a further bit is used for the calculation of MAC, which was not implemented in this project). The full 16-bit output is only required during initialization. As explained in section 4.1, from a hardware design perspective, the system can be modified in a way where the initial states of the registers are directly loaded. If the algorithm is implemented in this way (we called this implementation SFINKS+), the *throughput per area* can be increased by more than 75%. Additionally, it is also less costly to increase the *radix* of SFINKS+.

The high logic complexity of the inverse function results in a relatively low *maximum clock frequency*. it can be increased by adding pipeline registers into the inverse function. Our implementation uses a single pipeline stage.

The documentation of SFINKS had some errors at the beginning, these were corrected later. Apart from the description of the inverse function, which is difficult to understand, the documentation is easy to follow.

3.6 Trivium

Trivium has a very simple structure that is well-suited for higher-*radix* implementations up to *radix*-64 without noticable hardware penalties. In fact, from a *hardware efficiency* point of view, it is wasteful to implement Trivium with a *radix* less than 64. *Radix*-64 Trivium is just 54% larger, and has a *maximum clock frequency* that is only 10% lower than a *radix*-1 implementation. Consequently, the *throughput per area* of the *radix*-64 version is roughly 40 times higher compared to the *radix*-1 alternative.

The main problem with Trivium is the reference C-code, which does not have any comments. This made it extremely difficult to integrate it into the verification flow.

3.7 VEST

The initial documentation set of VEST was not very easy to follow, and was not very clear regarding the input permutations to the non-linear functions in the accumulator. It was later discovered that the documentation had been updated in the meantime, and we believe that the problems were addressed in this revision. The reference C-code is not well suited for understanding the algorithm, as it is cluttered with pre-processor commands.

VEST has been described in separate families of functions for 4, 16, and 32 bit output per clock cycle, called VEST4, VEST16 and VEST32 respectively. The new documentation also includes the 8-bit version VEST8.

The algorithm is fairly complex and has an equally complex initialization sequence. The majority of the functions are described as look-up tables. Describing the algorithm in VHDL is not a very trivial task. We have modified the reference code so that it generated output that we could include in the VHDL description itself. The large number of look-up tables might be suitable for FPGA implementations, since FPGAs realize functions within small look-up tables. Nevertheless, for a custom ASIC solution the approach of using look-up tables is cumbersome. In fact, using our standard design flow it was only possible to synthesize VEST4 and VEST16. We will have to re-write the code for VEST32 so that we can pass it through the synthesis stage.

3.8 ZK-Crypt

ZK-Crypt has by far the worst documentation of all eSTREAM candidates that we have implemented. First, there is no overview which makes it extremely difficult to follow. A multitude of drawings has been provided, but some drawings are marked as *'conceptual'* and seem to be inconsistent with the documentation. The reference C-code fares better, but is also far from easy to understand. At least in one case there is an inconsistency between the reference C-code and one of the drawings, regarding how key bits 26 and 27 are handled. We have implemented the algorithm in such a way that is consistent with the reference C-code (which simply ignores the content of these two bits). From a hardware designer's point of view, this peculiarity might originate from an incomplete C-code, what would also result in non-exhaustive functional pattern generation for hardware verification once the functionality of these two bits is implemented. We decided to stick to the original C-code and to ignore any ambiguous information for the hardware design.

The algorithm is very difficult to implement, due to its irregular structure and many details, especially in the control state machines. We made no attempt to try different *radix* implementations. On the positive side, the algorithm does not have an initialization sequence and uses no IV.

3.9 Reference AES implementation

To serve as a reference, we have implemented an AES core that is configured to run in *output feedback mode* (OFB). The core accepts 128-bit keys, and uses an on-the-fly roundkey generator. It has 4 parallel look-up tables for the *SubBytes* function, and requires 41 clock cycles to compute a 128 bit output that is used as the key stream (resulting in a calculated *radix* of 3.12). For an independent implementation, the AES core would also have to store the cipherkey so that it can restart generating the roundkeys for each encryption. In this implementation, the cipherkey is stored in the interface which results in a slightly more compact realization (about 10% less circuit area).

4 Efficiency in Hardware

4.1 Initialization

Several eSTREAM candidates require an initialization phase. During the initialization phase, the internal registers are preset to a certain value, and the cipherkey and the initialization vector are loaded into specified registers. Some eSTREAM candidates require a number of operational cycles that will initialize all internal registers prior to generating the key stream.

From a cryptographic point of view, it may be important to differentiate between cipherkey and the initialization vector. However, from a VLSI designers point of view, only the internal registers responsible for key stream generation during the en-/decryption process need to be set to an initial state. This initial state of the registers can be derived mathematically from the cipherkey and the initialization vector and may then be loaded directly into the registers, saving precious setup time before being able to process any data. Moreover, the streamlining of the initialization procedure might also reveal some benefits in terms of hardware complexity: Basically, the control overhead and data switching through multiplexers is reduced, what leads to minor improvements in clock speed and area. But in rare cases, the optimization of the initialization procedure may result in a significantly improved *hardware efficiency*:

As an example, in SFINKS, the initialization routine requires a 16-bit multiplicative inverse which must be delayed by 6 clock cycles. The 16-bit output of the delay buffer implemented as 96 flip-flops (FF) is then fed back to the LFSR. This function is only required for the initialization procedure, during normal operation only the LSB output of the multiplicative inverse is used. If the algorithm is modified so that the initial state is calculated externally and loaded directly onto the hardware as seen in figure 2, the inverse function can be simplified and the delay elements can be substantially reduced as well. In this way, the *throughput per area* of SFINKS can be increased by more than 70%. This major improvement in *hardware efficiency* originates from both, reduction in circuit area and increase in clock speed.

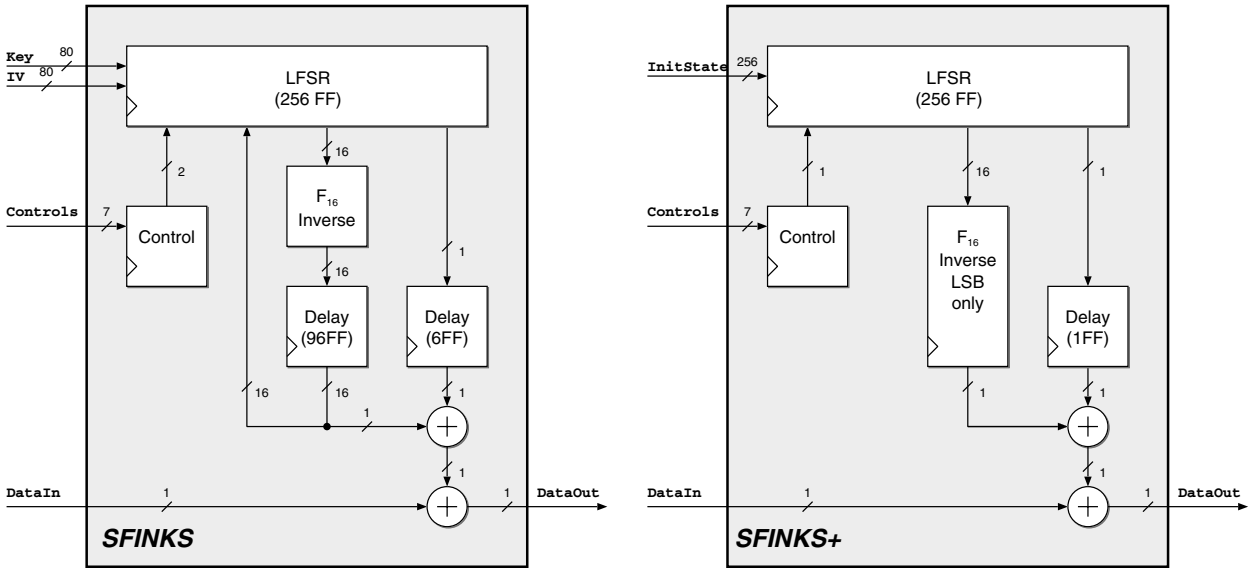


Fig. 2. Native implementation of SFINKS (left), as suggested by the original eSTREAM candidate submission, has significant overhead for initialization. SFINKS+ (right) does not have this overhead and is even more efficient in terms of circuit area, data throughput and initialization latency.

4.2 Stage delay

Synchronous digital circuits for ASICs are in general built from standard cell libraries. The elements in standard cell libraries are classified in logic and sequential cells. Sequential cells, such as flip-flops and latches, serve for storage of data, while logic cells are necessary to reflect the mathematic functions in hardware.

The *maximum clock frequency* of the circuit is determined by the longest path induced by numerous logic cells between two sequential elements in the circuit. Each logic cell (or *gate*) in the critical path will contribute some delay to the signal propagating through. For a simplified analysis, one can assume that all gates have a technology-dependent unit delay. For such analyses the FO4 delay is frequently used [14]. In this simplified analysis, the clock frequency can be expressed in terms of FO4 gate delays. This allows for simple extrapolation of circuit performance in other technologies.

State-of-the-art high performance digital circuits can be designed with as little as 10-20 FO4 delays. However, such designs require utmost precision in the back-end design phase (the *physical* design process: cell placement, routing and clock distribution) and are most often hand crafted. The back-end overhead for 20-50 FO4 delay circuits is still significant. It is a very challenging task to implement these circuits using standard cells. Circuits with roughly 50-100 FO4 delays are fast designs that are manageable with standard cell design methodologies, and implementing circuits with 100-200 FO4 delays is a straightforward procedure. Finally, circuits with more than 200 FO4 delays hardly pose timing related challenges.

For the UMC 0.25 µm technology, the FO4 delay is approximately 0.1 ns. Consequently, designs with up to 200 MHz can be realized without excessive overhead. Circuits that can be clocked faster can still be implemented, but they pose significant challenges to the back-end design process and are rarely practical.

4.3 Bits per clock cycle

There are two fundamental options that can be employed to improve the throughput of a cryptographic circuit. Either the *maximum clock frequency* can be raised or the *radix* of the circuit is increased. As already explained before, the

increase of the frequency is only viable to a certain extend. Nevertheless, a higher *radix* remains as second option and in general turns out to be a very powerful method to boost the overall throughput.

Not all algorithms are equally suited to produce multiple output bits per clock cycle. Some algorithms require replication of major operational blocks in order to increase the *radix*, and thus lead to an enlarged circuit area. Basically, if the *n*-fold increase in the *radix* requires *n*-fold increase in the area, both implementations would have a similar *throughput per area*, and thus would be equally efficient. Moreover, such changes increase often the critical path and decrease the maximum operating frequency as well. However, as can be seen in figure 3, several eSTREAM candidates have been designed with support for multiple-bit computation in mind. The performance of these algorithms can be improved considerably by increasing the *radix*.

Grain is an illustrative example for the typical VLSI challenge of trading in *throughput* against *circuit area*: If more throughput is required, the *radix* might be doubled, but the resulting gain is not two-fold since the *circuit area* slightly grows and/or the the *clock frequency* drops. Trivium is an extraordinary example where doubling the *radix* has almost no impact on the *area*, and the *clock frequency* can be sustained. Therefore, the achieved gain is nearly doubled and almost at the ideal curve. On the other hand, Achterbahn represents an implementation, which is not appropriate for architectural changes in *radix* in order to achieve a higher throughput. The best *hardware efficiency* is obtained with *radix*-2, increasing the *radix* further even degrades the efficiency of Achterbahn. This is because both values are nearly equally affected, the *circuit area* grows and the *clock frequency* degrades. From an efficiency point of view, it is not advisable to implement any other version than *radix*-2. For higher throughput rates rather than using higher-*radix* implementations, replicating several *radix*-2 versions of Achterbahn would be more efficient.

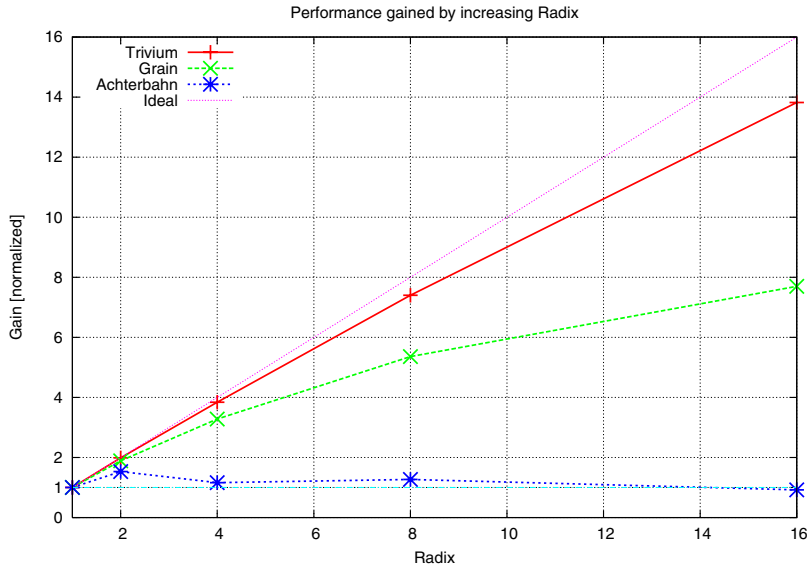


Fig. 3. Performance gained by increasing the area. Performance is expressed in terms of throughput per area, and is normalized to the *radix*-1 implementation of each algorithm.

5 Results

The numbers listed here are synthesis results. Post-layout results, including power figures, will be presented at the SASC 2006 workshop.

As a first step, all algorithms have been implemented to match their description. Apart from VEST and ZK-Crypt, this results in *radix*-1 implementations which have *throughputs* at around 0.3 Gbit/s. When compared to the reference

AES implementation, *radix*-1 algorithms with smaller *area* (Grain, MICKEY, Trivium) achieve a higher *throughput per area* ratio, while algorithms that require more *area* (Achterbahn, MOSQUITO and SFINKS) can not match the performance of AES. Both VEST and ZK-Crypt, which have a higher *radix* by definition, are able to outperform AES implementation noticably.

As a second step, we tried to optimize all algorithms in order to increase their performance. In most cases, significant performance gains can be obtained by increasing the *radix*. Especially Trivium, which has been designed with parallelization in mind, reaches an exceedingly high *throughput*. Table 1 compares the main performance figures for all algorithms. For algorithms that have multiple implementations, only the one with the highest *throughput per area* is listed. A graphical comparison of the results are given in figure 4 as well.

Table 1. Summary of results for eSTREAM candidates. For each algorithm, the most efficient implementation (high *throughput per area*) has been listed.

Algorithm	**A** (μm^2)	**f** (MHz)	**radix** (bits)	**T** (Gbit/s)	**TpA** (Gbit/s·mm^2)	**TpA** (norm)
Achterbahn	227,763	250	2	0.466	2.044	1.08
Grain	119,821	300	16	4.475	37.346	19.79
MICKEY	82,328	308	1	0.287	3.481	1.85
MOSQUITO	306,907	265	3	0.739	2.408	1.27
SFINKS+	361,643	167	8	1.242	3.434	1.82
Trivium	144,128	312	64	18.568	128.833	68.30
VEST	393,000	286	16	4.257	10.833	5.74
ZK-Crypt	142,007	203	32	6.057	42.656	22.61
AES (OFB)	280.098	182	3.12	0.528	1.886	1.00

Several algorithms (Achterbahn, MOSQUITO, SFINKS, VEST), even in their non-optimized forms, require an *area* comparable to AES. For higher-*radix* implementations, only few (Grain, MICKEY, Trivium, ZK-Crypt) are noticably smaller than AES. To achieve the stated performance, most algorithms require a *clock frequency* that is above the comfort zone for a standard-cell-based design (roughly 50 FO4 delays, 200 MHz for UMC 0.25 µm technology). Implementations with faster clock rates are possible, but have considerably more overhead during physical design.

Some algorithms are able to achieve significantly higher *throughput* (Grain, Trivium, VEST, ZK-Crypt) than the reference AES implementation. But the real efficiency comparison is the achieved *throughput per area*. Three algorithms (Grain, Trivium and ZK-Crypt) are at least 20 times more efficient than AES. Out of the remaining algorithms, only VEST is able to clearly distance itself from AES, while the others (Achterbahn, MICKEY, MOSQUITO and SFINKS) are only slightly better.

6 Conclusions

The expectations from an efficient cryptographic algorithm will differ depending on the specific application. Sometimes, small area will be of utmost importance, at other times, a certain data throughput will have to be maintained. It is therefore not practical to expect that a single implementation will satisfy all requirements. Our opinion is that the most important aspect for a hardware-efficient cryptographic algorithm is flexibility. It must be possible to trade-off total throughput with area over a wide range.

From the eight implemented eSTREAM candidates, there are several algorithms that can achieve significantly higher throughput per area ratings, and several others which are noticably smaller in area than the reference AES imple-

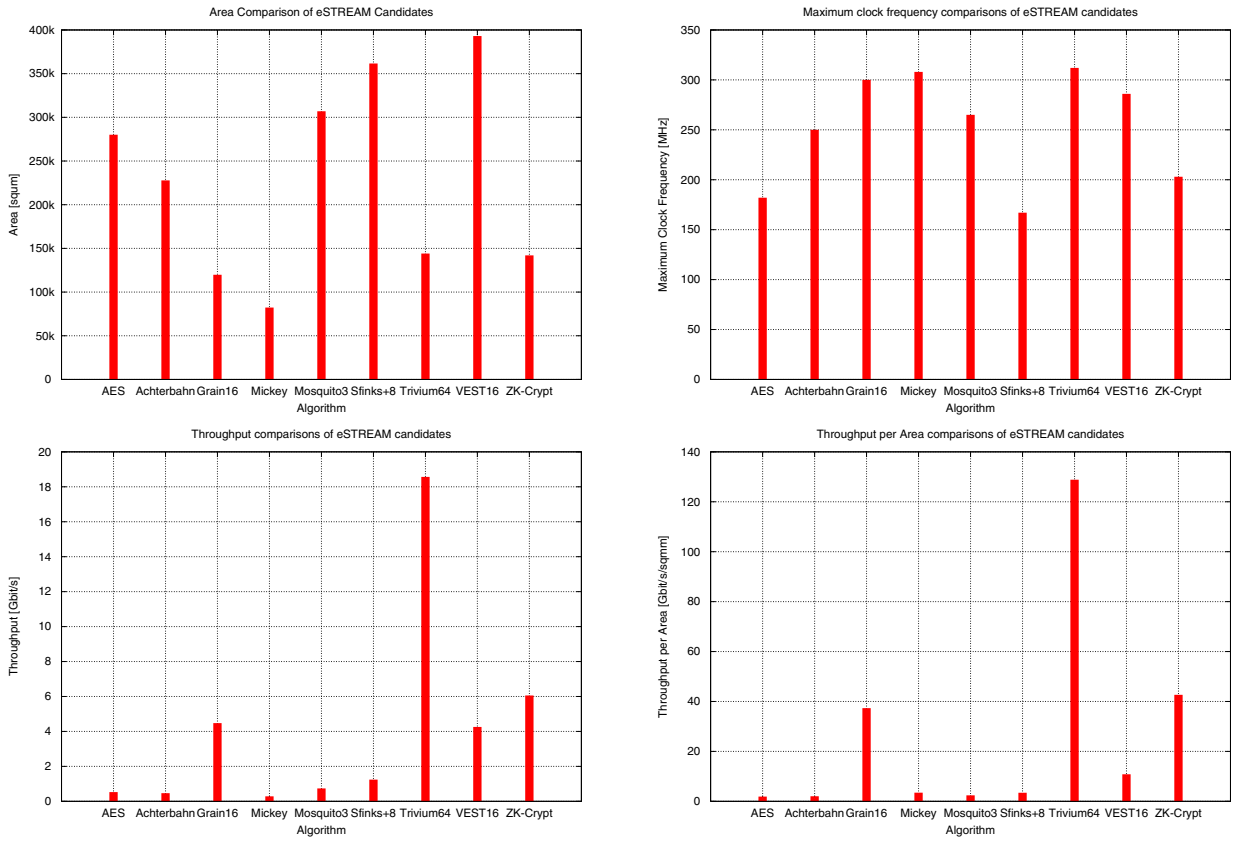


Fig. 4. Synthesis results for eSTREAM candidate algorithms, compared to an efficient AES implementation.

mentation. However, we belive that it is not possible to rate the presented algorithms without knowing their relative cryptographic qualities.

References

1. T. Villiger, J. Muttersbach, H. Kaeslin, N. Felber, and W. Fichtner, "A Globally-Asynchronous Locally-Synchronous VLSI Circuit for the SAFER Cryptoalgorithm," in *Handouts of the First ACiD-WG Workshop of the European Commission's Fifth Framework Programme, Neuchatel, Switzerland*, Feb. 2001, pp. 249–256.
2. A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner, "2 Gb/s Hardware Realizations of RIJNDAEL and SERPENT: A Comparative Analysis," in *Proc. Cryptographic Hardware and Embedded Systems - CHES 2002, LNCS 2523*, Aug. 2002, pp. 144–158, Springer-Verlag.
3. F. K. Gürkaynak, A. Burg, D. Gasser, F. Hug, N. Felber, H. Kaeslin, and W. Fichtner, "A 2Gb/s Balanced AES Crypto-Chip Implementation," in *Proc. of the Great Lakes Symposium on VLSI*, Apr. 2004, pp. 39–44, ACM Press.
4. E. Oswald, T. Johansson, and M. Robshaw, "Criteria to select estream candidates for implementation as iis student projects." personal communication, 2005.
5. M. Hell, T. Johansson, and W. Meier, "Grain - a stream cipher for constrained environments." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005. http://www.ecrypt.eu.org/stream.
6. S. Babbage and M. Dodd, "The stream cipher MICKEY." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/015, 2005. http://www.ecrypt.eu.org/stream.
7. J. Daemen and P. Kitsos, "The self-synchronizing stream cipher mosquito." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/018, 2005. http://www.ecrypt.eu.org/stream.
8. A. Braeken, J. Lano, N. Mentens, B. Preneel, and I. Verbauwhede, "SFINKS : A synchronous stream cipher for restricted hardware environments." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/026, 2005. http://www.ecrypt.eu.org/stream.
9. C. D. Cannière and B. Preneel, "Trivium - a stream cipher construction inspired by block cipher design principles." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. http://www.ecrypt.eu.org/stream.
10. S. O'Neil, B. Gittins, and H. Landman, "VEST - hardware-dedicated stream ciphers." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/032, 2005. http://www.ecrypt.eu.org/stream.
11. C. Gressel, R. Granot, and G. Vago, "ZK-crypt." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/035, 2005. http://www.ecrypt.eu.org/stream.
12. B. Gammel, R. Göttfert, and O. Kniffler, "The achterbahn stream cipher." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/002, 2005. http://www.ecrypt.eu.org/stream.
13. National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)," *FIPS Publication*, vol. 197, 2001.
14. R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.

Review of stream cipher candidates from a low resource hardware perspective

T. Good, W. Chelton and M. Benaissa

Department of Electrical & Electronic Engineering,
University of Sheffield, Mappin Street, Sheffield, S1 3JD, UK
{t.good, m.benaissa} @ sheffield.ac.uk

Abstract. This paper presents hardware implementation and analysis of a carefully selected sub-set of the candidate stream ciphers submitted to the European Union eStream project. Only the submissions without licensing restrictions have been considered. The sub-set of six was defined based on memory requirements versus the Advanced Encryption Standard and any published security analysis. A number of complete low resource designs for each of the candidates are presented together with FPGA results for both Xilinx Spartan II and Altera Cyclone FPGAs, ASIC results in terms of throughput, area and power are also included. The results are presented in tabular and graphical format. The graphs are further annotated with different cost functions in terms of throughput and area to simplify the identification of the lowest resource designs. Based on these results, the short-listed six ciphers are classified.

Keywords. Stream Ciphers, Hardware, FPGA, ASIC, Performance Evaluation.

1 Introduction

In 2004, a project under the Information Societies Technology (IST) Programme of the European Commission "eCrypt" network of excellence called "eStream" was started tasked with seeking a strong stream cipher. Thirty-four candidate ciphers have been submitted and are currently being evaluated in terms of security.

A stream cipher formally is a symmetric cipher which generates a sequence of cryptographically secure bits called the key stream which is then combined with either the plaintext or ciphertext, at the bit level, using the exclusive-or operation. The basic topology (Fig. 1) of a stream cipher consists of a register to store the key and an initialisation vector (IV) together with a function for its update (typically some sort of feedback shift register). This register forms the current state of the cipher and is clocked for successive bits of the keystream. The next component is a non-linear reduction function which takes part or all of this state and combines the bits in a non-linear fashion normally to yield a single bit of the keystream. This bit is then exclusive-or'ed with the plain/cipher text. In a second form, the plain or cipher text may be incorporated into the state update feedback function to effectively create a cipher-feedback mode.

A vital function, in terms of security, is the period of the initial key and IV mixing to prevent key recovery attacks. In this period, a cryptographically strong feedback function is needed to operate upon the state for a number of iterations (basically hashing). The reduction function used to output the keystream can be somewhat weaker.

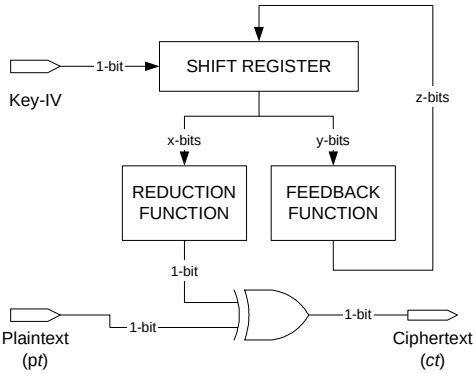


Fig. 1. Generic stream cipher

The call for cipher primitives[1] made provision for two profiles, one for software, requiring equivalent security of 2^{128}, and one for hardware, requiring 80-bit (2^{80}) security. An extension to the basic cipher was also defined for those wishing to supply a message authentication code (MAC). The call recognised the importance of resource utilisation for both profiles in that the deployed environments for stream ciphers often have very restricted resources (eg smart cards). To aid this, the call defined the Advanced Encryption Standard (AES) as a benchmark and submissions should use less resource and be "faster" then the AES.

There has been very little discussion or comparison of the hardware implementations of the candidate stream ciphers to date. The majority of the effort has been correctly directed towards the cryptanalysis of the algorithms. However, as will be shown in this paper, even early hardware results can provide a timely method for selecting a sub-set of the candidates for more intensive scrutiny. It is hoped that this paper will allow effort to be directed towards the "low resource" hardware submissions so that these are proved secure or broken more or less in the order of hardware "performance".

Section 2 of this paper describes how the list of ciphers was sifted to locate a smaller set for hardware evaluation. Briefly, this was in terms of the ciphers commercial status (i.e. "free-for-all"), the amount of internal state and identification of any large look-up tables (S-boxes). This is followed, in section 3, by details of the method used to evaluate the hardware performance and in section 4, the results for the selected ciphers. Where the hardware results were affected by the developers' choice of initialisation, this is highlighted, as tweaks to initialisation are permitted within the scope of the eStream call. Finally, in section 5 some conclusions are drawn. Appendix A (due to page number restrictions) gives details of each of the designs together with additional suggestions on possible 'tweaks' aimed at further reducing the hardware requirements.

Considering the results for Xilinx FPGA, Altera FPGA and power results for 0.13µm Standard Cell ASIC allows some strong conclusions to be drawn. Of the ciphers considered, this analysis excluded the "commercial" ones, Grain and Trivium can be ranked as the most efficient followed by Sfinks, Mosquito and Hermes. The raw results have been included to permit others to choose their own metrics and allow for further comparisons.

2 Selection Process

Any selection process will be inevitably coloured by the authors own position and beliefs. In the interest of academic fairness we will state ours:

1. We have no affiliation or predisposition to any of the candidate algorithms or their authors.
2. We would like to see the successful stream cipher be "free for all" to use. Thus we have not directed any efforts towards any of the "non-free for all" candidates.
3. We are concerned only with low resource hardware results and believe that both FPGA and ASIC results are important.
4. We do not wish to make any security claims about any of the candidates and where ciphers have been disregarded from this analysis on the grounds of security weakness we have relied on our interpretation of the results posted on the eStream web site.

There are some 34 candidate primitives submitted. From the information provided on the eStream web site [1], nine of the candidates were subject to some form of licensing or restriction so excluded from our analysis. For a further seven of the candidates the published cryptanalysis had highlighted, in the authors' view sufficient weakness for it to be excluded from this analysis. To reiterate, this is the authors' view for the purpose of reducing the number of candidates to implement in hardware and is not in any way concerned with the formal selection process by the eStream project.

The developers submitted their algorithms to either the software, hardware or both profiles. From initial examination of a few of the "software" submissions it was recognised that although these had not been submitted to both the software and hardware profiles they may have a low resource hardware implementation so should be considered.

For the eighteen remaining candidates, the reference designs and papers were examined in detail to determine any hardware results reported by the developers together with the amount of internal storage (in bits) and any memory requirement for "S-box" substitution operators. It was further noted, if any S-box had known or likely logic implementation which could be utilised to avoid a relatively large memory. In the case where the "S-box" is generated using the key or otherwise manipulated making implementation as a ROM not possible it was considered as part of the internal state.

A view was taken on what would be acceptable as low resource with the aim of approximately reducing the remaining candidates to produce our "top six". As a baseline for comparison we considered an earlier low resource FPGA implementation of the AES [2] which supported three feedback modes (OFB, CTR and CFB)

thus represents a well understood and relatively secure stream cipher. For a second baseline, the standard cell ASIC design of Feldhofer [3] was selected.

FPGA results can be obtained more rapidly than ASIC results so the evaluation was started with consideration of the FPGA performance. The FPGA AES baseline case was viewed as the limiting case that candidates must outperform. This implementation [2] required 704 bits of internal state and a 2kbit S-box (implemented using composite field logic). This design supported three feedback modes, if a single mode were selected such a design would only require approximately 400 bits of storage. Consequently, baseline limits of 400 bits of internal state and 2kbit of fixed-valued S-box were selected.

This selection process may at first glance appear relatively crude, however, in hardware, the area occupied by a D-type Flip Flop, which is the most likely means of storage of internal state bits, is relatively large compared with combinational gates, thus, will account for a significant proportion (>50%) of the area of any low-resource implementation. A similar argument can be made for the area consumed by requiring a few kilo-bits of memory (either RAM or ROM).

Table 1 lists all the candidates in alphabetical order together with the authors' reasons for selection or non selection for further analysis.

Table 1. Summary of selection of candidates

Cipher	Profile	Free for all	Internal state (bits)	Key & IV bits	S-box bits	Cut	Notes
ABC	1	yes	160 +KE (1024)	128 128	0	×	For software broken but still >2^80 so ok for hardware, however, Key Expansion of 32x32-bit words (1024 bits) and not sure from paper or code what is the "standard key expansion".
Achterbahn	2	yes	-	-	-	×	broken, linear ca in 2^73
CryptMT/ Fubuki	1	no	-	-	-	×	not free for all
Decim	2	no	-	-	-	×	broken, 2^29 IV to recover key
Dicing	1	yes	768	128/256 128/256	2k	×	disputed ca, large internal state
Dragon	1	yes	192	128/256 128/256	16k	×	large "randomly" generated s-boxes, disputed ca
Edon80	2	no	-	-	-	×	key period doubts, not free for all
F-FCSR	1&2	yes	-	-	-	×	broken, key recovery attack
Frogbit	1A	no	-	-	-	×	not free for all
Grain	2	yes	160	80 63	0	✓	ok, linear ca which required 2^61.4 bits of keystream
HC-256	1	yes	64k	256 256		×	2 x huge s-boxes (64 kbit) 1024 bit subtraction
Hermes8	1&2	yes	224	80 184	2k or logic	✓	ok, uses AES s-box
LEX	1&2	no	-	-	-	×	not free for all, key recovery in 2^61 IVs
MAG	1&2	yes	-	-	-	×	broken, low complexity distinguishing attack
MICKEY	2	yes	-	-	-	×	key stream entropy loss
Mir-1	1	yes	2432	128 64	2k or logic	×	too much internal state, uses AES s-box with key to generate own s-box

Table 1 continued…

Cipher	Profile	Free for all	Internal state (bits)	Key & IV bits	S-box bits	Cut	Notes
Mosquito	1A& 2A	yes	128	96 104	0	✓	ok
NLS	1A& 2A	yes	1184 ?	256 256	8k	×	two S-boxes (total 8x32 bit s-box). too much internal state & s-box!
Phelix	1A& 2A	yes	352	256 128	0	✓	ok
Polar Bear	1&2	yes	168	128 <248	2k or logic	×	5 round AES + RC4, one round of AES is still relatively large
Pomaranch	1&2	yes	184 ?	128 144	4k or logic	?	Unsure of status of broken then fixed submissions
Py (“Roo”)	1	yes	10400	256 128	0	×	too much internal state
Rabbit	1&2	no	-	-	-	×	not free for all
salsa20	1	yes	512	256 64	0	×	Too much internal state, disputed ca
Sfinks	2A	yes	256	80 80	64k or logic	✓	ok
Sosemanuk	1	yes	512	128/256 128	0.5k	×	Too much internal state
SSS	1A& 2A	yes	-	-	-	×	broken by J. Daemen 10 secs on a PC
Trbdk3 yaea	1&2	no	-	-	-	×	not free for all
Trivium	2	yes	288	80 80	0	✓	ok, linear ca to date shows strength
TSC-3	2	yes	-	-	-	×	broken, linear ca in 4 mins on a PC
VEST	2A	no	-	-	-	×	not free for all
WG	2	yes	-	-	-	×	broken, chosen IV attack
Yamb	1&2	yes	>3k	256 128	0	×	Too much internal state
ZK-Crypt	2	No	-	-	-	×	not free for all

This first-pass selection, as illustrated in Table 1 above, has resulted in a short list of six for further investigation:

Grain, Hermes-8, Mosquito, Phelix, Sfinks and Trivium.

3 Hardware Implementation

3.1 Method

Of the remaining six selected candidates, three are in the 1A and 2A profiles which offer a message authentication code (MAC) in addition to the stream cipher output. In order to achieve a fair comparison against the other candidates, the designs were implemented as pure stream ciphers out without any of the additional resources required for supporting MAC generation.

Some of the developers quoted hardware results to different degrees of confidence from "rough estimates" to detailed implementation details. However, there was no consistent methodology used and results differed greatly depending on a variety of factors such as the number of gates or transistors required to make up a D-type flip-flop and the supported interfacing. Such variations make it impossible to directly compare the developers' hardware results. Thus the decision was taken to develop an independent set of hardware results.

Stream ciphers are required to operate on a stream of bits thus the decision was taken to use a synchronous serial style of interface for input and output of the plaintext and ciphertext. More flexibility was adopted for entry of the key to be either serially or utilise a short word parallel format (eg 8 or 32 bits at a time).

The designs were developed for low resources, sacrificing throughput in the interests of saving area. First, results were obtained for Xilinx FPGA using ISE version 6.3 and Altera FPGA using Quartus II version 5.0 both of which use 0.13 μm CMOS processes. In line with the low resource nature of eStream, the smaller Spartan-II devices were selected (XC2S15, XC2S30 and XC2S50). The smallest available Altera Cyclone (EP1C3T100C7) is considerably larger than the smallest Spartan II parts thus the same part was used for all the designs. ASIC results for a commercial 0.13μm standard cell process were obtained using a Cadence Physically Knowledgeable Synthesis (PKS) version 5.14 design flow for Synthesis, Place and Route (SP&R) using PKS, BuildGates, AmbitWare, standard cell technology library and SiliconEnsemble. The flow incorporated worst-case parasitic extraction and back annotation using foundry data. Verification included static timing analysis, design rule checks, generation of expected switching data using ModelSim and power results from Cadence LPS.

3.2 Defining Performance

The results quoted for the FPGAs are actual post place and route results (not synthesis estimates). The maximum clock rate for the design together with the selected FPGA device and its area utilisation are given. However, due to the richness of modern FPGA fabric this alone would not be representative of the likely device performance for ASIC so a further gate based analysis is given.

For this analysis, throughput performance was measured in millions of bits per second (Mbps) for the output of ciphertext neglecting any initialisation time. The area of an FPGA is normally measured in terms of its cell usage: slices for Xilinx and Logic Elements (LE) for Altera.

To avoid specific metrics for individual devices, it is proposed to use the "gates" metric for measuring area. In this paper, one "gate" is equivalent to the area occupied by a two input NAND gate (6 transistors). Thus a two input XOR gate typically occupies an area equivalent to 2.33 NAND gates (14 transistors). The implementation of a D-type flip-flop is much more variable depending on what auxiliary inputs (eg preset, clear, clock enable) are required. In this paper an 8 gate equivalent for the flip-flop was chosen.

To allow readers to calculate their own gate count for different gates-per-flip-flop, the quoted gate results are separated into two figures one for flip-flops and the second for all other gates. On many processes, by sacrificing flip-flop functionality such as preset, reset and clock-enable, the overall "gate" count may be reduced.

These relatively modern FPGA devices have a rich fabric supporting a number of distributed memory storage primitives. The effectiveness of these, in particular the Xilinx SRL16, depends on precisely how the algorithm uses its memory storage elements. Some ciphers make good use of such FPGA-area saving components and others less so. There is a further complication in that the FPGA synthesis tools generally attempt to yield the most "adaptable" design fitting within the given speed and area constraints. This is done to minimise the impact of relatively minor design changes for a waterfall development cycle often used in prototyping. The area constraint is typically defined with a rectangle thus for low resource designs the area utilised is dominated by how well the design tessellates with the chosen rectangle rather than it minimum resource utilisation.

To overcome this issue, an alternative approach was taken rather than to simply quote the number of "slices" reported by the post place and route report. In our second approach, the map report was examined and the number of LUT Function Generators (FG) and associated resource such as carry-chains (counted as equivalent to an FG) were extracted together with the number of D-type flip-flops (FD). On some FPGAs, the LUTs may also be configured as memory resources (ROM or RAM) these figures were also obtained from the map report. Of particular concern was how to correctly account for the use of the SRL16 (16x1 bit shift register) resource. The decision was taken to only account for this in terms of the actual number of bits used for the given design. For example, if only a 6 bit SR was needed then account for this as 8x6 gates rather than 8x16. This approach is believed to be equivalent to a gate level analysis and is more representative of the likely ASIC results.

From the FD, LUT and memory (MEM) values (SRL, ROM & RAM) an equivalent ASIC 2-input NAND gate count was estimated as follows:

$$\text{gates} = 6 \times \text{FG} + 8 \times \text{FD} + 8 \times \text{bits} \times \text{MEM}$$

In general terms, there are two different goals for a "low resource" design. Firstly, designers may be concerned with minimising the peak power consumption. This is typical in inductively powered contact-less smart cards. However, for battery powered systems it is more important to minimise the total energy consumption. For the latter, a typical goal is the minimisation of the power-area product.

Both design objectives are sensitive to area, so here, as this is a first attempt at comparison between the stream ciphers it was chosen to simply minimise the area. A typical academic metric for efficiency would be to minimise the area-time product. "Time" being the time taken to perform the cryptographic operation. The power consumption, for CMOS, is dominated by the number of transitions per second (thus datapath width and the clock frequency).

The simple area-time product metric favours highly parallel pipelined loop unrolled designs which generally would not be described as "low resource". Area alone could be considered as the performance metric for a low resource design however would not discriminate between two designs of differing throughput which required the same area. A suitable metric must include both throughput and area weighted in such a way to avoid favouring simply unrolling a design to improve its "performance".

One option is to select a throughput based loosely on the currently emerging wireless data standards, say 5 to 15 Mbps and develop implementations of the ciphers to meet this rate by selecting the appropriate clock frequency. However, it is also common to design for a higher rate, say 100Mbps and then calculate power consumption at a reduced clock rate.

The formulation of such a metric would be entirely subjective, thus the decision was taken to present the results graphically with a set of lines indicating constant metric value for different formulations of the metric and leave it for potential readers to make their own judgements.

The ASIC power results were obtained by stimulating a cell-level back-annotated simulation model of the design under test with random test vectors. ModelSim was used to obtain switching data in terms of a value change dump. This data was converted to a suitable format and combined with foundry supplied power models for the cells to yield the expected modelled power results. A basic MonteCarlo analysis was carried out by repeating the results a number of times with different test vectors in order to validate the accuracy of the results (<1% error). The results incorporate both initialisation and operational phases of the design under test.

4 Results

4.1 FPGA Implementation Results

Table 2 summarises the results obtained for each of the selected ciphers. In the interest of completeness, the original developers' results are also presented where available. Details of the designs adopted and any design modifications made are illustrated in Appendix A for each of these ciphers. For readers interested solely in FPGA design then their attention is drawn to the device, slices and LE results. The smallest available Xilinx Spartan II device is the XC2S15, only the AES-B, Trivium-1, Grain-1 and Mosquito-B designs will fit within this device. Fig. 2, effectively shows throughput versus area for the Xilinx Spartan II FPGA (0.13 μm process) and Fig. 3 the corresponding results for the Altera Cyclone FPGA (0.13 μm process).

For these designs, the Altera results are generally the faster and in terms of throughput and the relative performance of the different designs more closely follows the gate level analysis. An approximate equivalence of 2 LE = 1 SLICE may be used to perform a crude comparison in terms of area. Thus, the processor style architectures (Phelix-C, Hermes8) occupy less area on the Xilinx FPGA.

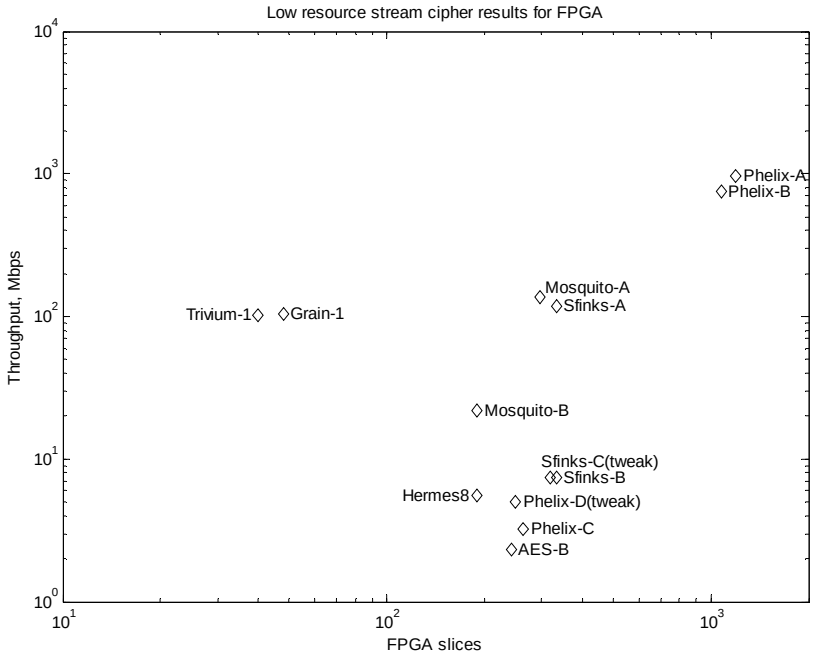


Fig. 2. Xilinx FPGA results

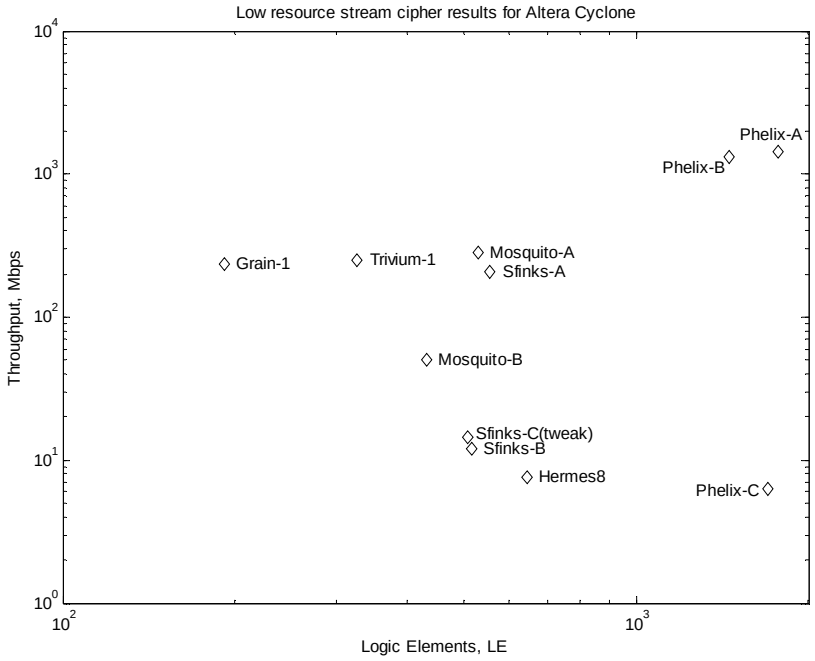


Fig. 3. Altera FPGA results

However, for ASIC designers, the “gates” column is more likely to be of interest. This clearly shows that Grain-1 and Trivium-1 are by far the smallest, yet still provide good throughput figures.

Table 2. FPGA results and gate level analysis

Cipher Design	Authors hardware results	Xilinx Spartan II FPGA	Altera Cyclone FPGA	Equiv. gates estimate	Notes
AES-A	0.35µm CMOS (Philips) 9 Mbps, 3,500 "gates" [3]	(ASIC) 9 Mbps	no result	6000	Gate count increased by 2500 to allow for feedback mode support
AES-B	Our basis for comparison [2]	XC2S15-5 2.34 Mbps 242 slices	no result	10426	Our ASIP design supporting OFB, CTR and CFB modes
Trivium-1	3488 gates [4]	XC2S15-5 102 Mbps 40 slices	EP1C3T-C7 249 Mbps 327 LE	2682	
Grain-1	ALTERA: 1435 "gates" MAX3000A 49Mbps MAX-II 200 Mbps Cyclone 282 Mbps [5]	XC2S15-5 105 Mbps 48 slices	EP1C3T-C7 335 Mbps 191 LE	1714	
Mosquito-A	Xilinx Virtex I 179 Mbps, 252 CLB & other FPGA results [6]	XC2S30-5 137 Mbps 298 slices	EP1C3T-C7 280 Mbps 530 LE	6844	(A) Pipelined as developers' paper
Mosquito-B		XC2S15-5 22 Mbps 190 slices	EP1C3T-C7 50 Mbps 431 LE	4178	(B) Our resource shared design (common hardware for logic stages 2-5)
Phelix-A	"Rough" estimates of 2Gbps, 20,000 gates [7]	XC2S100-5 960 Mbps 1198 slices	EP1C3T-C7 1440 Mbps 1772 LE	20404	(A) Full-round 160-bit design, as per developers paper
Phelix-B		XC2S100-5 750 Mbps 1077 slices	EP1C3T-C7 1312 Mbps 1455 LE	18080	(B) Our half-round 160-bit design
Phelix-C		XC2S30-5 3.26 Mbps 264 slices	EP1C3T-C7 6.31 Mbps 1697 LE	12314	(C) Our 32-bit datapath, control adversely affects area
Phelix-D		XC2S30-5 ~5 Mbps ~250 slices	no result	~8800	(D) Estimate initialisation was tweaked to simplify architecture
Sfinks-A	5265 gates (excluding MAC) [8]	XC2S30-5 118 Mbps 334 slices	EP1C3T-C7 207 Mbps 556 LE	5904	(A) Pipelined as per developers' paper
Sfinks-B		XC2S30-5 7.4 Mbps 334 slices	EP1C3T-C7 12.0 Mbps 517 LE	4910	(B) Our design comprising resource sharing in inversion – frustrated by requirements of initialisation (thus not efficient design)
Sfinks-C		XC2S30-5 7.4 Mbps 319 slices	EP1C3T-C7 14.6 Mbps 508 LE	3946	(C) Tweaked to remove feedback delay needed for initialisation
Hermes8	0.35 CMOS 4,026 gates (std cell) [9]	XC2S30-5 5.6 Mbps 190 slices	EP1C3T-C7 7.6 Mbps 645 LE	5022	Our 8-bit datapath architecture inclusive of control.

The results can be even more clearly expressed graphically in terms of throughput and area. In terms of area the further left, the smaller the design. In terms of speed the higher up the faster the design. As discussed in the method section of this paper, some "performance" metric would be most expedient.

Fig. 4 depicts the results with the dashed lines show constant speed versus area for each given design, however, this metric favours loop-unrolled and pipelined architectures so may not be considered the most appropriate.

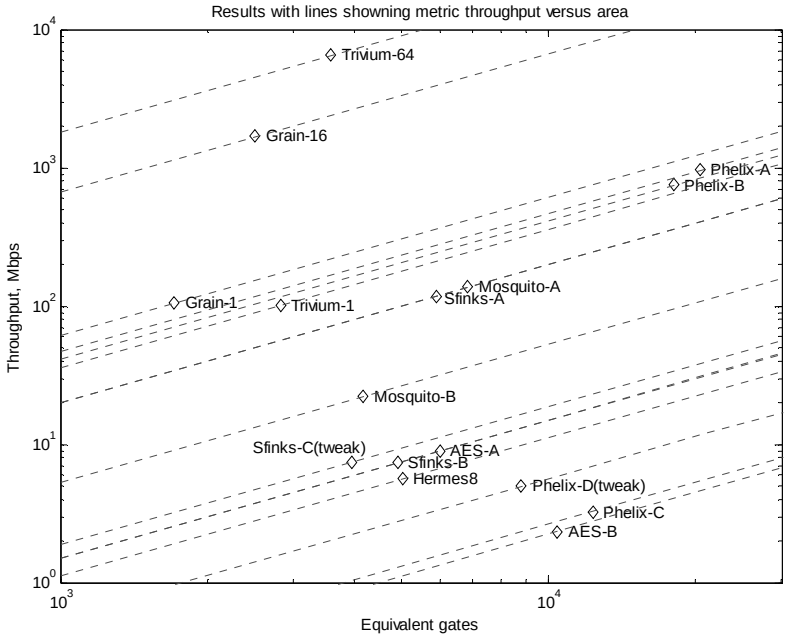


Fig. 4. Results annotated with lines of constant throughput versus area

The performance metric can be skewed more in favour of area by raising the area to a higher power than the throughput. Fig. 5 once again shows the low resource designs however this time the dashed lines are lines of constant area^2 versus speed.

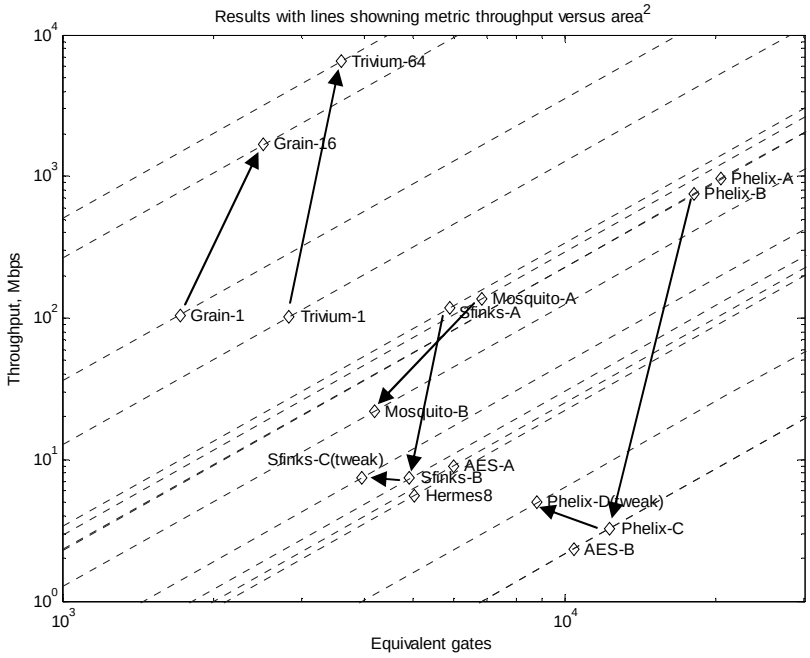


Fig. 5. Results annotated with lines of constant throughput versus area^2

However, as can be seen by the Grain-n and Trivium-n designs, the metric still favours unrolling and parallelism. The area is now raised to a still higher power (area^7.3) such that the resulting metric is now approximately neutral to the parallel construction of the smallest candidate. The resulting graph is presented as Fig. 6.

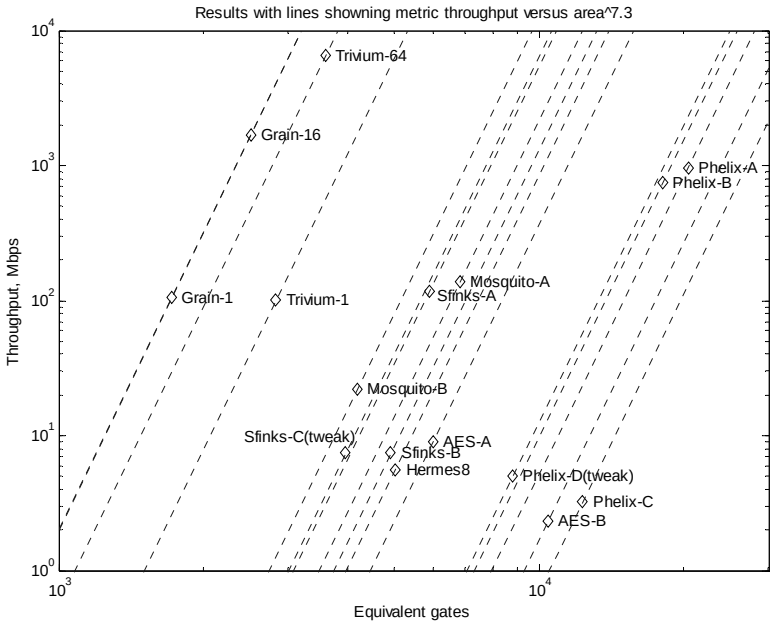


Fig. 6. Results annotated with lines of constant throughput versus area$^{7.3}$

This still may not be considered to be sufficiently area skewed so a final graph, Fig. 7, is presented for raising the area to the fifteenth power (as an extreme example). This is done to further illustrate that irrespective of the choice of cost function that both Grain and Trivium stand out as the lowest resource.

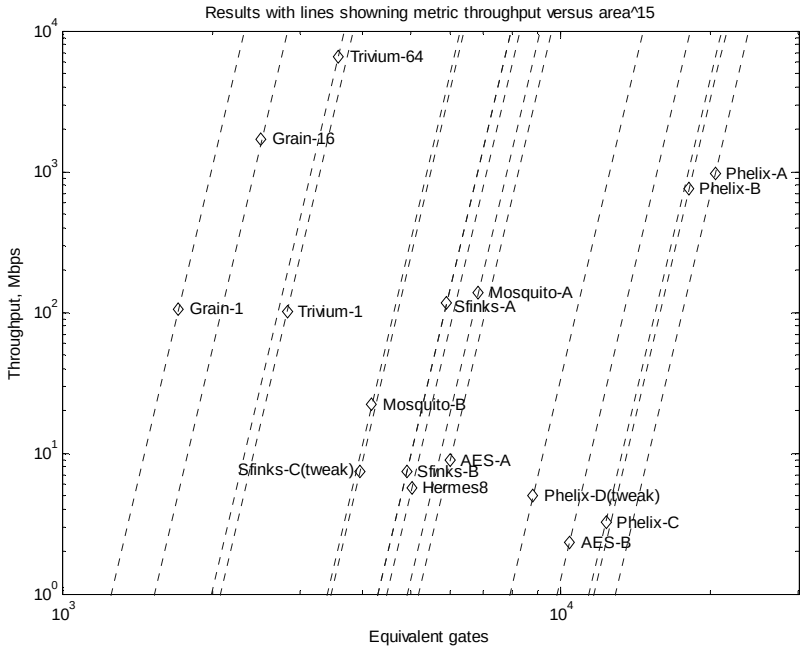


Fig. 7. Results annotated with lines of constant throughput versus area^{15}

For low resource design the metric "area^n * time" has been presented. The choice of a suitable value of n is subjective thus illustrative examples have been given for selected values between 1 and 15. It has been shown for the smallest candidate that a value of n=7.3 makes the metric neutral to pipelining. This value should be considered as the upper limit for n. A sensible choice would be to choose a value somewhere between the extremes of area * time (n=1 and 7.3), say, on a purely subjective basis n=2. However, irrespective of the precise value of n, as shown by the different results graphs, conclusions can be drawn and the selected ciphers categorised.

4.2 ASIC Results

To confirm the gates analysis above, obtain power results, and also for the sake of completeness, ASIC results were also obtained for a 0.13μm standard cell process using the Cadence Physically Knowledgeable Synthesis (PKS) flow. The results shown (Table 3) are the expected modelled results for the technology. The area is the occupied core area including routing. For readers wishing an ASIC 2-input NAND gate estimate simply multiply the area in μm^2 by 0.193. The power results were obtained using switching data resulting from loading the key and IV followed by initialisation and the encryption of a 10kbit stream of random data. Statistics from three different runs were compared in a basic Monte-Carlo analysis to validate the power results.

Table 3. ASIC throughput-area-power results

Design	Throughput, Mbps	Clock Period, ns	Critical path delay, ns	Area, μm^2	Power, mW
Trivium-1	1 10 100	1000 100 10	2.39	15,058	0.0347 0.227 2.154
Grain-1	1 10 100	1000 100 10	2.18	8,073	0.0238 0.156 1.476
Mosquito-A	1 10 100	1000 100 10	3.11 3.15 3.15	52,155 52,023 52,023	0.178 1.027 9.520
Mosquito-B resource shared	1 10 100	200 20 (2)	2.16 2.14	24,903 24,903 (no result)	0.137 1.136
Sfinks-A	1 10 100	1000 100 10	9.43	33,167	0.253 2.207 21.75
Sfinks-B resource shared	1 10 100	200 20 (2)	12.05 12.01	32,702 32,702 (no result)	2.211 21.83
Hermes8	1 10 100	125 12.5 (1.25)	7.36 7.34	35,672 35,773 (no result)	0.429 (tbc) 3.834 (tbc)

The results are summarised in terms of power versus area in Fig. 8. This figure shows that in terms of power-area efficiency Grain is the most efficient closely followed by Trivium. It also clearly shows the advantage of utilising a resource shared design for Mosquito. The power results again highlight the difficulty in attempting resource sharing for Sfinks (point too far off graph to plot).

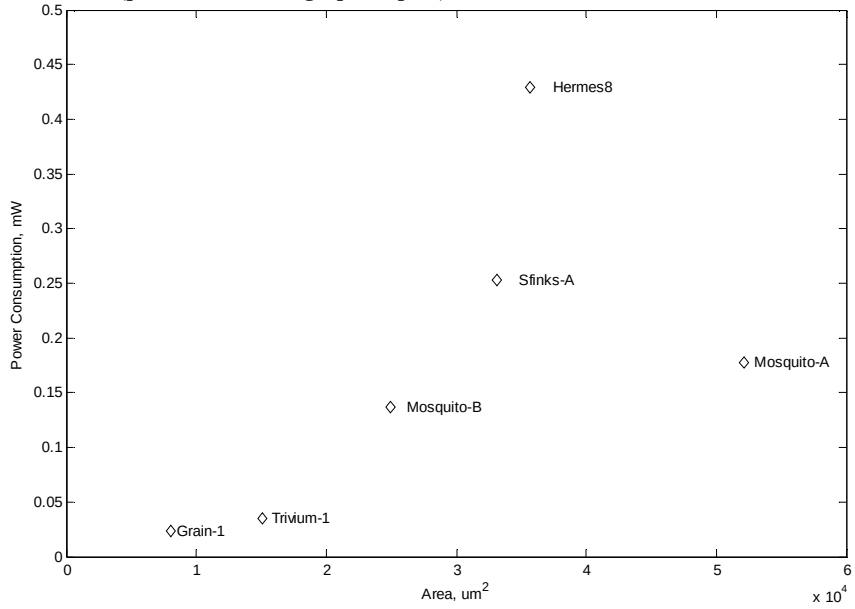


Fig. 8. ASIC results of power consumption versus area for designs operating at 1Mbps

5 Conclusions

Irrespective of how the results are presented Grain and Trivium are the smallest and most efficient designs and have straight forward parallel implementation which may ultimately be desirable to further enhance throughput and achieve improved energy per bit performance. The authors wish to urge those interested in the stream cipher project to analyse these thoroughly from a security perspective.

There is much more debate on which are the next ciphers to "perform" the best so they have been simply grouped together (Mosquito, Sfinks, Hermes8). More controversial, would be where to rank Phelix, in this paper it has been categorised as "moderate resource" due to its size not withstanding its higher throughput.

The authors of this paper do not wish to pass any comment (or expend effort) on those candidate ciphers which are not "free-for-all". It is left to others to carry out a similar analysis.

In summary, in terms of "low resource" hardware the considered candidates may be conveniently and fairly grouped as follows:

Category	**Candidate ciphers for low resource <u>hardware</u>**
Lowest Resource, High speed (~100Mbps)	Grain, Trivium
Low resource, moderate speed (~10Mbps)	Mosquito, Sfinks, Hermes-8
Moderate resource (~1000Mbps)	Phelix
High resource or broken	ABC, Achterbahn, Dicing, Dragon, F-FCSR, HC-256, MAG, MICKEY, Mir-1, NLS, Polar Bear, Pomaranch, Py ("Roo"), salsa20, Sosemanuk, SSS, TSC-3, WG, Yamb
Commercial i.e. "not free for all" (not considered in this treatment)	CryptMT, Decim, Edon80, Frogbit, Lex, Rabbit, Trbdk3, Vest, ZK-crypt

In summary, the purpose of this analysis is to encourage the security analysis community to direct their efforts towards analysing the security of the lowest resource candidates first before moving on to those requiring more resources. The benefits to this approach are two fold: firstly avoids wasted effort analysing a candidate which may not be considered to be "low resource" and secondly early rejection of those with low resource on security grounds will enable the hardware engineers to focus on adding side channel resistance to the remaining lower resource ciphers again avoiding wasted effort.

References

1 The eStream web site, http://www.ecrypt.eu.org/stream/

2 T. Good and M. Benaissa, “AES as a stream cipher on a small FPGA”, to appear ISCAS 2006.

3 M. Feldhofer, J. Wolkerstorfer and V. Rijmen, “AES implementation on a grain of sand”, IEE Proc. Info. Sec, Vol. 1, pp 13-20, 2005

4 C. de Canniere and B.Preneel, “Trivium Specifications”, http://www.ecrypt.eu.org/stream/

5 M.Hell, T.Johansson and W.Meier, “Grain – a stream cipher for constrained environments”, http://www.ecrypt.eu.org/stream/

6 J. Daemen and P. Kitsos, “Submission to ECRYPT call for stream ciphers: the self-synchronizing stream cipher Mosquito”, http://www.ecrypt.eu.org/stream/

7 D. Whiting, B.Schneier, S.Lucks and F.Muller, “Phelix: fast encryption and authentication in a single cryptographic primitive”, http://www.ecrypt.eu.org/stream/

8 A.Braeken, J.Lano, N.Mentens, B.Preneel and I.Verbauwhede, “SFINKS: A synchronous stream cipher for restricted hardware environments”, http://www.ecrypt.eu.org/stream/

9 U. Kaiser, “Hermes-8”, http://www.ecrypt.eu.org/stream/

10 NIST, “Recommendation for block cipher modes of operation”, Special Publication 800-38A, 2001, http://www.nist.gov/

11 NIST, “The Advanced Encryption Standard”, FIPS-197, http://www.nist.gov/

Acknowledgements

Funding by the UK Engineering and Physical Sciences Research Council (EPSRC) is acknowledged.

The authors wish to thank the developers of the candidate ciphers for all their commitment and effort in putting forward a submission and further for their assistance in understanding and resolving minor discrepancies between the descriptions and reference designs.

Appendix: A. Design Details

In the following sections, a brief description of each of the considered candidate algorithm is given and should be read in conjunction with the developers' original paper. Our designs and implementation results for each are given including where appropriate suggestions of possible tweaks to initialisation which may permit reduction of the required hardware resources.

A.1 AES (baseline)

The AES in a suitable feedback mode (eg Output Feedback) could be used as a "tried-and-tested" stream cipher. However, it is evident from the call that for "low resource" there is an aspiration to do better. Thus the AES forms one of the best baselines to date in terms of known security and as it was stated in the call as a suitable basis for comparison for the software profile it would be a sound judgement to use its low resource hardware implementations as a basis for comparison for the hardware ones too.

A previous FPGA design by the authors [2] looked at an 8-bit ASIP which supported three of the recognised [10] feedback modes for the Advanced Encryption Standard [11]. The modes were Output Feedback (OFB), Counter (CTR) and Cipher FeedBack (CFB) all of which generate a key stream which is then combined with the plain/cipher text using the XOR operation. This is an example of using a block cipher (such as AES) in a feedback mode to make it suitable for stream cipher applications. The use of block memory can be allowed for by adjusting the slice count with a cost of 32bits/slice for block memory usage.

A recently published [3] design for the AES showed that it is possible to construct a low resource ASIC to perform the core functions of the AES. With suitable additional memory, logic and interfacing it could operate autonomously in one of the feedback modes (OFB, CTR or CFB) to provide a low resource stream cipher. The additional logic including shift registers to support serial I/O and additional storage for key and IV required by a feedback mode such as OFB or CFB is estimated to total an additional 2500 gates.

Table 4. Implementation results for the AES

Design	Details	FPGA results	Gate level analysis (for Xilinx)
AES-A	Feldhofer's ASIC design, 3500 gates @ 9 Mbps on 0.35um ASIC Additional logic for feedback mode and serial I/o ~2500 gates	(ASIC result)	throughput: 9 Mbps approx. flip flop gates: 4848 approx. other gates: 1152 approx. total gates: 6000
AES-B	all: FG 211 FD 184 RAM 608bits ROM 200x16 bits	Xilinx (ISE): device: XC2S15-5 clock: 70 MHz bits/cycle: 128/3828 slices: 242 (120 + 2xBLKRAM)	throughput : 2.34 Mbps block RAM gates: 5220 block ROM gates: 2468 other flip flop gates: 1472 other gates : 1266 total FPGA gates: 10426

A.2 Trivium

Trivium [4] is a stream cipher consisting of three shift registers with interconnected non-linear feedback functions to form a recognisable Substitution-Permutation-Network and a final linear function is used to create the keystream. The shift registers are of different lengths (93, 84 and 111 bits) and all the feedback functions only combine five taps.

The feedback and output functions may be expressed in terms of their constituent taps as follows:

$t1(S) = S_{66} + S_{91}.S_{92} + S_{93} + S_{171}$

$t2(S) = S_{162} + S_{175}.S_{176} + S_{177} + S_{264}$

$t3(S) = S_{69} + S_{243} + S_{286}.S_{287} + S_{288}$

$z(S) = S_{66} + S_{93} + S_{162} + S_{177} + S_{243} + S_{288}$

To load the key the bit stream is (externally) prepared by padding out the key and IV to the required 288 bits as follows:

$S_{1..288} = K_{1..80}, \text{`0'}_{14}, IV_{1..80}, \text{`0'}_{111}, \text{`1'}, \text{`1'}, \text{`1'}$

The control was implemented using a state machine supported by an 11-bit counter to generate the necessary control (key loading, clocking and output latching) and handshaking signals. Loading the key-IV-padding word takes 288 cycles followed by 1152 cycles (4x288) of key mixing with the output suppressed. After initialisation one bit of keystream is output every cycle.

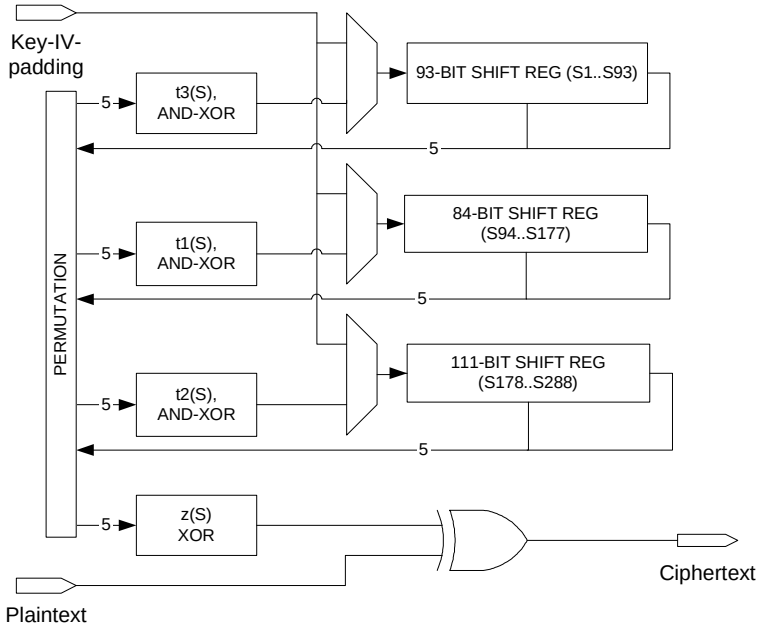


Fig. 9. Block diagram of Trivium

The design is very small and offers little scope for optimisation other than the usual logic and gate level manipulation which most synthesis tools perform automatically.

Table 5. Implementation results for Trivium

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Trivium-1	sr: FD 29 SRL 21 (or FD 288) funcs(t1,t2,t3,z): FG 27 ctrl: FG 36 FD 19 all: FG 63 FD 48 SRL 21 (or FG 63 FD 307)	Xilinx (ISE): device: XC2S15-5 clock: 102 MHz bits/cycle: 1 slices: 40 Altera (Quartus II): device: EP1C3T144C7 clock: 249 MHz area: 327 LE t'put: 249 Mbps	throughput: 102 Mbps flip flop gates: 2456 other gates : 378 total FPGA gates: 2834
Trivium-n	sr: FD 288 funcs: FG 25+2n ctrl: FG 36 FD 19 all: FG 61+2n FD 307	Xilinx (ISE): device: Spartan 2 clock: 102 MHz bits/cycle: n (n_{max} = 64)	Estimate for parallel generation throughput: 102n Mbps for n=64: 6528 Mbps total FPGA gates: 2822+12n for n=64: 3590

As shown in [4] it is possible to use parallel computation to enhance throughput without increasing the flip-flop count (up to x64). This will improve the throughput versus area metric but the overall area will be increased.

A.3 Grain

The grain submission [5] is a key stream generator comprising two 80-bit shift registers and three combinatorial functions, f(x), g(x) and h(x). The first, f(x) is a 7^{th} degree linear feedback polynomial for the first shift register. The second, g(x) is a non linear feedback polynomial utilising 11 taps of the second shift register with a maximum of 6 taps being ANDed together. The final nonlinear function, h(x) combines a total of 6 taps, here h(x) defined to include the XOR with the final output of shift register N, is used to create the keystream.

$$f(x) = x^0+x^{18}+x^{29}+x^{42}+x^{57}+x^{67}+x^{80}$$

$$\begin{aligned} g(x) = {} & x^0+x^{17}+x^{20}+x^{28}+x^{35}+x^{43}+x^{47}+x^{52}+x^{59}+x^{65}+x^{71}+x^{80} \\ & + x^{17}.x^{20} + x^{43}.x^{47} + x^{65}.x^{71} \\ & + x^{20}.x^{28}.x^{35} + x^{47}.x^{52}.x^{59} \\ & + x^{17}.x^{35}.x^{52}.x^{71} + x^{20}.x^{28}.x^{43}.x^{47} + x^{17}.x^{20}.x^{59}.x^{65} \\ & + x^{17}.x^{20}.x^{28}.x^{35}.x^{43} + x^{47}.x^{52}.x^{59}.x^{65}.x^{71} \\ & + x^{28}.x^{35}.x^{43}.x^{47}.x^{52}.x^{59} \end{aligned}$$

$$\begin{aligned} h(x) = {} & N^0 + L^{55} + N^{17} + L^{77}.L^{16} + L^{34}.L^{16} + L^{16}.N^{17} \\ & + L^{77}.L^{55}.L^{34} + L^{77}.L^{34}.L^{16} + L^{77}.L^{34}.N^{17} + L^{55}.L^{34}.N^{17} + L^{34}.L^{16}.N^{17} \end{aligned}$$

For initialisation the shift registers are loaded with key and IV (padded with ones to 80 bits). Initial key-IV mixing is then carried out for 160 cycles with the "keystream output bit" being fed back to both shift registers (using XOR).

Control was implemented using a finite state machine supported by an 8-bit counter. The overall design may be summarised by the following diagram.

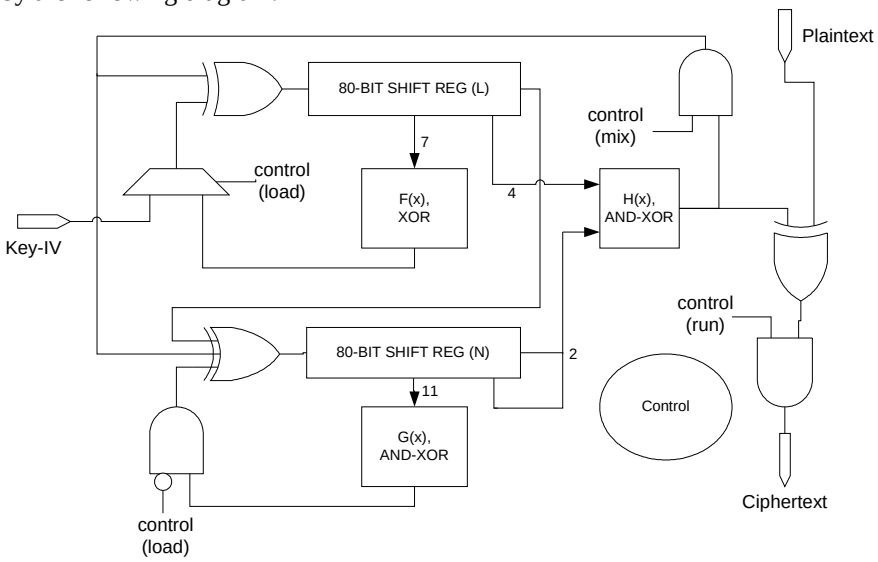


Fig. 10. Block diagram of Grain

The design is relatively simple and offers little scope for optimisation above the usual logic/gate-level optimisations that modern synthesis tools will automatically perform. The implementation had synchronous serial interfaces for cipher/plain text I/O and a separate serial input for loading key-IV.

Table 6. Implementation results for Grain

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Grain-1	sr: FD 22 SRL 19 funcs(f,g,h): FG 26 ctrl: FG 29 FD 13 all: FG 55 FD 35 SRL 19 (or FG 55 FD 173)	Xilinx (ISE): device: XC2S15-5 clock: 105 MHz bits/cycle: 1 slices: 48 Altera (Quartus II): device: EP1C3T144C7 clock: 235 MHz area: 191 LE t'put: 235 Mbps	throughput: 105 Mbps flip flop gates: 1384 other gates : 330 total FPGA gates: 1714
Grain-n	sr: FD 160 funcs: FG 16+10n ctrl: FG 29 FD 13 all: FG 45+10n FD 173	Xilinx (ISE): device: Spartan 2 clock: 105 MHz bits/cycle: n (n_{max} = 16)	Estimate for parallel generation throughput: 105n Mbps for n=16: 1680 Mbps total FPGA gates: 1550+60n for n=16: 2510

The original paper on the design [5] described how the feedback functions can be paralleled (up to x16) to improve the throughput-area metric however this is at the expense of additional area.

A.4 Mosquito

The Mosquito self-synchronising stream cipher [6] is based around a non-linear shift register followed by a combinatorial function which yields a single bit of the keystream. The "conditional complementing shift register", CCSR, connects each storage element with a small logic function derived from the proceeding element together with a key bit, K, and two further proceeding bits of the CCSR. Here, this is referred to as stage 0 and is defined by

$$G_i^{<0>} = G_{i-1}^{<0>} + K_{i-1} + G_v^{<0>}.(G_w^{<0>}+1) + 1, \quad 0 \le i < 128$$

where v and w are both functions of the bit index i. These values are defined in table 1 of the Mosquito specification [6].

This equation essentially expresses, the elemental non-linear logic function (2xXOR, 1xNAND) used for all the logic stages. It has a convenient form in terms of FPGA implementation in that it is a 4-input 1-output function thus is described by a single LUT.

This function is repeated for seven combinational logic stages to produce the keystream bit, z, as described in table 7.

Table 7. Mosquito logic stages

Stage	Equation
1	$G_{4i \bmod 53}^{<1>} = G_{128-i}^{<0>} + G_{i+18}^{<0>} + G_{113-i}^{<0>}.(G_{i+1}^{<0>}+1) + 1, \ 0 \le i < 53$
2 to 5	$G_{4i \bmod 53}^{<j>} = G_i^{<j-1>} + G_{i+3}^{<j-1>} + G_{i+1}^{<j-1>}.(G_{i+2}^{<j-1>}+1) + 1, \ 0 \le i < 53$
6	$G_i^{<6>} = G_{4i}^{<5>} + G_{4i+3}^{<5>} + G_{4i+1}^{<5>}.(G_{4i+2}^{<5>}+1) + 1, \ 0 \le i < 12$
7	$G_i^{<7>} = G_{4i}^{<6>} + G_{4i+1}^{<6>} + G_{4i+2}^{<6>} + G_{4i+3}^{<6>}, \ 0 \le i < 3$
output	$z = G_0^{<7>} + G_1^{<7>} + G_2^{<7>}$

In this implementation, the resources for stages 2-5 share a single round based implementation, saving of 212 LUTs, at the cost of a 53-bit register and a 53-bit two-way multiplexer (53 LUTs and 53 DFFs). This is an equivalent saving of 530 gates at the cost of a factor of five reduction in throughput.

Once the 80-bit key has been entered serially together with the 128-bit IV (loaded into the CCSR), the initial key mixing of 105 iterations (each of 5 clock cycles) is performed with the plaintext input and ciphertext output zeroed. Subsequently, a new bit of keystream is available once in every 5 clock cycles. The control was implemented using a state machine supported by a 7-bit counter.

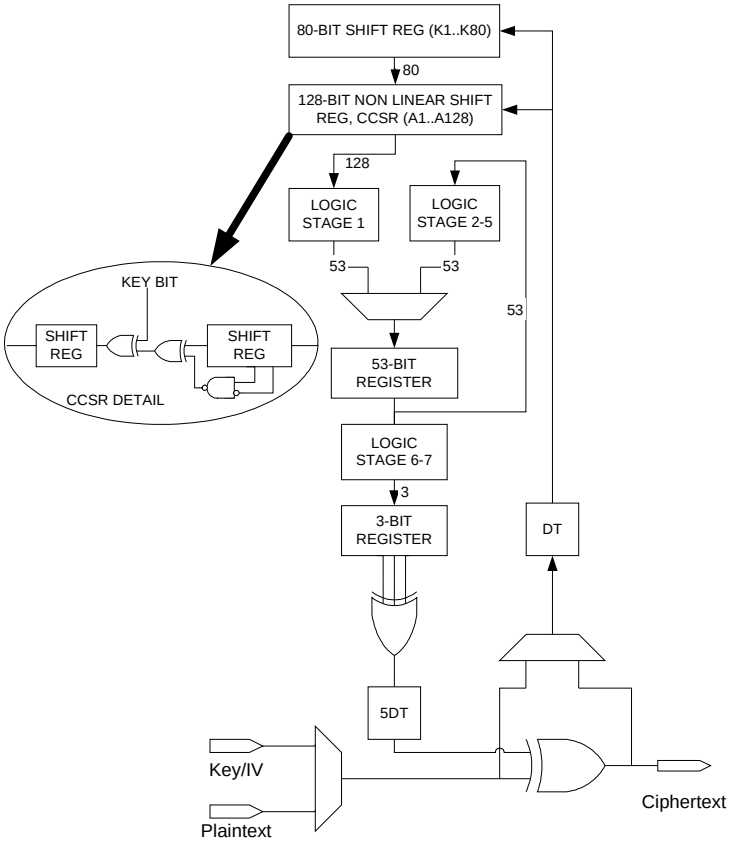


Fig. 11. Mosquito

The implementation results are tabulated below (Table 8), firstly for a repetition of the developers design (Mosquito-A) followed by the above resource shared architecture (Mosquito-B).

In these designs the key and IV were loaded serially, a tweak to the definition for initialisation would permit direct loading of the IV into the CCSR accepting the complementing due to the key. This would simplify the CCSR design avoiding needed the larger flip-flops with a reset capability.

Table 8. Implementation results for Mosquito

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Mosquito-A	Design as per developers paper all: FG 450 FD 518	Xilinx (ISE): device: XC2S30-5 clock: 137 MHz bits/cycle: 1 slices: 298 Altera (Quartus II): device: EP1C3T144C7 clock: 280 MHz area: 530 LE t'put: 280 Mbps	throughput: 137 Mbps flip flop gates : 4144 other gates: 2700 total FPGA gates: 6844
Mosquito-B	keyreg: FD 80 ccsr: FG 130 FD 128 stages: FG 122 FD 56 ctrl: FG 27 FD 23 other: FG 4 FD 23 all: FG 283 FD 305 SRL 1 (or FG 283 FD 310)	Xilinx (ISE): device: XC2S15-5 clock: 110 MHz bits/cycle: 1/5 slices: 190 Altera (Quartus II): device: EP1C3T144C7 clock: 254 MHz area: 431 LE t'put: 50 Mbps	throughput: 22 Mbps flip flop gates: 2480 other gates : 1698 total FPGA gates: 4178

A.5 Phelix

Philix [7] consists of five strands each of 32-bit data which are twisted together using shifting and arithmetic operations to form a helix like structure (hence its name). The cipher is supplied with a 256 bit key and 128-bit IV (nonce). Its operation could be viewed as a Feistal block cipher operating in a hybrid counter - cipher feedback mode to provide a keystream. However, as pointed out by the developers, when the MAC is not used then there exists a low complexity differential cryptanalysis against a CFB based decryptor. To avoid this, the "plaintext" applied to Quarter Round A should always be zero making the keystream generation a hybrid OFB-CTR mode. The datapath may be decomposed into a simple operator consisting of a programmable shift and 32-bit add/xor operation, thus a 32-bit processor style architecture could be considered as 20 rounds of this simple operator plus a key schedule computed using the same datapath. Additionally, Phelix supports a message authentication code which was not considered in these hardware results.

First, initial consideration is given to folding the round by a factor of two and four to exploit symmetry within the datapath, forming 160-bit half round and quarter round implementations respectively.

Folding in half is relatively straight forward and gains the expected approximate factor of two reduction in area from the unrolled baseline design. However, if a second fold is made to use a flexible quarter round function then the multiplexers required to select between hardwired shifts would negate the advantage. Thus the flexible quarter-round design has not been progressed further.

The half round function based design is approximately half the size of the unrolled design and would produce approximately half the throughput. However, the area required would be best described as "moderate resource" rather than low resource when compared with the AES but its expected throughput is much higher (due to its simple operations and wide datapath) than the other candidates.

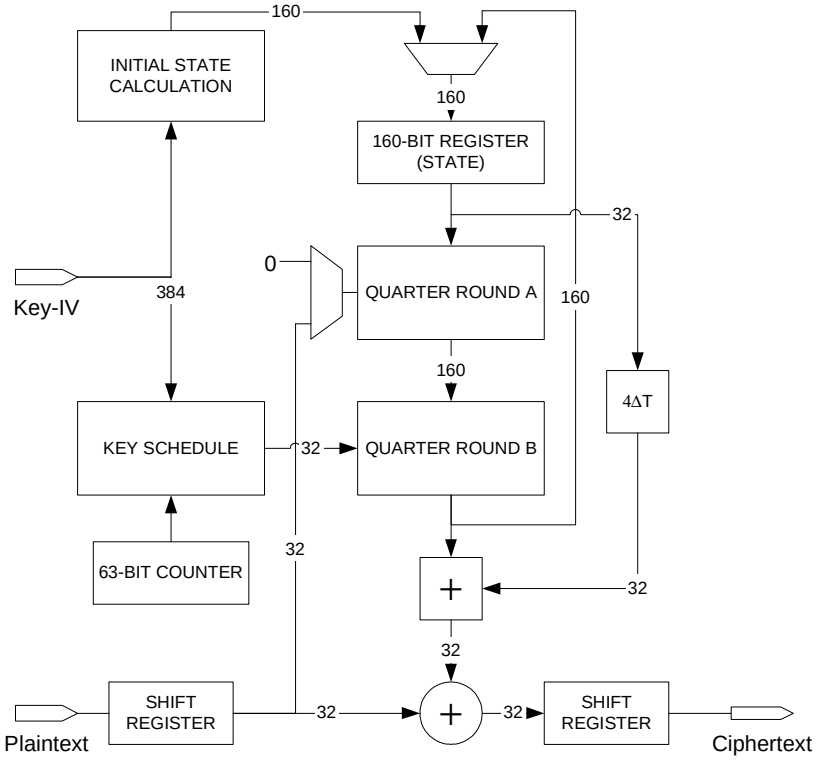


Fig. 12. Phelix half-round implementation

The next option was to consider a processor style architecture with a 32-bit datapath and controlled by state machines. The datapath consisted of a small register file (32x32-bit), a sequential shifter and configurable xor/add operation. The "height" in slices of the smaller FPGAs made it difficult to implement a fast ripple carry adder thus this was implemented as four separate 8-bit adders with registered carry propagation.

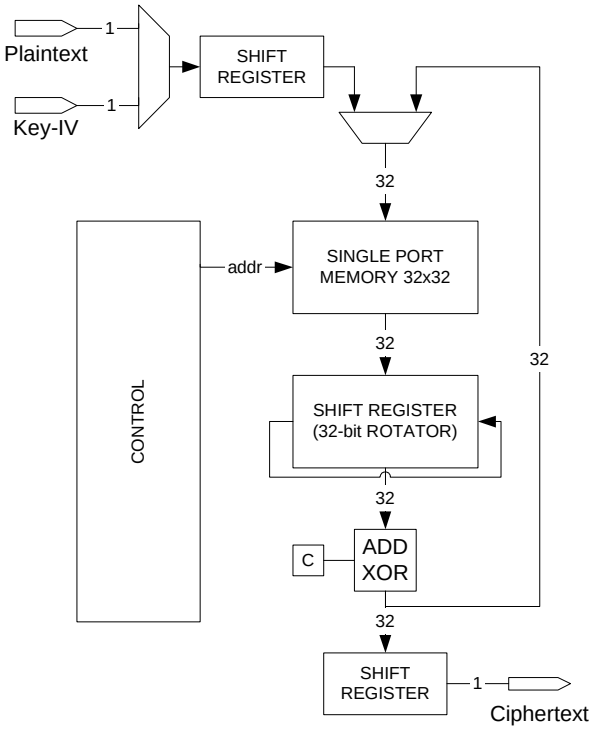


Fig. 13. Phelix datapath architecture

The register file consisted of 32 locations and contained the following registers:

Eight registers to hold the padded key;
Eight registers to hold the expanded nonce/IV;
Five registers for the current state (Z0, Z1, Z2, Z3, Z4);
Four registers to keep the previous four states of Z4;
Two registers to hold a 64-bit counter value;
Two temporary storage locations;
and three constants (zero, one and key length).

A 22-bit instruction word was defined to control the datapath and instructions supplied from a set of subroutines contained within a 64x22-bit ROM (implemented using random-logic) and controlled by a second state machine. In order to allow for a constant rate of data bits in and out a third state machine together with shift registers was used to control the input and output.

The main operations required for the "helix" structure are straight forward to implement however, the key schedule, although carried out using the same datapath, is more difficult and dominates the area for the controller. There may be some scope for "tweaks" to be considered for the initialisation of the key schedule to simplify its hardware implementation. One option would be a change to the key schedule so that a simple 64-bit counter could be loaded with the IV then simply incremented and incorporated as part of the key schedule together with some simplification of setting the initial "Z(-8)" state.

It should be stressed that these are initial results and some further optimisation may be possible. The ciphers area is dominated by the 512 bits required to store the expanded key and nonce. If some tweaks were permitted then the nonce could be loaded into the counter saving 256 bits of memory!

Considering an ASIC implementation, the constants and any constant bit (eg in key-length) may be hard wired further reducing the flip-flop count by an additional 93 bits. In total the saving, in flip-flops alone is estimated to be 2,792 equiv. gates. Further, if 32-bit I/O was acceptable then a further 710 gates could be saved. This results in a final estimate for a tweaked "Phelix" datapath based implementation of 8,800 gates. However, the 160-bit half round function only requires 7128 gates to implement which has simpler control and would be two orders of magnitude faster. However, it is the implementation of the keyschedule which is problematic and requires a number of 32-bit multiplexers and 32-bit binary adders together with two, partly overlapping 32-bit counters. The authors of this paper would urge the developers of Phelix to consider a revised keyschedule making more use of XOR rather than binary addition and being defined such that can be operated using a counter initially loaded with the nonce.

Table 9. Implementation results for Phelix

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Phelix-A	160-bit Whole round design datapath: FG 1282 FD 320 keysched: FG 972 FD 540 all: FG 2254 FD 860	Xilinx (ISE): device: XC2S100-5 clock: 30 MHz bits/cycle: 32 slices: 1198 Altera (Quartus II): device: EP1C3T144C7 clock: 45 MHz area: 1772 LE t'put: 1440 Mbps	throughput: 960 Mbps flip flop gates: 6880 other gates: 13524 total FPGA gates: 20404
Phelix-B	160-bit Half round design helicies: FG 544 mux/reg: FG 260 FD 197 MEM 4x32 keysched: FG 1036 FD 555 all: FG 1840 FD 752 MEM 4x32 (or FG 1840 FD 880)	Xilinx (ISE): device: XC2S100-5 clock: 47 MHz bits/cycle: 32/2 slices: 1077 Altera (Quartus II): device: EP1C3T144C7 clock: 82 MHz area: 1455 LE t'put: 1312 Mbps	throughput : 750 Mbps flip flop gates: 7040 other gates: 11040 total FPGA gates: 18080
Phelix-C	32-bit Datapath Architecture datapath: FG 128 FD 68 regfile: MEM32x32 ctrl: FG 240 FD 81 I/O: FG 33 FD 64 all: FG 403 FD 213 MEM32x32 (or FG 403 FD 1237)	Xilinx (ISE): device: XC2S30-5 clock: 30 MHz bits/cycle: 32 / 294 slices: 264 Altera (Quartus II): device: EP1C3T144C7 clock: 58 MHz area: 1697 LE t'put: 6.31 Mbps	throughput: 3.26 Mbps flip flop gates: 9896 other gates: 2418 total FPGA gates: 12314
Phelix-D (Tweak)	non-compliant estimate allowing for some tweaks datapath: FG 128 FD 68 regfile: FD 675 ctrl: FG 240 FD 81 all: FG 368 FD 824	Xilinx (ISE): device: Spartan 2 clock: 30 MHz bits/cycle: 32 / 192 slices: 250 ESTIMATES	throughput: ~ 5 Mbps flip flop gates: 6592 other gates: 2208 total FPGA gates: 8800 ESTIMATES

In summary, Phelix, as currently defined, is difficult to implement efficiently in a rolled-up architecture however, in its half-round form performs with high throughput so may be worthy of further consideration.

A.6 Sfinks

The Sfinks [8] cipher comprises a 256-bit shift register together with a 16-bit multiplicative inversion in GF 2^{16}. This inversion derives its 16-bit input from a set of taps within the shift register. A single bit of its output is combined with a further bit from the shift register is used to generate the keystream. However, all 16-bits are utilised in a permuted order during the initial key mixing process. Thus the inversion is used in its complete form to create a "strong" SPN network for key-IV mixing and as the reduction operation for generation of the keystream bits. The paper [8] included message authentication code, however, in order to be consistent with the pure stream cipher model adopted in this paper it was omitted.

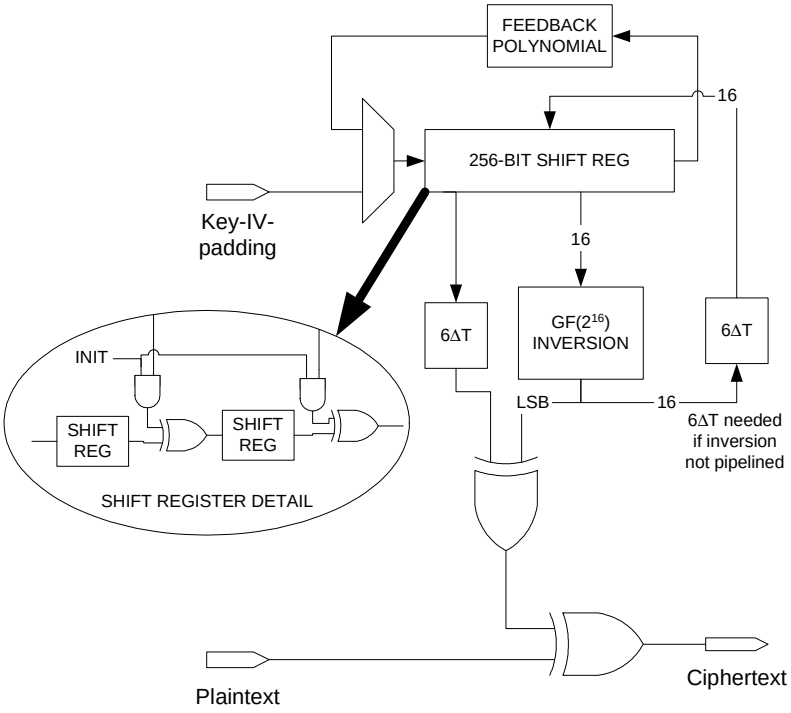


Fig. 14. Sfinks architecture

As shown by the AES, such multiplicative inverses can be efficiently implemented using composite field arithmetic thus the inverse was computed in the 16-bit GF((((2²)²)²)²) field with suitable isomorphisms. There are a number of opportunities for sharing resources within this inverse to reduce area at the expense of throughput. In the paper [8] the inversion was pipelined to enhance throughput, here the opposite approach is taken in that the 8-bit GF(((2²)²)²) multipliers and the 4-bit GF((2²)²) in the GF(((2²)²)²) inversion are resource shared using appropriate multiplexing and registers. Thus the inversion takes 5 cycles to complete.

The field construction was as follows:

Table 10. Sfinks composite field construction

Field	**Polynomial**	**Binary representation**
GF(2)	n/a	b_0
$GF(2^2)$	$P_u(u) = u^2+u+1$	$b_1u + b_0$
$GF((2^2)^2)$	$P_z(z) = z^2+z+u$	$z(b_3u+b_2) + b_1u+b_0$
$GF(((2^2)^2)^2)$	$P_y(y) = y^2+y+(uz+z)$	$y(b_7zu+b_6z+b_5u+b_4) + b_3zu+b_2z+b_1u+b_0$
$GF((((2^2)^2)^2)^2)$	$P_x(x) = x^2+x+uzy$	$x(b_{15}yzu+b_{14}yz+b_{13}yu+b_{12}y+b_{11}zu+b_{10}z+b_9u+b_8)$ $+b_7yzu+b_6yz+b_5yu+b_4y+b_3zu+b_2z+b_1u+b_0$

$$x^{-1}_{GF2^{16}}(v) \equiv \delta^{-1}\left(x^{-1}_{GF2^{2,2,2,2}}(\delta(v))\right)$$

and the isomorphisms between $GF(2^{16})$ and GF((((2²)²)²)²) may be represented in hexadecimal form as:

$$\delta(x)=0001.x_0+7C91.x_1+4604.x_2+43DA.x_3+6C13.x_4+7E9D.x_5+6B49.x_6+1190.x_7 +5A36.x_8+707F.x_9+454F.x_{10}+B430.x_{11}+5EFD.x_{12}+D6CA.x_{13}+104D.x_{14}+6A24.x_{15}$$

$$\delta^{-1}(x)=0001.x_0+ACCB.x_1+90C4.x_2+86FA.x_3+C583.x_4+AE57.x_5+7C62.x_6+8684.x_7 +444A.x_8+161C.x_9+C1D6.x_{10}+2D90.x_{11}+2A5D.x_{12}+C215.x_{13}+470A.x_{14}+4A4A.x_{15}$$

The resource sharing allows the reduction in gate count for the "operational" phase of the cipher but is somewhat frustrated by the initial key mixing stage. The algorithm requires that all 16-bits of the inversion to be fed back with a delay of 6-clocks which matches the developers' pipelined inversion. A resource-shared or non-pipelined version of the inversion would require an additional 96 flip-flops to implement the required delay to match the algorithm definition for initialisation. This effectively overcomes any advantage in area from using resource sharing in the inversion and mandates a 6-stage pipelined inversion. This may be one area where a "tweak" could be considered to permit more flexibility in terms of implementation (lower area or higher throughput).

The "Sfinks-A" design is essentially follows the developers intended architecture. Sfinks-B is a compliant design with resource sharing as described above. Finally, Sfinks-C shows the area saving if the design key mixing was changed to avoid the need to delay the inverse.

Table 11. Implementation results for Sfinks

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Sfinks-A	FPGA results for pipelined design lfsr FG 22 FD 262 inv: FG 289 FD 92 SRL 16 ctrl: FG 41 FD 18 SRL 1 all: FG 352 FD 372 SRL 17 (or FG 352 FD 474)	Xilinx (ISE): device: XC2S30-5 clock: 118 MHz bits/cycle: 1 slices: 334 Altera (Quartus II): device: EP1C3T144C7 clock: 207 MHz area: 556 LE t'put: 207 Mbps	throughput: 118 Mbps flip flop gates: 3792 other gates: 2112 total FPGA gates: 5904
Sfinks-B	compliant with SFINKS paper lfsr: FG 22 FD 262 inv: FG 177 FD 27 feedback: SRL 17 ctrl: FG 42 FD 42 all: FG 241 FD 331 SRL 17 (or FG241 FD 433)	Xilinx (ISE): device XC2S30-5 clock: 37 MHz bits/cycle: 1/5 slices: 334 Altera (Quartus II): device: EP1C3T144C7 clock: 60 MHz area: 517 LE t'put: 12.0 Mbps	throughput: 7.4 Mbps flip flop gates: 3464 other gates : 1446 total FPGA gates: 4910
Sfinks-C (tweak)	initialisation "tweaked" lfsr: FG 22 FD 262 inv: FG177 FD 27 ctrl: FG 40 FD 25 all: FG 239 FD 314	Xilinx (ISE): device: XC2S30-5 clock: 37 MHz bits/cycle: 1/5 slices: 319 Altera (Quartus II): device: EP1C3T144C7 clock: 73 MHz area: 508 LE t'put: 14.6 Mbps	throughput: 7.4 Mbps flip flop gates: 2512 other gates: 1434 total FPGA gates: 3946

A.7 Hermes-8

The Hermes-8 [9] has been designed around an 8-bit SPN architecture. The choice for the substitution operation was the well known 8-bit AES S-box. The permutation was carried out at the byte level (rather than the more usual bit level) by selecting differing indexes into the state and key registers.

The algorithm requires modulo arithmetic in order to carry out the indexed addressing (modulo-7, 10 and 23). In the running phase this can be accomplished by simply incrementing specific modulo counters which automatically reset when the correct modulus is reached. However, for initialisation these counters must be loaded with a modulo-value derived from the XOR of a number of key-bytes. The low resource implementation of this is to use conditional subtraction by the required modulus either for a fixed number of iterations or terminate when no further modulo reduction is required. The latter would leak significant side channel information during initialisation so would be most undesirable. Looking for an alternative for initialising the modulo counters would be a good starting point for a "tweak" to simplify hardware implementation.

The controller is split into two state machines, one specifically to control the datapath given an instruction word and the second to carry out the global control and generate the required sequence of instruction words. A

single port memory was used for the register file containing both key and state values (36x8 bits total). The controller also contained a number of counters: two off mod-23, mod-10 and mod-7. The values of these counters were used to provide all the necessary addresses for indexing into the register file.

The datapath consists of an 8-bit XOR operation, the AES S-box implemented using composite field arithmetic in $GF((2^2)^2)^2)$ with resource sharing of the 4-bit $GF((2^2)^2)$ multiplier and a dedicated unit for performing modulo reduction (conditional subtraction of modulus) of an 8-bit value and is only used in the initialisation phase.

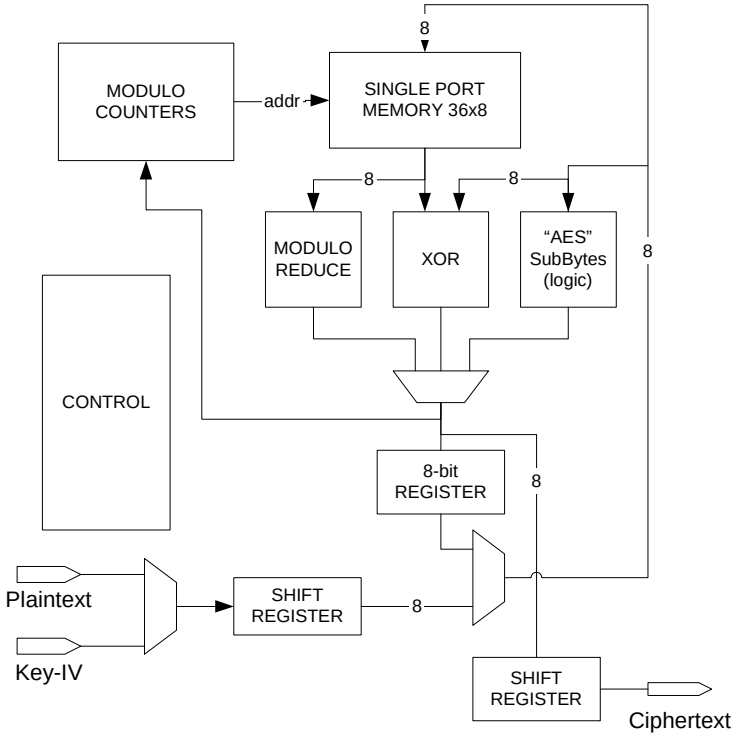


Fig. 15. Hermes-8 architecture

Table 12. Implementation results for Hermes-8

Design	Details	FPGA results	Gate level analysis (for Xilinx)
Hermes8	regfile: RAM32x16 datapath-less-sbox: FG 63 FD 8 sbox: FG 52 FD 21 ctrl: FG 148 FD 101 other: FG 14 FD 2 all: FG 277 FD 132 RAM32x16 (or FG 277 FD 420)	Xilinx (ISE): device: XC2S30-5 clock 45 MHz bits/cycle: 64 / 512 slices: 190 Altera (Quartus II): device: EP1C3T144C7 clock: 61 MHz area: 645 LE t'put: 7.6 Mbps	throughput: 5.6 Mbps flip flop gates: 3360 other gates: 1662 total FPGA gates: 5022

A Guess-and-Determine Attack on the Stream Cipher Polar Bear

John Mattsson[1,2]

[1] CSC, Royal Institute of Technology, Stockholm, Sweden
[2] Communications Security Lab, Ericsson Research, Stockholm, Sweden
john.mattsson@gmail.com

Abstract. In this paper we present an effective guess-and-determine attack against the stream cipher Polar Bear. The attack requires knowledge of the first 24 bytes of plaintext and recovers the state with a computational complexity of $O(2^{79})$. We also briefly discuss how this weakness can be addressed by the authors in an updated version of Polar Bear.

Keywords: Steam Cipher, Polar Bear, Guess-and-determine, eSTREAM

1 Introduction

There are a variety of efficient and trusted block ciphers. Unfortunately this is not the case for stream ciphers. As a response to this, ECRYPT (a 4-year network of excellence funded by the European Union) manages and co-ordinates a multi-year effort called eSTREAM to identify new stream ciphers suitable for widespread adoption. The new stream cipher Polar Bear [1] is one of 35 candidates submitted to eSTREAM. It was created by Johan Håstad and Mats Näslund and claimed to be suitable for both profile I (software) and profile II (hardware). In this paper we present the first known attack on Polar Bear. Recently a similar attack with improved complexity has been presented by Hasanzadeh *et al* [2]. We also analyze why this attack is possible and suggest how the cipher can be fixed to avoid this type of attack.

2 Description of Polar Bear

The cipher uses one 7-word (112-bit) LFSR R^0 and one 9-word (144-bit) LFSR R^1. These are viewed as acting over $\mathbb{F}_{2^{16}}$. Besides these registers, the internal state of the cipher also depends on a word quantity, S, and a dynamic permutation of bytes, D_8.

The cipher is primarily designed for a key length of 128 bits. The IV can be any number of bytes up to a maximum of 31. The key schedule is (in the case of 128-bit keys) identical to the Rijndael key schedule.

On each message to be processed, the cipher is initialized by taking the key (more precisely, the expanded key), interpreting the IV as a cleartext block, and applying a (slightly modified) five round Rijndael encryption with block length

256. The resulting cipher text block is loaded into R^0 and R^1. Finally, D_8 is initialized to equal the table T_8, the Rijndael S-box, and S is set to zero.

Output is produced 4 bytes at a time. To this end, the two LFSRs are first irregularly clocked, determined by S. Eight bytes, selected from R^0 and R^1, are run through the permutation D_8 to produce the four output bytes. Selected entries in D_8 are swapped. Finally, S and R^0 are modified in preparation for the next output cycle. Entries in R^1 are not modified apart from the LFSR stepping.

2.1 The output cycle

After each update of the cipher's internal state, four bytes are output. Before the first output byte, and between consecutive output pairs of bytes, a state update function is performed as specified below.

Next state function Let $\ell_0 = 7$ and $\ell_1 = 9$ be the lengths of the registers. Register R^i is stepped $2 + (\lfloor S/2^{14+i} \rfloor \bmod 2)$ steps with a sparse feedback where each step consists of

- set $f^i \leftarrow \theta^i R^i_{j^i} + \mu^i R^i_0$ for constants θ^i, j^i, and μ^i, where $j^0 = 1$ and $j^1 = 5$
- set $R^i_j \leftarrow R^i_{j+1}$ for $j = 0, 1, \ldots, \ell_i - 2$
- feedback $R^i_{\ell_i - 1} \leftarrow f^i$.

After stepping both R^0 and R^1 above, do the following steps, first for $i = 0$, then repeat them for $i = 1$:

- Write $(R^i_{\ell_i - 1}, R^i_{\ell_i - 2})$ as four bytes $\alpha^i_0 || \alpha^i_1 || \alpha^i_2 || \alpha^i_3$.
- Let $\beta^i_j = D_8(\alpha^i_j)$ for $j = 0, 1, 2, 3$.
- Swap elements in D_8 by $D_8(\alpha^i_0) \leftrightarrow D_8(\alpha^i_2)$ and $D_8(\alpha^i_1) \leftrightarrow D_8(\alpha^i_3)$.

Next, update S and R^0

- Update S according to $S \leftarrow S +_{16} \beta^1_0 || \beta^1_1$.
- Update R^0 according to $R^0_5 \leftarrow R^0_5 +_{16} \beta^1_2 || \beta^1_3$.

At this point, the internal state is updated, and the output is formed from the above (β^0_j, β^1_j)-pairs as described next.

Output generation Form four output bytes $b_0 || b_1 || b_2 || b_3$ where

$$b_j = \beta^0_j \oplus \beta^1_j.$$

If more output bytes are required, the output cycle above is repeated. For a more complete description of Polar Bear, see [1].

3 The Attack

In this section we present an effective guess-and-determine attack on Polar Bear requiring only a very small amount of known plaintext. Under the assumption of a certain stepping of the registers, a certain sequence of α-values, and a known plaintext, the state can be recovered in $O(2^{79})$ time. Only knowledge of the first 24 bytes of plaintext is needed.

A first observation of Polar Bear is that it is relatively straightforward that the attack resistance does not meet the key size. For instance, by guessing one (the shorter) LFSR value, it is possible to deduce the value of the other by observing output. Hence, we have an attack with complexity about 2^{112}.

Let the state of LFSR R^i after t steppings of the register be

$$(R^i_{t+\ell_i-1}, R^i_{t+\ell_i-2}, \ldots, R^i_t)$$

where ℓ_i is the length of register R^i. The notation ${}^*R^0_i$ will be used for stages in R^0 after their update.

Let the first 24 bytes of plaintext be known and let the corresponding first twelve 16-bit block of keystream be $Z_0, Z_1, \ldots, Z_{11}$.

For the attack to be successful, three assumptions have to be made.

1. During the first six updates of the state, let the steppings for both LFSR R^0 and R^1 be 2-steppings where the register is stepped two steps. This happens if the fourteenth and fifteenth bit of the word quantity S is 0. Because the word quantity S is initialized to zero the first stepping for both registers is always a 2-stepping. The probability that the six first steppings is 2-steppings can therefore be assumed to be $(1/2)^{10} = 2^{-10}$.
2. Let no pair of the first 8 α be equal. The probability for this is
$$\frac{256!}{(256-8)! \cdot 256^8} \approx 0.90$$
3. Let no pair of the following 40 α be equal. The probability for this is
$$\frac{256!}{(256-40)! \cdot 256^{40}} \approx 0.04$$

Because all the steppings for both the registers are 2-steppings all the stages in both LFSRs are used to generate keystream. The probability that all three of the above assumptions holds is greater than 2^{-15}.

Under these assumptions it suffices to guess the four stages $R^1_9, R^1_{10}, R^1_{11}$ and R^1_{13} (a total of 64 bits) to recover the state. The state can now be recovered with the four equations obtained from the feedback polynomials, the output function and the nonlinear update of R^0.

$$R^0_i = \theta^0 \cdot^{(*)} R^0_{i-6} + \mu^0 \cdot^{(*)} R^0_{i-7} \tag{1}$$
$$R^1_i = \theta^1 \cdot R^1_{i-4} + \mu^1 \cdot R^1_{i-9} \tag{2}$$
$$Z_i = \Delta(R^0_{i+7}) + \Delta(R^1_{i+9}) \tag{3}$$
$${}^*R^0_i = R^0_i +_{16} R^1_{i+2} \tag{4}$$

All operations in (1)–(3) are in the finite field $\mathbb{F}_{2^{16}}$, whereas the $+_{16}$ in (4) is addition modulo 2^{16}. The constants are the ones from the feedback polynomials and the function $\Delta(x)$ is obtained by looking up the two bytes of x in $D8$ and then concatenate them.

From $R_9^1, R_{10}^1, R_{11}^1, R_{13}^1$ and (3) we get $R_7^0, R_8^0, R_9^0, R_{11}^0$. With a knowledge of the four stages R_7^0, R_8^0, R_9^1 and R_{10}^1 we can calculate how $D8$ will be permutated after the first update of the inner state. Let the result of this permutation be $D8'$. As the next 32 α-values are all different we can treat $D8$ as a constant equal to $D8'$ during the next five updates of the state. The rest of the stages in the registers can now be determined in the following order.
(Where R_i, (3) $\rightarrow R_j$ should be read as R_i and (3) gives R_j)

$$
\begin{array}{rcl}
R_7^0,\ R_9^0,\ R_{11}^0,\ R_9^1,\ R_{11}^1,\ R_{13}^1,\ (4) & \rightarrow & {}^*R_7^0,\ {}^*R_9^0,\ {}^*R_{11}^0 \\
{}^*R_7^0,\ R_8^0,\ {}^*R_9^0,\ (1) & \rightarrow & R_{14}^0,\ R_{15}^0 \\
R_{14}^0,\ R_{15}^0,\ (3) & \rightarrow & R_{16}^1,\ R_{17}^1 \\
R_{15}^0,\ R_{17}^1,\ (4) & \rightarrow & {}^*R_{15}^0 \\
R_{11}^1,\ R_{16}^1,\ (2) & \rightarrow & R_{20}^1 \\
R_{20}^1,\ (3) & \rightarrow & R_{18}^0 \\
{}^*R_{11}^0,\ R_{18}^0,\ (1) & \rightarrow & R_{12}^0 \\
R_{12}^0,\ (3) & \rightarrow & R_{14}^1 \\
R_9^1,\ R_{14}^1,\ (2) & \rightarrow & R_{18}^1 \\
R_{18}^1,\ (3) & \rightarrow & R_{16}^0 \\
{}^*R_9^0,\ R_{16}^0,\ (1) & \rightarrow & R_{10}^0 \\
R_{10}^0,\ (3) & \rightarrow & R_{12}^1 \\
R_{10}^0,\ {}^*R_{11}^0,\ (3) & \rightarrow & R_{17}^0 \\
R_{17}^0,\ (3) & \rightarrow & R_{19}^1 \\
R_{17}^0,\ R_{19}^1,\ (4) & \rightarrow & {}^*R_{17}^0 \\
R_{10}^1,\ R_{19}^1,\ (2) & \rightarrow & R_{15}^1 \\
R_{15}^1,\ (3),\ (4) & \rightarrow & R_{13}^0,\ {}^*R_{13}^0
\end{array}
$$

From $R_9^0, \ldots, R_{17}^0$ and starred and unstarred $R_7^0, \ldots, R_{13}^0$ we can determine $D8$ and S which is the whole state. From this can all future keystream be calculated.

To try all possible values for the 4 register stages takes $O(2^{64})$ time and the probability that such an attack is successful is 2^{-15}. The time complexity for the above attack is therefore $O(2^{79})$.

Hasanzadeh *et al* [2] have lowered the time complexity in a recently presented paper. By using the same attack principle, but with a more careful analysis and selection of 'guessed' values, they reach an overall attack complexity of $O(2^{57.4})$

4 Analysis and update of Polar Bear

There are several unfortunate coincidences that make this attack possible. The most obvious is that the dynamic permutation of bytes $D8$ is not permutated and therefore known initially. Other reasons are the short length of the LFSRs, the use of feedback trinoms, the choice of nonlinear updating of R^0, and that register stages are too related.

We propose that the security is enhanced by adding a key-dependent premixing of the D8 table in conjunction with the key schedule. We propose that three full rounds of mixing of D8 is used to this end:

1. Expand the key to 768 bytes of expanded key
2. For $i = 0$ to 767
   Swap($D8[i \pmod{256}]$, $D8[\text{key}[i]]$)

This will only affect the performance of the key schedule. As far as we have been able to tell, no other change is needed.

Optimization We have been able to optimize the reference code submitted with Polar Bear from 38 cycles/byte on a Pentium M to under 23 cycles/byte. By making a small tweak that change how the permutation of the dynamic permutation $D8$ is done, the code can be optimized further. Instead of reading all β-values and then make the swaps, two β-values is read, the corresponding $D8$-values are swapped and then the process is repeated for the other two β-values. This makes Polar Bear faster than AES-CTR.

5 Conclusion

The original specification of Polar Bear apparently has weaknesses, but this can easily be fixed with small changes to the algorithm. By making the permutation of $D8$ in the key setup we only lose performance when a new key is exchanged. This is tolerable as the time for key setup is seldom critical as the same key is typically used with a large number of different IVs, and time for key setup is usually small compared to the time used to generate and exchange a new key.

References

1. Johan Håstad and Mats Näslund. *The Stream Cipher Polar Bear*. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/021. 2005. http://www.ecrypt.eu.org/stream.
2. Mahdi Hasanzadeh, Elham Shakour and Shahram Khazaei. *Improved Cryptanalysis of Polar Bear*. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/084. 2006. http://www.ecrypt.eu.org/stream.

Improved Cryptanalysis of Polar Bear

Mahdi M. Hasanzadeh Elham Shakour Shahram Khazaei

Zaeim Electronic Industries Company, P.O. BOX 14155-1434, Tehran, Iran
{Hasanzadeh, shakour, Khazaei}@zaeim.com

Abstract. In this paper we propose a Guess-and-Determine based initial state recovery attack on Polar Bear, one of the ECRYPT stream cipher project candidates, which is an improvement of the recently proposed one by J. Mattsson with computational complexity of $O(2^{79})$. The computational complexity and success probability of our attack are $O(2^{31})$ and $2^{-26.4}$ respectively which can also be considered as one with computational complexity of $O(2^{57.4})$.
Keywords. Stream Cipher, Guess-and-Determine Attack, Polar Bear, ECRYPT, Security Evaluation.

1 Introduction

Stream ciphers are widely used for fast encryption of sensitive data. Lots of old stream ciphers that have been formerly used can no longer be considered secure, because of their vulnerability to newly developed cryptanalysis techniques. In particular, the NESSIE project [6] did not select any of the proposed stream ciphers for its portfolio, as it was felt that none of the submissions was sufficiently strong. In order to create a portfolio of secure stream ciphers, the ECRYPT project [1] made a call for designs of new stream ciphers which led to submission of 35 proposals to the project by April 2005.

Polar Bear [2] is one of the ECRYPT stream cipher project candidates. The cipher was designed for software applications and dealing with keys of up to 128 bits length. John Mattsson recently found a weakness on the cipher which lead to an initial state recovery attack on it with computational complexity of $O(2^{79})$ according to his note [4]. The detail of this attack has not been published yet but it is going to appear in SASC 2006 [5]. In this paper we improve Mattsson's results and propose an attack with computational complexity of $O(2^{57.4})$. Our Analysis is a Guess-and-Determine based initial state recovery attack whose computational complexity and success probability are $O(2^{31})$ and $2^{-26.4}$ respectively which can also be considered as one with computational complexity of $O(2^{57.4})$.

The paper is organized as follows. In Section 2 a brief description of the key-stream generator of Polar Bear is given. The details of our attack are presented in Section 3 and, finally, the paper is concluded in Section 4.

2 Outline of Polar Bear

Polar Bear [2] works with 16-bit words and uses a 7-word LFSR R^0 and a 9-word LFSR R^1. These are viewed as acting over $GF(2^{16})$. Besides these registers, the internal state of the cipher also depends on a word quantity, S, and a dynamic permutation of bytes, D_8. The cipher deals with

keys of up to 128 bits length. The IV can be any number of bytes up to a maximum of 31. The initial states of R^0 and R^1 are determined through a certain key-IV set up, D_8 is initialized to the table T_8, the Rijndael S-box, and S is set to zero. The cipher produces two words at each cycle of operation. At each cycle, firstly, the two LFSRs are irregularly clocked according to S. Then, two words from each of R^0 and R^1 are selected and nonlinearly filtered using the permutation D_8 to produce two output words. Afterwards, some selected entries in D_8 are swapped. Finally, S and one word of R^0 are modified in preparation for the next cycle.

Let || denote concatenation of 16-bit words as well as 8-bit bytes. Moreover, let $\oplus$ and $+_{16}$ respectively denote bitwise XOR and addition modulo 2^{16} of 16-bit words. A complete description of Polar Bear can be given by the following pseudo-code.

1. Using the initialization process, determine the values of $(R_6^0, R_5^0, R_4^0, R_3^0, R_2^0, R_1^0, R_0^0)$ and $(R_8^1, R_7^1, R_6^1, R_5^1, R_4^1, R_3^1, R_2^1, R_1^1, R_0^1)$.
2. S $\leftarrow 0$, $D_8 \leftarrow T_8$.
3. For t = 1 to N/2 do (N is the required number of output words):
   - 3.1. $b_0 \leftarrow 2 + (\lfloor S / 2^{14} \rfloor \bmod 2)$, $b_1 \leftarrow 2 + (\lfloor S / 2^{15} \rfloor \bmod 2)$.
   - 3.2. Clock R^0 and R^1 LFSRs b_0 and b_1 times, respectively.
   - 3.3. $\alpha_0^0 \| \alpha_1^0 \| \alpha_2^0 \| \alpha_3^0 \leftarrow R_6^0 \| R_5^0$.
   - 3.4. $\beta_0^0 \| \beta_1^0 \| \beta_2^0 \| \beta_3^0 \leftarrow D_8(\alpha_0^0) \| D_8(\alpha_1^0) \| D_8(\alpha_2^0) \| D_8(\alpha_3^0)$.
   - 3.5. $\left(D_8(\alpha_0^0), D_8(\alpha_1^0), D_8(\alpha_2^0), D_8(\alpha_3^0)\right) \leftarrow \left(\beta_2^0, \beta_3^0, \beta_0^0, \beta_1^0\right)^*$.
   - 3.6. $\alpha_0^1 \| \alpha_1^1 \| \alpha_2^1 \| \alpha_3^1 \leftarrow R_8^1 \| R_7^1$.
   - 3.7. $\beta_0^1 \| \beta_1^1 \| \beta_2^1 \| \beta_3^1 \leftarrow D_8(\alpha_0^1) \| D_8(\alpha_1^1) \| D_8(\alpha_2^1) \| D_8(\alpha_3^1)$.
   - 3.8. $\left(D_8(\alpha_0^1), D_8(\alpha_1^1), D_8(\alpha_2^1), D_8(\alpha_3^1)\right) \leftarrow \left(\beta_2^1, \beta_3^1, \beta_0^1, \beta_1^1\right)^*$.
   - 3.9. $\gamma_0^1 \| \gamma_1^1 \leftarrow \beta_0^1 \| \beta_1^1 \| \beta_2^1 \| \beta_3^1$.
   - 3.10. $\gamma_0^0 \| \gamma_1^0 \leftarrow \beta_0^0 \| \beta_1^0 \| \beta_2^0 \| \beta_3^0$.
   - 3.11. $S \leftarrow S +_{16} \gamma_0^1$.
   - 3.12. $R_5^0 \leftarrow R_5^0 +_{16} \gamma_1^1$.
   - 3.13. $Z_0^t \leftarrow \gamma_0^1 \oplus \gamma_0^0$, $Z_1^t \leftarrow \gamma_1^1 \oplus \gamma_1^0$.

* These two lines of the pseudo-code are slightly different from those on the original specification of Polar Bear; refer to [3] for more details.

The sequence $\{Z_0^1, Z_1^1, Z_0^2, Z_1^2, \cdots, Z_0^{N/2}, Z_1^{N/2}\}$ is the output sequence of the cipher. The feedback polynomials of the registers are primitive over GF(2^{16}) and given by $\mu^0 x^7 + \theta^0 x^6 - 1 = 0$ and $\mu^1 x^9 + \theta^1 x^4 - 1 = 0$ in accordance with the recursive equations $R_{n+7}^0 = \theta^0 \cdot R_{n+1}^0 + \mu^0 \cdot R_n^0$ and $R_{n+9}^1 = \theta^1 \cdot R_{n+5}^1 + \mu^1 \cdot R_n^1$ for the output sequences of R^0 and R^1 LFSRs, respectively. Here '+' and '$\cdot$' respectively denote addition and multiplication operations of the finite field GF(2^{16}). Refer to [2] for more details on the cipher and definition of the finite field GF(2^{16}). In the rest of this paper we drop the multiplication operation symbol for simplicity.

3 Description of the Attack

In this section we present our attack on Polar Bear which is an improvement of one recently proposed by John Mattsson [4]. Both attacks use known plain-text scenario and recover the initial states of registers.

Mattsson's attack recovers the initial states of the registers under the assumption that in the first six cycles both registers are clocked two steps and all the values of α_j^i's, totally 48 values, are different. Under these conditions D_8 is known and is equal to T_8 for those entries used in the first six cycles.

Let $(R_{n+8}^1, R_{n+7}^1, R_{n+6}^1, R_{n+5}^1, R_{n+4}^1, R_{n+3}^1, R_{n+2}^1, R_{n+1}^1, R_n^1)$ be the state of LFSR R^1 after n steps. Mattsson guesses the 64 bits R_9^1, R_{10}^1, R_{11}^1 and R_{13}^1 to recover the unknown initial states of the registers in a Guess-and-Determine manner. According to Mattsson's notes [4], the time complexity of his attack is $O(2^{79})$.

Our attack recovers the initial states of the registers under the assumption that in the first eight cycles, R^0 is clocked two steps in all cycles, the sequence of number of steps for R^1 is {2, 3, 3, 3, 3, 2, 3, 2}, and all the values of α_j^i's, totally 64 values, are different. Under these conditions D_8 is known and is equal to T_8 for those entries used in the first eight cycles. Note since S is initialized to zero the two registers are always clocked twice in the first cycle. Therefore, the probability of validity of the assumed sequences for the number of steps for the registers in the first eight cycles is equal to 2^{-14}. The probability that all the 64 values of α_j^i's in the first eight cycles are different is equal to $256 \times 255 \times \cdots \times 193/256^{64} \approx 2^{-12.4}$. Our attack is a Guess-and-Determine based attack which first guesses the values of R_{18}^1 and R_{19}^1 and then recovers the initial states of the registers with a little effort. The total number of possible values for R_{18}^1 and R_{19}^1 is equal to 2^{31} (see the remark at the end of this section). Therefore, the computational complexity and success probability of our attack are $O(2^{31})$ and $2^{-26.4}$ respectively. One can interpret the attack as one with computational complexity of $O(2^{57.4})$.

Let $(R_{n+8}^1, R_{n+7}^1, R_{n+6}^1, R_{n+5}^1, R_{n+4}^1, R_{n+3}^1, R_{n+2}^1, R_{n+1}^1, R_n^1)$ be the state of the LFSR R^1 after n steps. We denote the state of the register R^0 after n steps by $(R_{n+6}^0, R_{n+5}^0, R_{n+4}^0, R_{n+3}^0, R_{n+2}^0, R_{n+1}^0, R_n^0)$ where R_{n+j}^0 ($0 \le j \le 6$) may have a hat and is replaced by $\hat{R}_{n+j}^0$. We use a hat for R_{n+j}^0 if it is a shifted value of the cell number five of the register R^0 and its value has been nonlinearly updated through the step 3.12 of the pseudo-code. For example, since the registers are clocked twice at the first cycle, the state of the register R^0 will be $(R_8^0, \hat{R}_7^0, R_6^0, R_5^0, R_4^0, R_3^0, R_2^0)$ after the first cycle. After the second cycle, the state of R^0 will be $(R_{10}^0, \hat{R}_9^0, R_8^0, \hat{R}_7^0, R_6^0, R_5^0, R_4^0)$ or $(R_{11}^0, \hat{R}_{10}^0, R_9^0, R_8^0, \hat{R}_7^0, R_6^0, R_5^0)$ if the register R^0 is respectively clocked two or three steps at the second cycle. And so on.

The 8 by 8 S-box T_8 acts on 8-bit bytes. For our convenience we define a 16 by 16 S-box T which acts on 16-bit words by applying T_8 on the two bytes of its input word. To be more precise, if w_1 and w_0 are two arbitrary 8-bit bytes, we have $T(w_0||w_1) = T_8(w_0)||T_8(w_1)$. Using this definition together with the introduced notations for the instantaneous internal state of R^0 and R^1, and taking

into account the assumed clocking way of the registers and the difference assumption of α_j^i's at the first eight cycles of cipher operation, one can easily trace the relations between different parts of the cipher and derive the relations between the internal state variables as well as the relations of output sequence of the cipher. We have derived and summarized these relations in the Table 1. We have not mentioned the relation for swapping the D_8 entries and updating of S.

Table 1. Internal and output relations of the first eight cycles of the cipher operation under our assumptions.

Cycle	R^0 Relations	R^1 Relations	Output Relations	R^0 Nonlinear Update
1	$(1)R_7^0=\theta^0 R_1^0+\mu^0 R_0^0$ $(2)R_8^0=\theta^0 R_2^0+\mu^0 R_1^0$	$(3)R_9^1=\theta^1 R_5^1+\mu^1 R_0^1$ $(4)R_{10}^1=\theta^1 R_6^1+\mu^1 R_1^1$	$(5)T(R_7^0)\oplus T(R_9^1)=Z_1^1$ $(6)T(R_8^0)\oplus T(R_{10}^1)=Z_0^1$	$(7)\hat{R}_7^0=R_7^0+_{16}T(R_9^1)$
2	$(1)R_9^0=\theta^0 R_3^0+\mu^0 R_2^0$ $(2)R_{10}^0=\theta^0 R_4^0+\mu^0 R_3^0$	$(3)R_{11}^1=\theta^1 R_7^1+\mu^1 R_2^1$ $(4)R_{12}^1=\theta^1 R_8^1+\mu^1 R_3^1$ $(5)R_{13}^1=\theta^1 R_9^1+\mu^1 R_4^1$	$(6)T(R_9^0)\oplus T(R_{12}^1)=Z_1^2$ $(7)T(R_{10}^0)\oplus T(R_{13}^1)=Z_0^2$	$(8)\hat{R}_9^0=R_9^0+_{16}T(R_{12}^1)$
3	$(1)R_{11}^0=\theta^0 R_5^0+\mu^0 R_4^0$ $(2)R_{12}^0=\theta^0 R_6^0+\mu^0 R_5^0$	$(3)R_{14}^1=\theta^1 R_{10}^1+\mu^1 R_5^1$ $(4)R_{15}^1=\theta^1 R_{11}^1+\mu^1 R_6^1$ $(5)R_{16}^1=\theta^1 R_{12}^1+\mu^1 R_7^1$	$(6)T(R_{11}^0)\oplus T(R_{15}^1)=Z_1^3$ $(7)T(R_{12}^0)\oplus T(R_{16}^1)=Z_0^3$	$(8)\hat{R}_{11}^0=R_{11}^0+_{16}T(R_{15}^1)$
4	$(1)R_{13}^0=\theta^0 \hat{R}_7^0+\mu^0 R_6^0$ $(2)R_{14}^0=\theta^0 R_8^0+\mu^0 \hat{R}_7^0$	$(3)R_{17}^1=\theta^1 R_{13}^1+\mu^1 R_8^1$ $(4)R_{18}^1=\theta^1 R_{14}^1+\mu^1 R_9^1$ $(5)R_{19}^1=\theta^1 R_{15}^1+\mu^1 R_{10}^1$	$(6)T(R_{13}^0)\oplus T(R_{18}^1)=Z_1^4$ $(7)T(R_{14}^0)\oplus T(R_{19}^1)=Z_0^4$	$(8)\hat{R}_{13}^0=R_{13}^0+_{16}T(R_{18}^1)$
5	$(1)R_{15}^0=\theta^0 \hat{R}_9^0+\mu^0 R_8^0$ $(2)R_{16}^0=\theta^0 R_{10}^0+\mu^0 \hat{R}_9^0$	$(3)R_{20}^1=\theta^1 R_{16}^1+\mu^1 R_{11}^1$ $(4)R_{21}^1=\theta^1 R_{17}^1+\mu^1 R_{12}^1$ $(5)R_{22}^1=\theta^1 R_{18}^1+\mu^1 R_{13}^1$	$(6)T(R_{15}^0)\oplus T(R_{21}^1)=Z_1^5$ $(7)T(R_{16}^0)\oplus T(R_{22}^1)=Z_0^5$	$(8)\hat{R}_{15}^0=R_{15}^0+_{16}T(R_{21}^1)$
6	$(1)R_{17}^0=\theta^0 \hat{R}_{11}^0+\mu^0 R_{10}^0$ $(2)R_{18}^0=\theta^0 R_{12}^0+\mu^0 \hat{R}_{11}^0$	$(3)R_{23}^1=\theta^1 R_{19}^1+\mu^1 R_{14}^1$ $(4)R_{24}^1=\theta^1 R_{20}^1+\mu^1 R_{15}^1$	$(5)T(R_{17}^0)\oplus T(R_{23}^1)=Z_1^6$ $(6)T(R_{18}^0)\oplus T(R_{24}^1)=Z_0^6$	$(7)\hat{R}_{17}^0=R_{17}^0+_{16}T(R_{23}^1)$
7	$(1)R_{19}^0=\theta^0 \hat{R}_{13}^0+\mu^0 R_{12}^0$ $(2)R_{20}^0=\theta^0 R_{14}^0+\mu^0 \hat{R}_{13}^0$	$(3)R_{25}^1=\theta^1 R_{21}^1+\mu^1 R_{16}^1$ $(4)R_{26}^1=\theta^1 R_{22}^1+\mu^1 R_{17}^1$ $(5)R_{27}^1=\theta^1 R_{23}^1+\mu^1 R_{18}^1$	$(6)T(R_{19}^0)\oplus T(R_{26}^1)=Z_1^7$ $(7)T(R_{20}^0)\oplus T(R_{27}^1)=Z_0^7$	$(8)\hat{R}_{19}^0=R_{19}^0+_{16}T(R_{26}^1)$
8	$(1)R_{21}^0=\theta^0 \hat{R}_{15}^0+\mu^0 R_{14}^0$ $(2)R_{22}^0=\theta^0 R_{16}^0+\mu^0 \hat{R}_{15}^0$	$(3)R_{28}^1=\theta^1 R_{24}^1+\mu^1 R_{19}^1$ $(4)R_{29}^1=\theta^1 R_{25}^1+\mu^1 R_{20}^1$	$(5)T(R_{21}^0)\oplus T(R_{28}^1)=Z_1^8$ $(6)T(R_{22}^0)\oplus T(R_{29}^1)=Z_0^8$	$(7)\hat{R}_{21}^0=R_{21}^0+_{16}T(R_{28}^1)$

All the relations of Table 1 are invertible in all the input variables. In other words, if we know all the input variables except one for each equation, the unknown variable is uniquely determined. Such kinds of equations are suitable to be solved in a Guess-and-Determine manner. In a Guess-

and-Determine attack, we first guess some variables and then try to recover the remaining variables efficiently. The less the space size of the guessed variables is, the less the computational complexity is required. The validity of a guess is determined using some additional check equations.

It is easy to show that it is not possible to uniquely solve the system of equations of Table 1 by guessing less than two variables. Moreover, guessing the values of R_{18}^1 and R_{19}^1 reveals the initial state of the registers, that is $(R_6^0, R_5^0, R_4^0, R_3^0, R_2^0, R_1^0, R_0^0)$ and $(R_8^1, R_7^1, R_6^1, R_5^1, R_4^1, R_3^1, R_2^1, R_1^1, R_0^1)$ which are our desires. We have summarized the steps which lead to recovering the initial states of the registers in Table 2.

Each step of Table 2 states which equation from Table 1 must be used to determine one of the variables using previously determined variables. For example, at 20th step the variable R_{10}^0 is determined using equation 1 at cycle 6 of the Table 1 because R_{17}^0 and $\hat{R}_{11}^0$ have already been determined at the 7th and 19th steps respectively. More precisely we have $R_{10}^0 = (R_{17}^0 - \theta^0 \hat{R}_{11}^0) / \mu^0$ where '–' and '/ ' are the subtraction and division operations of the finite field GF(2^{16}).

The correct initial state can be find by running the cipher some cycles and comparing the resulting output sequence with the given key-stream sequence.

Remark on the total number of possible values for R_{18}^1 : Although R_{18}^1 is an 16-bit word, under the assumed clocking way for the registers, there are only 2^{15} possibilities for it. Indeed, let S_4 and S_5 be the contents of S at the end of 4th and 5th cycles. We have $S_5 = S_4 +_{16} T(R_{18}^1)$. Since we have assumed that R^1 and R^0 have respectively clocked three times and twice at both the 4th and the 5th cycles, the two most significant bits of both S_4 and S_5 are 10. This proofs that the two most significant bits of $T(R_{18}^1)$ can be either 00 or 11 which shows the existence of 2^{15} possible choices for R_{18}^1.

4 Conclusion

In this paper we proposed a Guess-and-Determine based initial state recovery attack whose computational complexity and success probability are O(2^{31}) and $2^{-26.4}$ respectively. Our attack can be considered as one with computational complexity of O($2^{57.4}$) which is much better than one recently proposed by Mattsson with computational complexity of O(2^{79}). The weakness, which enables these attacks, can effectively be countered by initializing the dynamic permutation D_8 to an 8 by 8 key-IV dependent S-box provided that it seems random to an attacker. In [5] a remedy for fixing the attack has been proposed.

Acknowledgment. We would like to thank Mr. Mattsson for notifying us of a few typos on the paper.

Table 2. The details of the procedure of recovering the initial state of the registers, by guessing R_{18}^1 and R_{19}^1.

Step	Known Words	(Cycle-Relation)	Deduced Word
1	R_{18}^1	(4-6)	R_{13}^0
2	R_{13}^0, R_{18}^1	(4-8)	$\hat{R}_{13}^0$
3	R_{19}^1	(4-7)	R_{14}^0
4	$\hat{R}_{13}^0, R_{14}^0$	(7-2)	R_{20}^0
5	R_{20}^0	(7-7)	R_{27}^1
6	R_{27}^1, R_{18}^1	(7-5)	R_{23}^1
7	R_{23}^1	(6-5)	R_{17}^0
8	R_{23}^1, R_{19}^1	(6-3)	R_{14}^1
9	R_{14}^1, R_{18}^1	(4-4)	R_9^1
10	R_9^1	(1-5)	R_7^0
11	R_7^0, R_9^1	(1-7)	$\hat{R}_7^0$
12	$\hat{R}_7^0, R_{14}^0$	(4-2)	R_8^0
13	$\hat{R}_7^0, R_{13}^0$	(4-1)	R_6^0
14	R_8^0	(1-6)	R_{10}^1
15	R_{10}^1, R_{14}^1	(3-3)	R_5^1
16	R_5^1, R_9^1	(1-3)	R_0^1
17	R_{10}^1, R_{19}^1	(4-5)	R_{15}^1
18	R_{15}^1	(3-6)	R_{11}^0
19	R_{11}^0, R_{15}^1	(3-8)	$\hat{R}_{11}^0$
20	$R_{17}^0, \hat{R}_{11}^0$	(6-1)	R_{10}^0
21	R_{10}^0	(2-7)	R_{13}^1
22	R_{13}^1, R_9^1	(2-5)	R_4^1
23	R_{13}^1, R_{18}^1	(5-5)	R_{22}^1
24	R_{22}^1	(5-7)	R_{16}^0
25	R_{16}^0, R_{10}^0	(5-2)	$\hat{R}_9^0$
26	$\hat{R}_9^0, R_8^0$	(5-1)	R_{15}^0
27	R_{15}^0	(5-6)	R_{21}^1
28	R_{15}^0, R_{21}^1	(5-8)	$\hat{R}_{15}^0$
29	$\hat{R}_{15}^0$, R_{16}^0	(8-2)	R_{22}^0
30	R_{22}^0	(8-6)	R_{29}^1
31	$\hat{R}_{15}^0$, R_{14}^0	(8-1)	R_{21}^0
32	R_{21}^0	(8-5)	R_{28}^1
33	R_{28}^1, R_{19}^1	(8-3)	R_{24}^1
34	R_{24}^1, R_{15}^1	(6-4)	R_{20}^1
35	R_{20}^1, R_{29}^1	(8-4)	R_{25}^1
36	R_{25}^1, R_{21}^1	(7-3)	R_{16}^1
37	R_{16}^1, R_{20}^1	(5-3)	R_{11}^1
38	R_{16}^1	(3-7)	R_{12}^0
39	R_{12}^0, R_6^0	(3-2)	R_5^0
40	R_5^0, R_{11}^0	(3-1)	R_4^0
41	R_{10}^0, R_4^0	(2-2)	R_3^0
42	$\hat{R}_{13}^0, R_{12}^0$	(7-1)	R_{19}^0
43	R_{19}^0	(7-6)	R_{26}^1
44	R_{26}^1, R_{22}^1	(7-4)	R_{17}^1
45	R_{17}^1, R_{21}^1	(5-4)	R_{12}^1
46	R_{12}^1	(2-6)	R_9^0
47	R_9^0, R_3^0	(2-1)	R_2^0
48	R_2^0, R_8^0	(1-2)	R_1^0
49	R_1^0, R_7^0	(1-1)	R_0^0
50	R_{17}^1, R_{13}^1	(4-3)	R_8^1
51	R_{16}^1, R_{12}^1	(3-5)	R_7^1
52	R_{15}^1, R_{11}^1	(3-4)	R_6^1
53	R_{12}^1, R_8^1	(2-4)	R_3^1
54	R_{11}^1, R_7^1	(2-3)	R_2^1
55	R_{10}^1, R_6^1	(1-4)	R_1^1

References

1. eSTREAM, the ECRYPT Stream Cipher Project (2005) http://www.ecrypt.eu.org/stream/.
2. Håstad J. and Näslund M., The Stream Cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/021 (2005) http://www.ecrypt.eu.org/stream/.
3. Näslund M., Typos in Polar Bear code and spec. eSTREAM, ECRYPT Stream Cipher Project, Discussion Forum (2005) http://www.ecrypt.eu.org/stream/phorum/read.php?1,161/.
4. Mattsson J., Weakness in Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Discussion Forum (2005) http://www.ecrypt.eu.org/stream/phorum/read.php?1,219/.
5. Mattsson J., A Guess-and-Determine Attack on the Stream Cipher Polar Bear. State of Art of Stream Ciphers (SASC'06), Feb. 2006, Leuven, Belgium.
6. NESSIE: New European Schemes for Signature, Integrity and Encryption, http://www.nessie.eu.org/nessie/.

Linear Distinguishing Attack on NLS

Joo Yeon Cho and Josef Pieprzyk

Centre for Advanced Computing – Algorithms and Cryptography,
Department of Computing,
Macquarie University,
NSW, Australia, 2109
{jcho,josef}@ics.mq.edu.au

Abstract. We present a distinguishing attack on NLS which is one of the stream ciphers submitted to the eSTREAM project. We build the distinguisher by using linear approximations of both the non-linear feedback shift register (NFSR) and the non-linear filter function (NLF). Since the bias of the distinguisher depends on the $Konst$ value, which is a key-dependent word, we estimate the average bias to be around $O(2^{-34})$. Therefore, we claim that NLS is distinguishable from truly random cipher after observing $O(2^{68})$ keystream words on the average. In addition, we present how to reduce a fraction of $Konst$ values for which our attack fails.
Keywords : Distinguishing Attacks, Stream Ciphers, Linear Approximations, eSTREAM, Modular Addition, NLS.

1 Introduction

The European Network of Excellence in Cryptology (ECRYPT) launched a stream cipher project called eSTREAM [1] whose aim is to come up with a collection of stream ciphers that can be recommended to industry and government institutions as secure and efficient cryptographic primitives. It is also likely that some or perhaps all recommended stream ciphers may be considered as de facto industry standards. It is interesting to see a variety of different approaches used by the designers of the stream ciphers submitted to the eSTREAM call. A traditional approach for building stream ciphers is to use a linear feedback shift register (LFSR) as the main engine of the cipher. The outputs of the registers are taken and put into a nonlinear filter that produces the output stream that is added to the stream of plaintext.

One of the new trends in the design of stream ciphers is to replace LFSR by a nonlinear feedback shift register (NFSR). From the ciphers submitted to the eSTREAM call, there are several ciphers that use the structure based on NFSR amongst them the NLS cipher follows this design approach. The designers of the NLS cipher are Gregory Rose, Philip Hawkes, Michael Paddon and Miriam Wiggers de Vries from Qualcomm Australia.

The paper studies the NLS cipher and its resistance against distinguishing attacks using linear approximation. Typically, distinguishing attacks do not allow to recover any secret element of the cipher such as the cryptographic key or the secret initial state of the NFSR but instead they permit to tell apart the cipher from the truly random cipher. In this sense these attacks are relatively weak. However, the existence of a distinguishing attack is considered as an early warning sign of possible major security flaws.

In our analysis, we derive linear approximations of both NFSR and the nonlinear filter (NLF). The main challenge has been to combine the obtained linear approximations in a such way that the internal state bits of NFSR have been eliminated leaving the observable

output bits only. Our approach is an extension of the linear masking method introduced by Coppersmith, Halevi, and Jutla in [3]. Note that the linear masking method was applied for the traditional stream ciphers based on LFSR so it is not directly applicable for the ciphers with NFSR.

The work is structured as follows. Section 2 briefly describes the NLS cipher. In Section 3, we study best linear approximations for both NFSR and NLF. A simplified NLS cipher is defined in Section 4 and we show how to design a distinguisher for it. Our distinguisher for the original NLS cipher is examined in Section 5. We show how it works and also discuss its limitations. Section 6 concludes our work.

2 Brief description of NLS stream cipher

As we said the NLS keystream generator uses NFSR whose outputs are given to the nonlinear filter NLF that produces output keystream bits. Note that we concentrate on the cipher itself and ignore its message integrity function as irrelevant to our analysis. For details of the cipher, the reader is referred to [2].

NLS has two components: NFSR and NLF whose work is synchronised by a clock. The state of NFSR at time t is denoted by $\sigma_t = (r_t[0], \ldots, r_t[16])$ where $r_t[i]$ is a 32-bit word. The state is determined by 17 words (or equivalently 544 bits). The transition from the state σ_t to the state σ_{t+1} is defined as follows:

1. $r_{t+1}[i] = r_t[i+1]$ for $i = 0, \ldots, 15$;
2. $r_{t+1}[16] = f((r_t[0] \lll 19) \boxplus (r_t[15] \lll 9) \boxplus Konst) \oplus r_t[4]$;
3. if $t = 0$ (modulo f16), $r_{t+1}[2] = r_{t+1}[2] \boxplus t$;

where $f16$ is 65537 and $\boxplus$ is the addition modulo 2^{32}. The *Konst* value is a 32-bit key-dependent constant. The function $f : \{0,1\}^{32} \to \{0,1\}^{32}$ is constructed using an S-box with 8-bit input and 32-bit output and defined as $f(a) = \text{S-box}(a_H) \oplus a$ where a_H is the most significant 8 bits of 32-bit word a. Each output keystream word ν_t of NLF is obtained as

$$\nu_t = NLF(\sigma_t) = (r_t[0] \boxplus r_t[16]) \oplus (r_t[1] \boxplus r_t[13]) \oplus (r_t[6] \boxplus Konst). \quad (1)$$

The cipher uses 32-bit words to ensure a fast keystream generation.

3 Analysis of NFSR and NLF

Unlike a LFSR that applies a connection polynomial, the NFSR uses a much more complex nonlinear transition function f that mixes the XOR addition (linear) with the addition modulo 2^{32} (nonlinear). According to the structure of the non-linear shift register, the following equation holds for the least significant bit. Let us denote α_t to be a 32-bit output of the S-box that defines the transition function f. Then, we observe that for the least significant bit, the following equation holds

$$\alpha_{t,(0)} \oplus r_t[0]_{(13)} \oplus r_t[15]_{(23)} \oplus Konst_{(0)} \oplus r_t[4]_{(0)} \oplus r_{t+1}[16]_{(0)} = 0 \quad (2)$$

where $\alpha_{t,(0)}$ and $x_{(i)}$ stand for the i-th bits of the 32-bit words α_t and x, respectively.

To make our analysis simpler we assume initially that *Konst* is zero. This assumption is later dropped (i.e. *Konst* is non-zero) when we discuss our distinguishing attack on the NLS stream cipher.

3.1 Linear approximations of $\alpha_{t,(0)}$

Recall that α_t is the 32-bit output taken from the S-box and $\alpha_{t,(0)}$ is its least significant bit. The input to the S-box comes from the eight most significant bits of the addition $((r_t[0] \lll 19) \boxplus (r_t[15] \lll 9) \boxplus Konst)$. Assuming that $Konst$=0, the input to S-box is $(r_t[0]' \boxplus r_t[15]')$, where $r_t[0]' = r_t[0] \lll 19$ and $r_t[15]' = r_t[15] \lll 9$. Thus, $\alpha_{t,(0)}$ is completely determined by the contents of two registers $r_t[0]'$ and $r_t[15]'$. Observe that the input of the S-box is affected by the eight most significant bits of the two registers $r_t[0]'$ (we denote the 8 most significant bits of the register by $r_t[0]'_{(H)}$) and $r_t[15]'$ (the 8 most significant bits of the register are denoted by $r_t[15]'_{(H)}$) and by the carry bit c generated by the addition of two 24 least significant bits of $r_t[0]'$ and $r_t[15]'$. Therefore

$$\text{the input of the S-box} = r_t[0]'_{(H)} \boxplus r_t[15]'_{(H)} \boxplus c.$$

Now we would like to find the best linear approximation for $\alpha_{t,(0)}$. We build the truth table with 2^{17} rows and 2^{16} columns. Each row corresponds to the unique collection of input variables (8 bits of $r_t[0]'_{(H)}$, 8 bits of $r_t[15]'_{(H)}$, and a single bit for c). Each column relates to the unique linear combination of bits from $r_t[0]'_{(H)}$ and $r_t[15]'_{(H)}$. Table 1 displays a collection of best linear approximations that are going to be used in our distinguishing attack. In particular, the third row of Table 1 has relatively high bias. This seems to be caused by the reason that $r_t[0]_{(12)} \oplus r_t[15]_{(22)}$ is the only input to the MSB of input of the S-box that is not diffused to other order bits. Note that $r_t[0]'_{(H)} = (r_t[0] \lll 19)_{(H)} = (r_t[0]_{(12)}, \ldots, r_t[0]_{(5)})$

linear approximations of $\alpha_{t,(0)}$	bias
$r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_t[15]_{(20)} \oplus r_t[15]_{(16)} \oplus r_t[15]_{(15)}$	1/2+0.024414
$r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_t[0]_{(5)} \oplus r_t[15]_{(20)} \oplus r_t[15]_{(16)}$	1/2+0.024414
$r_t[0]_{(12)} \oplus r_t[15]_{(22)}$	1/2-0.022705
$r_t[0]_{(11)} \oplus r_t[15]_{(21)}$	1/2+0.002441
$r_t[0]_{(10)} \oplus r_t[15]_{(20)}$	1/2-0.017578

Table 1. Linear approximations for $\alpha_{t,(0)}$ when $Konst = 0$

and $r_t[15]'_{(H)} = (r_t[15] \lll 9)_{(H)} = (r_t[15]_{(22)}, \ldots, r_t[15]_{(15)})$. Note also that none of the approximations contains the carry bit c, in other words, the approximations do not depend on c.

3.2 Linear approximations for NFSR

Having a linear approximation of $\alpha_{t,(0)}$, it is easy to obtain a linear approximation for NFSR. Let us choose the first approximation from Table 1, so we are getting the following linear equation:

$$\alpha_{t,(0)} = r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_t[15]_{(20)} \oplus r_t[15]_{(16)} \oplus r_t[15]_{(15)} \tag{3}$$

with the bias $0.024414 = 2^{-5.35}$. Now we combine Equations (2) and (3) and as the result we have the following approximation for NFSR

$$\begin{aligned} &r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_t[15]_{(20)} \oplus r_t[15]_{(16)} \oplus r_t[15]_{(15)} \\ &\oplus r_t[0]_{(13)} \oplus r_t[15]_{(23)} \oplus Konst_{(0)} \oplus r_t[4]_{(0)} \oplus r_{t+1}[16]_{(0)} = 0 \end{aligned} \tag{4}$$

with the bias of $2^{-5.35}$.

3.3 Linear approximation for NLF

Recall that Equation (1) defines the output keystream generated by NLF. As we have assumed that $Konst$ is zero, we get

$$\nu_t = (r_t[0] \boxplus r_t[16]) \oplus (r_t[1] \boxplus r_t[13]) \oplus r_t[6]$$

Let us take a closer look at the addition $\boxplus$, we know that the least significant bits are linear so the following equation holds $(r[x] \boxplus r[y])_{(0)} = r[x]_{(0)} \oplus r[y]_{(0)}$. Consequently, we obtain the relation for the least significant bits in the following form

$$\nu_{t,(0)} = (r_t[0]_{(0)} \oplus r_t[16]_{(0)}) \oplus (r_t[1]_{(0)} \oplus r_t[13]_{(0)}) \oplus (r_t[6]_{(0)}) \tag{5}$$

that holds with probability one.
All consecutive bits $i > 0$ of $\boxplus$ are nonlinear. Consider the function $(r[x] \boxplus r[y])_{(i)} \oplus (r[x] \boxplus r[y])_{(i-1)}$. The function has a linear approximation as follows

$$(r[x] \boxplus r[y])_{(i)} \oplus (r[x] \boxplus r[y])_{(i-1)} = r[x]_{(i)} \oplus r[y]_{(i)} \oplus r[x]_{(i-1)} \oplus r[y]_{(i-1)} \tag{6}$$

that has the bias of 2^{-2}. Using the above approximation we can argue that, for $2 \leq i \leq 31$, NLF function possesses a linear approximation of the following form

$$\begin{aligned} \nu_{t,(i)} \oplus \nu_{t,(i-1)} = {} & (r_t[0]_{(i)} \oplus r_t[16]_{(i)} \oplus r_t[0]_{(i-1)} \oplus r_t[16]_{(i-1)}) \\ & \oplus (r_t[1]_{(i)} \oplus r_t[13]_{(i)} \oplus r_t[1]_{(i-1)} \oplus r_t[13]_{(i-1)}) \\ & \oplus (r_t[6]_{(i)} \oplus r_t[6]_{(i-1)}) \end{aligned} \tag{7}$$

with the bias of $2(2^{-2})^2 = 2^{-3}$.

4 Distinguishing attack on a simplified NLS

In this section we assume that the structure of NFSR is unchanged but the structure of NLF is modified by replacing the addition $\boxplus$ by $\oplus$. Thus, Equation (1) that describes the keystream becomes

$$\mu_t = (r_t[0] \oplus r_t[16]) \oplus (r_t[1] \oplus r_t[13]) \oplus (r_t[6] \oplus Konst). \tag{8}$$

This linear function is valid for 32-bit words so it can be equivalently re-written as a system of 32 equations each equation valid for the particular ith bit. Hence, for $0 \leq i \leq 31$,

$$\mu_{t,(i)} = (r_t[0]_{(i)} \oplus r_t[16]_{(i)}) \oplus (r_t[1]_{(i)} \oplus r_t[13]_{(i)}) \oplus (r_t[6]_{(i)} \oplus Konst_{(i)}). \tag{9}$$

To build a distinguisher we combine approximations of NFSR given by Equation (4) with linear equations defined by (9). For the clocks t, $t+1$, $t+6$, $t+13$, and $t+16$, consider the following approximations of NFSR

$$\begin{aligned} & r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_t[15]_{(20)} \oplus \cdots \oplus r_{t+1}[16]_{(0)} = 0 \\ & r_{t+1}[0]_{(10)} \oplus r_{t+1}[0]_{(6)} \oplus r_{t+1}[15]_{(20)} \oplus \cdots \oplus r_{t+2}[16]_{(0)} = 0 \\ & r_{t+6}[0]_{(10)} \oplus r_{t+6}[0]_{(6)} \oplus r_{t+6}[15]_{(20)} \oplus \cdots \oplus r_{t+7}[16]_{(0)} = 0 \\ & r_{t+13}[0]_{(10)} \oplus r_{t+13}[0]_{(6)} \oplus r_{t+13}[15]_{(20)} \oplus \cdots \oplus r_{t+14}[16]_{(0)} = 0 \\ & r_{t+16}[0]_{(10)} \oplus r_{t+16}[0]_{(6)} \oplus r_{t+16}[15]_{(20)} \oplus \cdots \oplus r_{t+17}[16]_{(0)} = 0 \end{aligned} \tag{10}$$

Since $r_{t+p}[0] = r_t[p]$, we can rewrite the above system of equations (10) equivalently as follows:

$$\begin{array}{l} r_t[0]_{(10)} \oplus r_t[0]_{(6)} \oplus r_{t+15}[0]_{(20)} \oplus \cdots \oplus r_{t+17}[0]_{(0)} = 0 \\ r_t[1]_{(10)} \oplus r_t[1]_{(6)} \oplus r_{t+15}[1]_{(20)} \oplus \cdots \oplus r_{t+17}[1]_{(0)} = 0 \\ r_t[6]_{(10)} \oplus r_t[6]_{(6)} \oplus r_{t+15}[6]_{(20)} \oplus \cdots \oplus r_{t+17}[6]_{(0)} = 0 \\ r_t[13]_{(10)} \oplus r_t[13]_{(6)} \oplus r_{t+15}[13]_{(20)} \oplus \cdots \oplus r_{t+17}[13]_{(0)} = 0 \\ r_t[16]_{(10)} \oplus r_t[16]_{(6)} \oplus r_{t+15}[16]_{(20)} \oplus \cdots \oplus r_{t+17}[16]_{(0)} = 0 \end{array} \tag{11}$$

Consider the columns of the above system of equations. Each column describes a single bit output of the filter (see Equation (9)), therefore the system (11) gives the following approximation:

$$\begin{array}{l} \mu_{t,(10)} \oplus \mu_{t,(6)} \oplus \mu_{t+15,(20)} \oplus \mu_{t+15,(16)} \oplus \mu_{t+15,(15)} \oplus \mu_{t,(13)} \\ \oplus \mu_{t+15,(23)} \oplus \mu_{t+4,(0)} \oplus \mu_{t+17,(0)} = K \end{array} \tag{12}$$

where $K = Konst_{(10)} \oplus Konst_{(6)} \oplus Konst_{(20)} \oplus Konst_{(16)} \oplus Konst_{(15)} \oplus Konst_{(13)} \oplus Konst_{(23)}$. Note that the bit K is constant (zero or one) during the session. Therefore, by the piling-up lemma, the bias of (12) is $\mathbf{2} \cdot \mathbf{2^4} \cdot (\mathbf{2^{-5.35}})^{\mathbf{5}} = \mathbf{2^{-22}}$.

5 Distinguishing attack on NLS

In this Section, we describe a distinguishing attack on the real NLS. The main idea is to find the best combination of approximations for both NFSR and NLF, while the state bits of the shift register vanish and the bias of the resulting approximation is as big as possible. We study the case for $Konst = 0$ at first and then, extend our attack to the case for $Konst \neq 0$. Note that only a non-zero most significant byte of $Konst$ is allowed in NLS cipher.

5.1 Case for $Konst = 0$

The linear approximations of $\alpha_{t,(0)}$ are given in Table 1. We choose this time the third approximation from the table so

$$\alpha_{t,(0)} = r_t[0]_{(12)} \oplus r_t[15]_{(22)} \tag{13}$$

and the bias of this approximation is $0.022705 = 2^{-5.46}$. By combining Equations (2) and (13), we have the following approximation

$$r_t[0]_{(12)} \oplus r_t[15]_{(22)} \oplus r_t[0]_{(13)} \oplus r_t[15]_{(23)} \oplus r_t[4]_{(0)} \oplus r_{t+1}[16]_{(0)} = 0 \tag{14}$$

that has the same bias. Let us now divide (14) into two parts : the least significant bit and the other bits, so we get

$$\begin{array}{l} l_1(r_t) = r_t[4]_{(0)} \oplus r_{t+1}[16]_{(0)} \\ l_2(r_t) = r_t[0]_{(12)} \oplus r_t[0]_{(13)} \oplus r_t[15]_{(22)} \oplus r_t[15]_{(23)} \end{array} \tag{15}$$

Clearly, $l_1(r_t) \oplus l_2(r_t) = 0$ with the bias $2^{-5.46}$. Since $l_1(r_t)$ has only the least significant bit variables, we apply (5) which is true with probability one. Then, we obtain

$$\begin{array}{l} l_1(r_t) = r_t[4]_{(0)} \oplus r_{t+1}[16]_{(0)} \\ l_1(r_{t+1}) = r_{t+1}[4]_{(0)} \oplus r_{t+2}[16]_{(0)} \\ l_1(r_{t+6}) = r_{t+6}[4]_{(0)} \oplus r_{t+7}[16]_{(0)} \\ l_1(r_{t+13}) = r_{t+13}[4]_{(0)} \oplus r_{t+14}[16]_{(0)} \\ l_1(r_{t+16}) = r_{t+16}[4]_{(0)} \oplus r_{t+17}[16]_{(0)} \end{array} \tag{16}$$

If we add up all approximations of (16), then, by applying Equation (5), we can write

$$l_1(r_t) \oplus l_1(r_{t+1}) \oplus l_1(r_{t+6}) \oplus l_1(r_{t+13}) \oplus l_1(r_{t+16}) = \nu_{t+4,(0)} \oplus \nu_{t+17,(0)} \tag{17}$$

Now, we focus on $l_2(r_t)$ where the bit positions are 12, 13, 22, and 23 so

$$\begin{array}{l} l_2(r_t) = r_t[0]_{(12)} \oplus r_t[0]_{(13)} \oplus r_t[15]_{(22)} \oplus r_t[15]_{(23)} \\ l_2(r_{t+1}) = r_{t+1}[0]_{(12)} \oplus r_{t+1}[0]_{(13)} \oplus r_{t+1}[15]_{(22)} \oplus r_{t+1}[15]_{(23)} \\ l_2(r_{t+6}) = r_{t+6}[0]_{(12)} \oplus r_{t+6}[0]_{(13)} \oplus r_{t+6}[15]_{(22)} \oplus r_{t+6}[15]_{(23)} \\ l_2(r_{t+13}) = r_{t+13}[0]_{(12)} \oplus r_{t+13}[0]_{(13)} \oplus r_{t+13}[15]_{(22)} \oplus r_{t+13}[15]_{(23)} \\ l_2(r_{t+16}) = r_{t+16}[0]_{(12)} \oplus r_{t+16}[0]_{(13)} \oplus r_{t+16}[15]_{(22)} \oplus r_{t+16}[15]_{(23)} \end{array} \tag{18}$$

Since $r_{t+p}[0] = r_t[p]$, the above approximations can be presented as follows

$$\begin{array}{l} l_2(r_t) = r_t[0]_{(12)} \oplus r_t[0]_{(13)} \oplus r_{t+15}[0]_{(22)} \oplus r_{t+15}[0]_{(23)} \\ l_2(r_{t+1}) = r_t[1]_{(12)} \oplus r_t[1]_{(13)} \oplus r_{t+15}[1]_{(22)} \oplus r_{t+15}[1]_{(23)} \\ l_2(r_{t+6}) = r_t[6]_{(12)} \oplus r_t[6]_{(13)} \oplus r_{t+15}[6]_{(22)} \oplus r_{t+15}[6]_{(23)} \\ l_2(r_{t+13}) = r_t[13]_{(12)} \oplus r_t[13]_{(13)} \oplus r_{t+15}[13]_{(22)} \oplus r_{t+15}[13]_{(23)} \\ l_2(r_{t+16}) = r_t[16]_{(12)} \oplus r_t[16]_{(13)} \oplus r_{t+15}[16]_{(22)} \oplus r_{t+15}[16]_{(23)} \end{array} \tag{19}$$

Recall the approximation (7) of NLF. If we combine (19) with (7), then we have

$$\begin{array}{l} l_2(r_t) \oplus l_2(r_{t+1}) \oplus l_2(r_{t+6}) \oplus l_2(r_{t+13}) \oplus l_2(r_{t+16}) = \\ \nu_{t,(12)} \oplus \nu_{t,(13)} \oplus \nu_{t+15,(22)} \oplus \nu_{t+15,(23)} \end{array} \tag{20}$$

By combining the approximations (17) and (20), we obtain the final approximation that defines our distinguisher, i.e.

$$\begin{array}{l} l_1(r_t) \oplus l_1(r_{t+1}) \oplus l_1(r_{t+6}) \oplus l_1(r_{t+13}) \oplus l_1(r_{t+16}) \\ \oplus l_2(r_t) \oplus l_2(r_{t+1}) \oplus l_2(r_{t+6}) \oplus l_2(r_{t+13}) \oplus l_2(r_{t+16}) \\ = \nu_{t,(12)} \oplus \nu_{t,(13)} \oplus \nu_{t+15,(22)} \oplus \nu_{t+15,(23)} \oplus \nu_{t+4,(0)} \oplus \nu_{t+17,(0)} \\ = 0 \end{array} \tag{21}$$

The second part of the approximation can be computed from the output keystream that can be observed by the adversary. The bias can be computed using the piling-up lemma. As we use the approximation (14) five times and the approximation (7) twice, the bias of the approximation (21) is $\mathbf{2} \cdot (\mathbf{2^4}(\mathbf{2^{-5.46}})^{\mathbf{5}}) \cdot (\mathbf{2}(\mathbf{2^{-3}})^{\mathbf{2}}) = \mathbf{2^{-27.3}}$.

5.2 Case for $Konst \neq 0$

Recall that $Konst$ takes part in the input of NFSR and NLF. If $Konst$ is not zero, then, the biases of linear approximations for $\alpha_{t,(0)}$ and NLF are changed according to the values of $Konst$. Let us denote that $Konst_{(H)} = (Konst_{(31)}, \ldots, Konst_{(24)})$, and $Konst_{(L)} = (Konst_{(23)}, \ldots, Konst_{(0)})$.

Biases of linear approximations of $\alpha_{t,(0)}$ and NLF with $Konst_{(H)}$ Since the most significant 8 bits of $Konst$ contribute to form of the bit $\alpha_{t,(0)}$, the bias of the approximation (13) fluctuates mostly according to the 8-bit $Konst_{(H)}$. This relation is illustrated in Figure 1. From this figure, we can see that (13) has the smallest bias when $Konst_{(H)} = 51$ and 179, even though the bias of (13) is $\mathbf{2^{-6.4}}$ on the average.

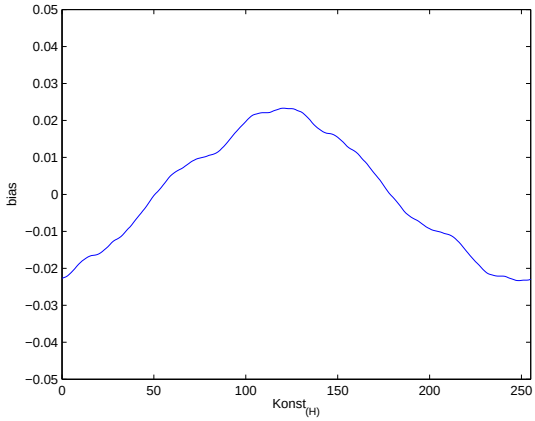


Fig. 1. Bias of $\alpha_{t,(0)} = r_t[0]_{(12)} \oplus r_t[15]_{(22)}$ with $Konst_{(H)}$

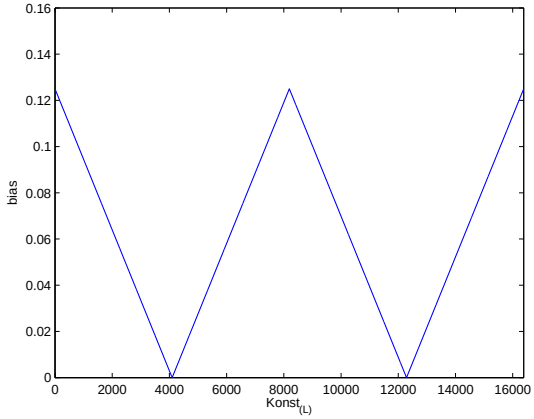


Fig. 2. Bias of (22) with $Konst_{(L)}$ when $i = 13$

$Konst_{(H)}$	best linear approximations of $\alpha_{t,(0)}$	bias
1	$r_t[0]_{(12)} \oplus r_t[15]_{(22)}$	1/2-0.022522
51	$r_t[0]_{(12)} \oplus r_t[0]_{(11)} \oplus r_t[0]_{(10)} r_t[15]_{(22)} \oplus r_t[15]_{(21)} \oplus r_t[15]_{(20)}$	1/2+0.022705
120	$r_t[0]_{(12)} \oplus r_t[15]_{(22)}$	1/2+0.011353
179	$r_t[0]_{(12)} \oplus r_t[0]_{(11)} \oplus r_t[0]_{(10)} \oplus r_t[15]_{(22)} \oplus r_t[15]_{(21)} \oplus r_t[15]_{(20)}$	1/2+0.011353

Table 2. A partial table of best approximations for $\alpha_{t,(0)}$ with $Konst_{(H)}$

Hence, in order to maximize the bias of our distinguisher, we need to find the best approximations for $\alpha_{t,(0)}$ when $Konst_{(H)}$ runs through all possible values, i.e. from 0 to 255. Note that the best approximation of $\alpha_{t,(0)}$ means one which results in maximum bias of distinguisher when the approximation of NLF is combined. Table 2 shows a partial table for approximations of $\alpha_{t,(0)}$. When $Konst_{(H)}$ is around 1 or 120, we use the following approximation for NLF.

$$\begin{aligned}\nu_{t,(i)} \oplus \nu_{t,(i-1)} = &(r_t[0]_{(i)} \oplus r_t[16]_{(i)} \oplus r_t[0]_{(i-1)} \oplus r_t[16]_{(i-1)}) \\ &\oplus(r_t[1]_{(i)} \oplus r_t[13]_{(i)} \oplus r_t[1]_{(i-1)} \oplus r_t[13]_{(i-1)}) \\ &\oplus(r_t[6]_{(i)} \oplus Konst_{(i)} \oplus r_t[6]_{(i-1)} \oplus Konst_{(i-1)})\end{aligned} \tag{22}$$

On the other side, when $Konst_{(H)}$ is around 51 or 179, we use the following approximation:

$$\begin{aligned}\nu_{t,(i)} \oplus \nu_{t,(i-1)} \oplus \nu_{t,(i-2)} \oplus \nu_{t,(i-3)} = \\ &(r_t[0]_{(i)} \oplus r_t[16]_{(i)} \oplus r_t[0]_{(i-1)} \oplus r_t[16]_{(i-1)} \\ &\oplus r_t[0]_{(i-2)} \oplus r_t[16]_{(i-2)} \oplus r_t[0]_{(i-3)} \oplus r_t[16]_{(i-3)}) \\ &\oplus(r_t[1]_{(i)} \oplus r_t[13]_{(i)} \oplus r_t[1]_{(i-1)} \oplus r_t[13]_{(i-1)} \\ &\oplus r_t[1]_{(i-2)} \oplus r_t[13]_{(i-2)} \oplus r_t[1]_{(i-3)} \oplus r_t[13]_{(i-3)}) \\ &\oplus(r_t[6]_{(i)} \oplus Konst_{(i)} \oplus r_t[6]_{(i-1)} \oplus Konst_{(i-1)} \\ &\oplus r_t[6]_{(i-2)} \oplus Konst_{(i-2)} \oplus r_t[6]_{(i-3)} \oplus Konst_{(i-3)})\end{aligned} \tag{23}$$

Instead of Approximation (6), we need the following linear approximation in order to compute the bias of (23),

$$\begin{aligned}&(r[x] \boxplus r[y])_{(i)} \oplus (r[x] \boxplus r[y])_{(i-1)} \oplus (r[x] \boxplus r[y])_{(i-2)} \oplus (r[x] \boxplus r[y])_{(i-3)} = \\ &r[x]_{(i)} \oplus r[y]_{(i)} \oplus r[x]_{(i-1)} \oplus r[y]_{(i-1)} \oplus r[x]_{(i-2)} \oplus r[y]_{(i-2)} \oplus r[x]_{(i-3)} \oplus r[y]_{(i-3)}\end{aligned} \tag{24}$$

that has the bias of 2^{-3}.

Biases of linear approximations of NLF with $Konst_{(L)}$ In Approximation (22), the bias of the following approximation fluctuates depending on $Konst_{(L)}$.

$$(r_t[6] \boxplus Konst)_i \oplus (r_t[6] \boxplus Konst)_{i-1} = (r_t[6]_{(i)} \oplus Konst_{(i)} \oplus r_t[6]_{(i-1)} \oplus Konst_{(i-1)}) \tag{25}$$

Figure 2 displays the bias distribution according to $Konst_{(L)}$ in (22) when $i = 13$. Note that this graph shows the distribution from 14 LSBs of $Konst_{(L)}$ (that is, 2^{14}) since the bits $Konst_{(23)}, \ldots, Konst_{(14)}$ have not effect on the bias for $i = 13$. We should consider 24 bits of $Konst_{(L)}$ when $i = 23$ in (22). However, the distribution graph is similar to Figure 2 with only the slope changed. On the average, the bias of (22) is $\mathbf{2^{-4}}$. A very similar analysis is possible for Approximation (23). The bias of (23) is $\mathbf{2^{-7}}$ on the average.

Average bias of distinguisher From both Approximations (22) and (23) with biases shown in Table 2, we can build two distinguishers as follows.

$$\nu_{t,(12)} \oplus \nu_{t,(13)} \oplus \nu_{t+15,(22)} \oplus \nu_{t+15,(23)} \oplus \nu_{t+4,(0)} \oplus \nu_{t+17,(0)} = 0 \tag{26}$$

$$\begin{aligned}&\nu_{t,(10)} \oplus \nu_{t,(11)} \oplus \nu_{t,(12)} \oplus \nu_{t,(13)} \oplus \nu_{t+15,(20)} \oplus \nu_{t+15,(21)} \\ &\oplus \nu_{t+15,(22)} \oplus \nu_{t+15,(23)} \oplus \nu_{t+4,(0)} \oplus \nu_{t+17,(0)} = 0\end{aligned} \tag{27}$$

The bias of best distinguisher for each $Konst_{(H)}$ is displayed in Table 3. We take the average biases of Approximations (22) and (23).

$Konst_{(H)}$	distinguisher	details	bias	data complexity
1	(26)	$2 \cdot (2^4(2^{-5.47})^5) \cdot (2(2^{-4})^2)$	$2^{-29.4}$	$2^{58.8}$
51	(27)	$2 \cdot (2^4(2^{-6.46})^5) \cdot (2(2^{-7})^2)$	$2^{-40.3}$	$2^{80.6}$
120	(26)	$2 \cdot (2^4(2^{-5.42})^5) \cdot (2(2^{-4})^2)$	$2^{-29.1}$	$2^{58.2}$
179	(27)	$2 \cdot (2^4(2^{-6.46})^5) \cdot (2(2^{-7})^2)$	$2^{-40.3}$	$2^{80.6}$

Table 3. A partial table of biases for distinguisher with $Konst_{(H)}$

If we select the distinguisher (26), then, the average bias of approximation of $\alpha_{t,(0)}$ over $Konst_{(H)}$ is $2^{-6.4}$. Therefore, the bias of distinguisher appears to be around $\mathbf{2} \cdot (\mathbf{2^4}(\mathbf{2^{-6.4}})^{\mathbf{5}}) \cdot (\mathbf{2}(\mathbf{2^{-4}})^{\mathbf{2}}) = \mathbf{2^{-34}}$ on the average. Note that an adversary should avoid the keystream that is produced around clock $t = 0 \pmod{65537}$ as the feedback has an additional step at this clock. (See Step 3 at Section 2)

For some values of $Konst_{(H)}$ (e.g. $Konst_{(H)} = 51$ or 179), the bias of the distinguisher (26) becomes less than 2^{-40}. In order to compensate this "small-bias" area, an adversary observes the distinguisher (26) and (27) simultaneously in such a way that a bigger bias among those are always chosen. Note that the amount of the keystream for both distinguishers is not increased since the keystream is produced by words.

Therefore, the minimum bias observable by both distinguisher (26) and (27) will be $2^{-40.3}$ even $Konst_{(H)}$ is close to 51 or 179.

5.3 When does our distinguishing attack fail?

Let us denote the bias of the approximation of $\alpha_{t,(0)}$ by ϵ_1, the bias of the approximation of a single addition (for example, Approximations (6) and (24)) by ϵ_2 and the bias of the approximation of $(r_t[6] \boxplus Konst)$ by ϵ_3. Since the specification of the NLS cipher allows the adversary to observe up to 2^{80} keystream words per one key/nonce pair, we assume that our attack is not successful if the bias of distinguisher satisfies the following condition:

$$\left.\begin{array}{l}\text{bias of linear approx. of } \alpha_{t,(0)} \ : d_1 = 2^4(\epsilon_1^5) \\ \text{bias linear approx. of NLF} : d_2 = 2^2(\epsilon_2)^2\epsilon_3\end{array}\right\} \Rightarrow 2 \cdot d_1 \cdot 2 \cdot (d_2)^2 < 2^{-40}. \quad (28)$$

Note that ϵ_1 is affected by $Konst_{(H)}$, and ϵ_3 by $Konst_{(L)}$.

When the bias becomes zero? In Figure 2, the bias of (25) becomes zero when

1. $Konst_{(L)} = (b_{31}, \ldots, b_{23}, 1, 0, \ldots, 0)$
2. $Konst_{(L)} = (b_{31}, \ldots, b_{13}, 1, 0, \ldots, 0)$

where b_i can have any bit (0 or 1). Hence, the bias of this distinguisher is zero for around 2^{19} out of 2^{32} possible values of $Konst$.

5.4 Multiple distinguishers

Since the NLS produces 32-bit keystream words per a clock, we may reduce the unsuccessful portion of $Konst$ by considering multiple distinguishers without increasing the necessary volume of observed data. For example, let us consider the following approximation of $\alpha_{t,(0)}$

$$\alpha_{t,(0)} = r_t[0]_{(11)} \oplus r_t[15]_{(21)} \quad (29)$$

whose bias on the average is around $0.012911 = 2^{-6.28}$. The corresponding approximation of NLF is

$$\begin{aligned}\nu_{t,(i)} \oplus \nu_{t,(i-2)} = &(r_t[0]_{(i)} \oplus r_t[16]_{(i)} \oplus r_t[0]_{(i-2)} \oplus r_t[16]_{(i-2)}) \\ &\oplus (r_t[1]_{(i)} \oplus r_t[13]_{(i)} \oplus r_t[1]_{(i-2)} \oplus r_t[13]_{(i-2)}) \\ &\oplus (r_t[6]_{(i)} \oplus Konst_{(i)} \oplus r_t[6]_{(i-2)} \oplus Konst_{(i-2)})\end{aligned} \tag{30}$$

and the bias of this approximation is around $2^2(2^{-3})^3 = 2^{-7}$.

As in Section 5.1, an another distinguisher based on a different relation can be built. The relation is as follows:

$$\nu_{t,(11)} \oplus \nu_{t,(13)} \oplus \nu_{t+15,(21)} \oplus \nu_{t+15,(23)}\nu_{t+4,(0)} \oplus \nu_{t+17,(0)} = 0 \tag{31}$$

The bias for the distinguisher on the average is around $2 \cdot (2^4(2^{-6.28})^5) \cdot (2(2^{-7})^2) = 2^{-39.4}$. It is a known fact that the bias of this distinguisher also fluctuates depending on the actual value of $Konst$. However, this time, the phase of fluctuation has been shifted from that of the distinguisher (26).

Even though our attack based on the distinguisher (26) fails for some values of $Konst$, it may be still successful by observing the bias of the other distinguisher (31). Note that the number of observations of keystream required for multiple distinguishers remains same as for a single distinguisher.

We intend to investigate our attack in more detail, in particular, we would like to determine the fraction of the values of $Konst$ for which the distinguishing attack works.

6 Conclusion

In this paper, we presented a linear distinguishing attack on NLS. The bias of distinguisher appears to be 2^{-34} on the average so that NLS is distinguishable from a random function by observing 2^{68} keystream words. Even though there are a fraction of $Konst$ which requires the data complexity bigger than 2^{80}, we show that it is possible for attacker to reduce the fraction of $Konst$ by combining multiple distinguishers which have biases of less than 2^{-40} on the average.

Acknowledgment We are grateful to Philip Hawkes and anonymous referees of SASC 2006 for their very helpful comments. The second author acknowledges the support received from Australian Research Council (projects DP0451484 and DP0663452).

References

1. http://www.ecrypt.eu.org/stream/.
2. http://www.ecrypt.eu.org/stream/nls.html.
3. Don Coppersmith, Shai Halevi, and Charanjit Jutla. Cryptanalysis of stream ciphers with linear masking. Cryptology ePrint Archive, Report 2002/020, 2002. http://eprint.iacr.org/.

Cryptanalysis of Grain

Côme Berbain[1], Henri Gilbert[1], and Alexander Maximov[2]

[1] France Telecom Research and Development
38-40 rue du Général Leclerc, 92794 Issy-les-Moulineaux, France
[2] Dept. of Information Technology, Lund University, Sweden
P.O. Box 118, 221 00 Lund, Sweden
{come.berbain, henri.gilbert}@francetelecom.com
movax@it.lth.se

Abstract. Grain [11] is a lightweight stream cipher submitted by M. Hell, T. Johansson, and W. Meier to the eSTREAM call for stream cipher proposals of the European project ECRYPT [5]. Its 160-bit internal state is divided into a LFSR and an NLFSR of length 80 bits each. A filtering boolean function is used to derive each keystream bit from the internal state. By combining linear approximations of the feedback function of the NLFSR and of the filtering function, it is possible to derive linear approximation equations involving the keystream and the LFSR initial state. We present a key recovery attack against Grain which requires 2^{43} computations and 2^{38} keystream bits to determine the 80-bit key.

Keywords: Stream cipher, Correlation attack, Walsh transform

1 Introduction

Stream ciphers are symmetric encryption algorithms based on the concept of pseudo-random keystream generator. In the typical case of a binary additive stream cipher, the key and an additional parameter named initialization vector (IV) are used to generate a binary sequence called keystream which is bitwise combined with the plaintext to provide the ciphertext. Although it seems rather difficult to construct a very fast and secure stream cipher, some efforts to achieve this have recently been deployed. The NESSIE project [24] launched in 1999 by the European Union did not succeed in selecting a secure enough stream cipher. Recently, the European Network of Excellence in Cryptology ECRYPT launched a call for stream cipher proposals named eSTREAM [5]. The candidate stream ciphers were submitted in May 2005. Those candidates are divided into software oriented and hardware oriented ciphers.

Hardware oriented stream ciphers are specially designed so that their implementation requires a very small number of gates. Such ciphers are useful in mobile systems, e.g. mobile phones or RFID, where minimizing the number of gates and power consumption is more important than very high speed.

[0] The work described in this paper has been supported in part by Grant VR 621-2001-2149, in part by the French Ministry of Research RNRT X-CRYPT project and in part by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT.

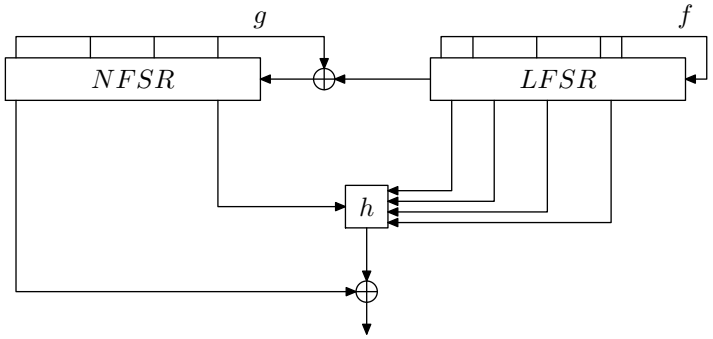
One of the new hardware candidates submitted to eSTREAM is a stream cipher named Grain [11] which was developed by M. Hell, T. Johansson, and W. Meier [3] as an alternative to stream ciphers like GSM A5/1 or Bluetooth E_0. It uses a 80-bit key and a 64-bit initialization vector to fill in an internal state of size 160 bits divided into a *nonlinear feedback shift register* (NLFSR) and a *linear feedback shift register* (LFSR) of length 80 bits each. At each clock pulse, one keystream bit is produced by selecting some bits of the LFSR and of the NLFSR and applying a boolean function. It is well known that LFSR sequences satisfy several statistical properties one would expect from a random sequence, but do not offer any security. Their combination with NLFSR sequences is expected to improve the security. However, NLFSR based constructions have not yet been as well studied as LFSR based constructions. The claimed security level of Grain is 2^{80}, and it was conjectured by the authors of Grain that there exists no attack significantly faster than exhaustive search.

In this paper, we describe two key recovery attacks against Grain. The proposed attacks exploit linear approximations of the output function. The first one requires 2^{55} operations, 2^{49} bits of memory, and 2^{51} keystream bits, and the second one requires 2^{43} operations, 2^{42} bits of memory, and 2^{38} keystream bits.

This paper is organized as follows. We first describe the Grain stream cipher (Section 2) and we derive some linear approximations involving the LFSR and the keystream (Section 3). We then present two techniques for recovering the initial state of the LFSR (Section 4). Finally, we present a technique allowing to recover the initial state of the NLFSR once we know the LFSR initial state (Section 5).

2 Description of Grain

Grain [11] is based upon three main building blocks: an 80-bit linear feedback shift register, an 80-bit nonlinear feedback shift register, and a nonlinear filtering function. Grain is initialized with the 80-bit key K and the 64-bit initialization value IV. The cipher output is an L-bit keystream sequence $(z_t)_{t=0,\ldots,L-1}$.



The current LFSR content is denoted by $Y^t = (y_t, y_{t+1}, \ldots, y_{t+79})$. The LFSR is governed by the linear recurrence:

$$y_{t+80} = y_{t+62} \oplus y_{t+51} \oplus y_{t+38} \oplus y_{t+23} \oplus y_{t+13} \oplus y_t.$$

[3] The design of Grain was also submitted and recently accepted for publication in *the International Journal of Wireless and Mobile Computing, Special Issue on Security of Computer Network and Mobile Systems.*

The current NFSR content is denoted by $X^t = (x_t, x_{t+1}, \ldots, x_{t+79})$. The NFSR feedback is disturbed by the output of the LFSR, so that the NFSR content is governed by the recurrence:

$$x_{t+80} = y_t \oplus g(x_t, x_{t+1}, \ldots, x_{t+79}),$$

where the expression of nonlinear feedback function g is given by

$$\begin{aligned}
&x_{t+63} \oplus x_{t+60} \oplus x_{t+52} \oplus x_{t+45} \oplus x_{t+37} \oplus x_{t+33} \oplus x_{t+28} \oplus x_{t+21} \oplus x_{t+15} \oplus x_{t+9} \oplus x_t \\
&\oplus x_{t+63}x_{t+60} \oplus x_{t+37}x_{t+33} \oplus x_{t+15}x_{t+9} \oplus x_{t+60}x_{t+52}x_{t+45} \oplus x_{t+33}x_{t+28}x_{t+21} \\
&\oplus x_{t+63}x_{t+45}x_{t+28}x_{t+9} \oplus x_{t+60}x_{t+52}x_{t+37}x_{t+33} \oplus x_{t+63}x_{t+60}x_{t+21}x_{t+15} \\
&\oplus x_{t+63}x_{t+60}x_{t+52}x_{t+45}x_{t+37} \oplus x_{t+33}x_{t+28}x_{t+21}x_{t+15}x_{t+9} \\
&\oplus x_{t+52}x_{t+45}x_{t+37}x_{t+33}x_{t+28}x_{t+21}.
\end{aligned}$$

The cipher output bit z_t is derived from the current LFSR and NFSR states as the exclusive or of the masking bit x_t and a nonlinear filtering function h as follows:

$$\begin{aligned}
z_t &= x_t \oplus h(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}, x_{t+63}) \\
&= h'(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}, x_t, x_{t+63}) \\
&= x_t \oplus x_{t+63}p_t \oplus q_t,
\end{aligned}$$

where p_t and q_t are the functions of $y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}$ given by:

$$\begin{aligned}
p_t &= 1 \oplus y_{t+64} \oplus y_{t+46}(y_{t+3} \oplus y_{t+25} \oplus y_{t+64}), \\
q_t &= y_{t+25} \oplus y_{t+3}y_{t+46}(y_{t+25} \oplus y_{t+64}) \oplus y_{t+64}(y_{t+3} \oplus y_{t+46}).
\end{aligned}$$

The boolean function h is correlation immune of the first order. As noticed in [11], "this does not preclude that there are correlations of the output of $h(x)$ to sums of inputs", but the designers of Grain appear to have expected the NFSR masking bit x_t to make it impractical to exploit such correlations.

The key and IV setup consists of loading the key bits in the NFSR, loading the 64-bit IV followed by 16 ones in the LFSR, and clocking the cipher 160 times in a special mode where the output bit is fed back into the LFSR and the NFSR. Once the key and IV have been loaded, the keystream generation mode described above is activated and the keystream sequence (z_t) is produced.

3 Deriving Linear Approximations of the LFSR Bits

3.1 Linear Approximations Used to Derive the LFSR Bits

The purpose of the attack is, based on a keystream sequence $(z_t)_{t=0\ldots L-1}$ corresponding to an unknown key K and a known IV value, to recover the key K. The initial step of the attack is to derive a sufficient number N of linear approximation equations involving the $n = 80$ bits of the initial LFSR state $Y^0 = (y_0, \ldots, y_{79})$ (or equivalently a sufficient number N of linear approximation equations involving bits of the sequence (y_t)) to recover the value of Y^0. Hereafter, as will be shown in Section 5, the initial NFSR state X^0 and the key K can then be easily recovered.

The starting point for the attack consists in noticing that though the NFSR feedback function g is balanced, the function g' given by $g'(X^t) = g(X^t) \oplus x_t$ is unbalanced. We have:

$$Pr\{g'(X^t) = 1\} = \frac{522}{1024} = \frac{1}{2} + \epsilon_{g'},$$

where $\epsilon_{g'} = \frac{5}{512}$. It is useful to notice that the restriction of g' to input values X^t such that $x_{t+63} = 0$ is totally balanced and that the imbalance of the function g' is exclusively due to the imbalance of the restriction of g' to input values X^t such that $x_{t+63} = 1$.

If one considers one single output bit z_t, the involvement of the masking bit x_t in the expression of z_t makes it impossible to write any useful approximate relation involving only the Y^t bits. But if one considers the sum $z_t \oplus z_{t+80}$ of two keystream bits output at a time interval equal to the NFSR length $n = 80$, the $x_t \oplus x_{t+80}$ contribution of the corresponding masking bits is equal to $g'(X^t) \oplus y_t$, and is therefore equal to y_t with probability $\frac{1}{2} + \epsilon_{g'}$. As for the other terms of $z_t \oplus z_{t+80}$, they can be approximated by linear functions of the bits of the sequence (y_t). In more details:

$$\begin{aligned} z_t \oplus z_{t+80} = g'(X^t) \oplus y_t \oplus h(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}, x_{t+63}) \\ \oplus h(y_{t+83}, y_{t+105}, y_{t+126}, y_{t+144}, x_{t+143}). \end{aligned}$$

Since the restriction of $g'(X^t)$ to input values such that $x_{t+63} = 0$ is balanced, we can restrict our search to linear approximations of the term $h(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}, x_{t+63})$ to input values such that $x_{t+63} = 1$, which amounts to finding linear approximations of $p_t \oplus q_t$.

We found a set of two best linear approximations for this function, namely:

$$L_1 = \{\ y_3 \oplus y_{25} \oplus y_{64} \oplus 1; y_{25} \oplus y_{46} \oplus y_{64} \oplus 1\}.$$

Each of the approximations of L_1 is valid with a probability $\frac{1}{2} + \epsilon_1$, where $\epsilon_1 = \frac{1}{4}$.

Now the term $h(y_{t+83}, y_{t+105}, y_{t+126}, y_{t+144}, x_{t+143})$ is equal to either $p_{t+80} \oplus q_{t+80}$ or q_{t+80}, with a probability $\frac{1}{2}$ for both expressions. We found a set of 8 best simultaneous linear approximations for these two expressions, namely:

$$\begin{aligned} L_2 = \{\ & y_{t+83} \oplus y_{t+144} \oplus 1; \\ & y_{t+83} \oplus y_{t+126} \oplus y_{t+144}; \\ & y_{t+83} \oplus y_{t+105}; \\ & y_{t+83} \oplus y_{t+105} \oplus y_{t+126}; \\ & y_{t+83} \oplus y_{t+105} \oplus y_{t+126} \oplus y_{t+144} \oplus 1; \\ & y_{t+83} \oplus y_{t+105} \oplus y_{t+144} \oplus 1; \\ & y_{t+105} \oplus y_{t+144}; \\ & y_{t+105} \oplus y_{t+126} \oplus y_{t+144} \oplus 1\}. \end{aligned}$$

Each of the 8 approximations of L_2 has an average probability $\epsilon_2 = \frac{1}{8}$ of being valid.

Thus, we have found 16 linear approximations of $z_t \oplus z_{t+80}$, namely all the linear expressions of the form

$$y_t \oplus l_1(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}) \oplus l_2(y_{t+83}, y_{t+105}, y_{t+126}, y_{t+144}),$$

where $l_1 \in L_1$ and $l_2 \in L_2$. Each of these approximations is valid with a probability $\frac{1}{2} + \epsilon$, where ϵ is derived from $\epsilon_{g'}$, ϵ_1, and ϵ_2 using the Piling-up Lemma:

$$\epsilon = \frac{1}{2} \cdot 2^2 \cdot \epsilon_{g'} \cdot \epsilon_1 \cdot \epsilon_2 = \frac{5}{4096} \simeq 2^{-9.67}.$$

The extra multiplicative factor of $\frac{1}{2}$ takes into account the fact that the considered approximations are only valid when $x_{t+63} = 1$. The LFSR derivation attacks of Section 4 exploit these 16 linear approximations.

3.2 Generalisation of the Attack Method

In this Section, we try to generalise the previous approximation method. The purpose is not to find better approximations than those identified in Section 3.1, but to derive some design criteria on the boolean functions g and h'. However in the previous approximation, we used the fact that the bias of g depends on the value of x_{t+63}, so that the approximations of g and h' are not correct independently. We do not take this phenomenon into account in this Section. Therefore, we only provide a simplified picture of potential generalised attacks.

The function $g(X^t, Y^t)$ operates on $w(g) = w_L(g) + w_N(g)$ variables taken from the LFSR and the NFSR, where $w_L(g)$ is the number of variables taken from the LFSR and $w_N(g)$ the number of variables taken from the NFSR. Let the function $A_g(X^t, Y^t)$ be a linear approximation of the function g, i.e.

$$A_g(X^t, Y^t) = \bigoplus_{i=0}^{w_N(g)-1} d_i x_{t+\phi_g(i)} \oplus \bigoplus_{j=0}^{w_L(g)-1} c_j y_{t+\psi_g(j)}, \quad c_j, d_i \in \mathbb{F}_2, \tag{1}$$

such that the distance between $g(\cdot)$ and $A_g(\cdot)$ defined by:

$$d_g = \#\{x \in \mathbb{F}_2^{w(g)} : A_g(x) \neq g(x)\} > 0,$$

is strictly larger than zero. Then, we have

$$\Pr\{A_g(x) \neq g(x)\} = \frac{1}{2^{w(g)}} d_g,$$

i.e.

$$\Pr\{A_g(x) + g(x) = 0\} = 1/2 + \epsilon_g,$$

where the bias is:

$$\epsilon_g = 1/2 - 2^{-w(g)} d_g.$$

Similarly, the function $h'(X^t, Y^t)$ can also be approximated by some linear expressions of the form:

$$A_{h'}(X^t, Y^t) = \bigoplus_{i=0}^{w_N(h')-1} k_i x_{t+\phi_{h'}(i)} \oplus \bigoplus_{j=0}^{w_L(h')-1} l_j y_{t+\psi_{h'}(j)}, \quad k_j, l_i \in \mathbb{F}_2. \tag{2}$$

Recall, $z_t \stackrel{p}{=} A_{h'}(\cdot)_t$ with some probability p. Having the expressions (1) and (2), one can sum up together $w_N(A_g(\cdot))$ expressions of $A_{h'}(\cdot)$ at different times t, in such a way

that all terms X^t will be eliminated (just because the terms X^t will be cancelled due to the parity check function $A_g(\cdot)$, leaving the terms Y^t and noise variables only). Note also that any linear combination of $A_{h'}(\cdot)$ is a linear combination of the keystream bits z_t.

The sum of $w_N(A_g(\cdot))$ approximations $A_{h'}(\cdot)$ will introduce $w_N(A_g(\cdot))$ independent noise variables due to the approximation at different time instances. Moreover, the cancellation of the terms X^t in the sum will be done by the parity check property of the approximation $A_g(\cdot)$. If the function $A_{h'}(\cdot)$ contains $w_N(A_{h'})$ terms from X^t, then the parity cancellation expression $A_g(\cdot)$ will be applied $w_N(A_{h'})$ times. Each application of the cancellation expression $A_g(\cdot)$ will introduce another noise variable due to the approximation $N_g : g(\cdot) \rightarrow A_g(\cdot)$. Therefore, the application of the expression $A_g(\cdot)$ $w_N(A_{h'})$ times will introduce $w_N(A_{h'})$ additional noise variables N_g. Accumulating all above and following the Piling-up Lemma, the final correlation of such a sum (of the linear expression on Y^t) is given by the following Theorem.

Theorem 1. *There always exists a linear relation in terms of bits from the state of the LFSR and the keystream, which have the bias:*

$$\epsilon = 2^{(w_N(A_{h'})+w_N(A_g)-1)} \cdot \epsilon_g^{w_N(A_{h'})} \cdot \epsilon_{h'}^{w_N(A_g)},$$

where $A_g(\cdot)$ and $A_{h'}(\cdot)$ are linear approximations of the functions $g(\cdot)$ and $h'(\cdot)$, respectively, and:

$$\Pr\{A_g(\cdot) = g(\cdot)\} = 1/2 + \epsilon_g, \qquad \Pr\{A_{h'}(\cdot) = h'(\cdot)\} = 1/2 + \epsilon_{h'}.$$

This theorem gives us a criteria for a proper choice of the functions $g(\cdot)$ and $h'(\cdot)$. The biases ϵ_g and $\epsilon_{h'}$ are related to the *nonlinearity* of these boolean functions, and the values $w_N(A_g)$ and $w_N(A_{h'})$ are related to the *correlation immunity* property; however, there is a well-known trade-off between these two properties [27]. Unfortunately, in the case of Grain the functions $g(\cdot)$ and $h'(\cdot)$ were improperly chosen.

4 Deriving the LFSR Initial State

In the former Section, we have shown how to derive an arbitrary number R of linear approximation equations in the $n = 80$ initial LFSR bits, of bias $\epsilon \simeq 2^{-9.67}$ each, from a sufficient number of keystream bits. Let us denote these equations by:

$$\bigoplus_{i=0}^{n-1} \alpha_i^j \cdot y_i = b^j, j = 1, \ldots, R.$$

In this Section we show how to use these relations to derive the initial LFSR state Y^0. This can be seen as a decoding problem, up to the fact that the code length is not fixed in advance and one has to find an optimal trade-off between the complexities of deriving a codeword (i.e. collecting an appropriate number of linear approximation equations) and decoding this codeword.

An estimate of the number N of linear approximation equations needed for the right value of the unknown to maximize the indicator

$$I = \sharp\left\{ j \in \{1, \ldots, N\} \;\middle|\; \bigoplus_{i=0}^{n-1} \alpha_i^j \cdot y_i = b^j \right\},$$

or at least to be very likely to provide say one of the two or three highest values of I, can be determined as follows.

Under the heuristic assumption that for the correct (resp. incorrect) value of Y^0, I is the sum of N independent binary variables x_i distributed according to the Bernoulli law of parameters $p = Pr\{x_i = 1\} = \frac{1}{2} - \epsilon$ and $q = Pr\{x_i = 0\} = \frac{1}{2} + \epsilon$ (resp. the Bernoulli law of parameters $Pr\{x_i = 1\} = Pr\{x_i = 0\} = \frac{1}{2}$, mean value $\mu = \frac{1}{2}$, and standard deviation $\sigma = \frac{1}{2}$), N can be derived by introducing a threshold of say $T = N(\frac{1}{2} + \frac{3\epsilon}{4})$ for I and requiring: *(i)* that the probability that I is larger than T for an incorrect value of Y^0 is less than a suitably chosen false alarm probability p_{fa}; *(ii)* that the probability that I is lower than T for the correct value is less than a non detection probability p_{nd} of say 1%. For practical values of p_{fa}, the first condition is by far the most demanding. Setting the false alarm rate to $p_{fa} = 2^{-n}$ ensures that the number of false alarms is less than 1 in average.

Due to the Central Limit Theorem, $\frac{\sum x_i - N\mu}{\sqrt{N}\sigma}$ is distributed according to the normal law, so that:

$$Pr\left\{\frac{1}{N}\sum x_i - \mu > \frac{3\epsilon}{4}\right\} = Pr\left\{\frac{\sum x_i - N\mu}{\sqrt{N}\sigma} > \frac{3\sqrt{N}\epsilon}{4\sigma}\right\} \quad (3)$$

can be approximated by $\frac{1}{\sqrt{2\pi}}\int_{\lambda}^{+\infty} e^{-\frac{t^2}{2}}dt$, where $\lambda = \frac{3\sqrt{N}\epsilon}{2}$. Consequently, if N is selected in such a way that $\frac{3\sqrt{N}\epsilon}{2} = \lambda$, i.e.

$$N = \left(\frac{2\lambda}{3\epsilon}\right)^2,$$

where λ is given by:

$$\frac{1}{\sqrt{2\pi}}\int_{\lambda}^{+\infty} e^{-\frac{t^2}{2}}dt = p_{fa} = 2^{-n},$$

then inequality 3 is satisfied.

A naive LFSR derivation method would consist of collecting N approximate equations, computing the indicator I independently for each of the 2^n possible values of Y^0 and retaining those Y^0 candidates leading to a value of I larger than the $N(\frac{1}{2} + \frac{3\epsilon}{4})$ threshold. This method would require a low number of keystream bits (say $\frac{N+80}{16}$) but the resulting complexity $N \cdot 2^{80}$ would be larger than the one of exhaustive key search.

In the rest of this Section, we show that much lower complexities can be obtained by using the fast Walsh transform algorithm and a few extra filtering techniques in order to speed up computations of correlation indicators. Former examples of applications of similar Fast Fourier Transform techniques in order to significantly decrease the total complexity of correlation attacks can be found in [4] [9] [16].

4.1 Use of the Fast Walsh Transform to Speed up Correlation Computations

Basic Method. Let us consider the following problem. Given a sufficient number M of linear approximation equations of bias ϵ involving m binary variables y_0 to y_{m-1}, how to efficiently determine these m variables? Let us denote these M equations by

$\sum_{i=0}^{m-1} \alpha_i^j \cdot y_j = b^j, j = 1, \ldots, M$. For a sufficiently large value of M, one can expect the right value of $(y_0, \ldots, y_{m-1})$ to be the one maximizing the indicator:

$$I(y_0, \ldots, y_{m-1}) = \sharp\left\{ i \in \{1, \ldots, M\} \;\middle|\; \sum_{i=0}^{m-1} \alpha_i^j \cdot y_j = b^j \right\}$$
$$= \frac{N}{2} + 2 \cdot S(y_0, \ldots, y_{m-1}),$$

where:

$$S(y_0, \ldots, y_{m-1}) = \sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; \sum_{i=0}^{m-1} \alpha_i^j \cdot y_i = b^j \right\}$$
$$- \sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; \sum_{i=0}^{m-1} \alpha_i^j \cdot y_i \neq b^j \right\}.$$

Equivalently one can expect $(y_0, \ldots, y_{m-1})$ to be the value which maximizes the indicator $S(y_0, \ldots, y_{m-1})$. Instead of computing all of 2^m values of $S(y_0, \ldots, y_{m-1})$ independently, one can derive these values in a combined way using fast Walsh transform computations in order to save time.

Let us recall the definition of the Walsh transform. Given a real function of m binary variables $f(x_1, \ldots, x_{m-1})$, the Walsh transform of f is the real function of m binary variables $F = W(f)$ defined by:

$$F(u_0, \ldots, u_{m-1}) = \sum_{x_0, \ldots, x_{m-1} \in \{0,1\}^m} f(x_0, \ldots, x_{m-1})(-1)^{u_0 x_0 + \ldots + u_{m-1} x_{m-1}}.$$

Let us define the function $s(\alpha_0, \ldots, \alpha_{m-1})$ by:

$$s(\alpha_0, \ldots, \alpha_{m-1}) = \sharp\big\{ j \in \{1, \ldots, M\} \;\big|\; (\alpha_0^j, .., \alpha_{m-1}^j) = (\alpha_0, \ldots, \alpha_{m-1}) \wedge b_j = 1 \big\}$$
$$- \sharp\big\{ j \in \{1, \ldots, M\} \;\big|\; (\alpha_0^j, .., \alpha_{m-1}^j) = (\alpha_0, \ldots, \alpha_{m-1}) \wedge b_j = 0 \big\}.$$

The function s can be computed in M steps. Moreover, it is easy to check that the Walsh transform of s is S, i.e.

$$\forall (y_0, \ldots, y_{m-1}) \in \{0,1\}^m, W(s)(y_0, \ldots, y_{m-1}) = S((y_0, \ldots, y_{m-1})).$$

Therefore, the computational cost of the estimation of all the 2^m values of S using fast Walsh transform computations is $M + m \cdot 2^m$; the required memory is 2^m.

Improved Hybrid Method. More generally, if $m_1 < m$, one can use the following hybrid method between exhaustive search and Walsh transform in order to save space.

For each of the 2^{m-m_1} values of $(y_{m_1}, \ldots, y_{m-1})$, define the associated restriction S' of S as the m_1 bit boolean function given by:

$$S'(y_0, \ldots, y_{m_1-1}) = \sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; \sum_{i=0}^{m_1-1} \alpha_i^j \cdot y_i = \sum_{i=m_1}^{m} \alpha_i^j \cdot y_i \oplus b^j \right\}$$
$$- \sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; \sum_{i=0}^{m_1-1} \alpha_i^j \cdot y_i \neq \sum_{i=m_1}^{m} \alpha_i^j \cdot y_i \oplus b^j \right\}.$$

It is easy to see that if we define:

$$s'(\alpha_0, \ldots, \alpha_{m_1-1}) =$$
$$\sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; (\alpha_0^j, \ldots, \alpha_{m_1-1}^j) = (\alpha_0, \ldots, \alpha_{m_1-1}) \wedge \sum_{i=m_1}^{m} \alpha_i^j \cdot y_i \oplus b^j = 1 \right\}$$
$$- \sharp\left\{ j \in \{1, \ldots, M\} \;\middle|\; (\alpha_0^j, \ldots, \alpha_{m_1-1}^j) = (\alpha_0, \ldots, \alpha_{m_1-1}) \wedge \sum_{i=m_1}^{m} \alpha_i^j \cdot y_i \oplus b^j = 0 \right\},$$

then S' is the Walsh transform of s'.

Therefore, the computational cost of the estimation of all the 2^m values of S using this method is $2^{m-m_1}N + m_1 \cdot 2^{m_1}$. If we compare this with the former basic Walsh transform method, we see that the required memory decreases from 2^m to 2^{m_1}, whereas the time complexity increases remains negligible as long as $m_1 << log_2(M)$.

4.2 First LFSR Derivation Technique

In order to reduce the LFSR derivation complexity when compared with the naive method of complexity $N \cdot 2^n$, we can exploit more keystream to produce more linear approximation equations in the unknowns y_0 to y_{n-1}, and retain only those equations involving the $m < n$ variables y_0 to y_{m-1}, i.e. which coefficients in the $n - m$ variables y_m to y_{n-1} are equal to 0.

Thus a fraction of about 2^{m-n} of the relations are retained and we have to collect about $N2^{n-m}$ approximate relations to retain N relations. This requires a number of keystream bits of:

$$\frac{N2^{n-m} + 80}{16}.$$

As seen in the former Section, once the relations have been filtered, the computational cost of the derivation of the values of these m variables using fast Walsh transform computations is about $m2^m$ for the basic method, and more generally $2^{m-m_1}(N + m_1 2^{m_1})$ if fast Walsh transform computations are applied to a restricted set $m_1 < m$ variables.

Thus, the overall time complexity of this method is:

$$N2^{n-m} + m2^m,$$

and more generally:

$$N2^{n-m} + 2^{m-m_1}(N + m_1 2^{m_1}).$$

Once the m variables y_0 to y_{m-1} have been recovered, one can either reiterate the same technique for other choices of the m unknown variables, which increases the complexity by a factor of less than 2 if $m \geq \frac{n}{2}$, or test each of the 2^{n-m} candidates in the next step of the attack (NFSR and key derivation).

An estimate of the number N of equations needed is given by

$$N = \left(\frac{2\lambda}{3\epsilon}\right)^2,$$

where λ is determined by the condition $\frac{1}{\sqrt{2\pi}} \int_{\lambda}^{+\infty} e^{-\frac{t^2}{2}} dt = 2^{-m}$. This condition ensures that the expected number of false alarm is less than 1.

The minimal complexity is obtained for $m = 49$. For this parameter value, we have $\lambda = 7.87$ and $N = 2^{24}$. The attack complexity is about 2^{55}, the number of keystream bits needed is around 2^{51}, and the memory needed is about 2^{49}.

4.3 Second LFSR Derivation Technique

An alternative method is to derive new linear approximation equations (of lower bias) involving $m < n$ unknown variables y_0 to y_{m-1} by combining the R available approximate equations of bias ϵ pairwise, and retaining only those pairs of relations for which the $n - m$ last coefficients collide. One obtains in this way about $N' = R^2 \cdot 2^{m-n-1}$ new affine equations in y_0 to y_{m-1}, of bias $\epsilon' = 2\epsilon^2$. The allocation of the m variables maximizing the number of satisfied equations can be found by fast Walsh computations as explained in the former Section.

The number N' of relations needed is about $\left(\frac{2\lambda}{3\epsilon'}\right)^2$, where λ is determined by the condition $\frac{1}{\sqrt{2\pi}} \int_{\lambda}^{+\infty} e^{-\frac{t^2}{2}} dt = 2^{-m}$. The required number R of relations of bias ϵ is therefore $R = (N' 2^{n-m-1})^{\frac{1}{2}}$, and the number of keystream bits needed is about $\frac{R+80}{16}$. The complexity of the derivation of the N' relations is $\max(R, N') = \max((N' 2^{n-m-1})^{\frac{1}{2}}, N')$.

Once the N' relations have been derived, the computational cost of the derivation of the values of these m variables using fast Walsh transform computations is about $m \cdot 2^m$ for the basic method, and more generally $2^{m-m_1}(N' + m_1 \cdot 2^{m_1})$ if fast Walsh transform computations are applied to a restricted set $m_1 < m$ variables.

Thus the total complexity of the derivation of the m LFSR bits is:

$$\max((N' 2^{n-m-1})^{\frac{1}{2}}, N') + m2^m,$$

and more generally:

$$\max((N' 2^{n-m-1})^{\frac{1}{2}}, N') + 2^{m-m_1}(N' + m_1 2^{m_1}).$$

The minimal complexity is obtained for $m = 36$. For this parameter value, we have $\lambda = 6.65$ and $N' = 2^{41}$. The attack complexity is about 2^{43}, the number of keystream bits needed is about 2^{38} and the memory required is about 2^{42}.

5 Recovering the NFSR Initial State and the Key

Once the initial state of the LFSR has been recovered, we want to recover the initial state $(x_0, \ldots, x_{79})$ of the NFSR. Fortunately, the knowledge of the LFSR removes the nonlinearity of the output function and we can express each keystream bit z_i by one of the following four equations depending on the initial state of the LFSR:

$$\begin{aligned} z_i &= x_i, \\ z_i &= x_i \oplus 1, \\ z_i &= x_i \oplus x_{63+i}, \\ z_i &= x_i \oplus x_{63+i} \oplus 1. \end{aligned}$$

Since functions p and q underlying h are balanced, each equation has the same occurrence probability. We are going to use the non linearity of the output function to recover the initial state of the NFSR by writing the equations corresponding to the first keystream bits.

The 16 first equations are linear equations involving only bits of the initial state of the NFSR because $63 + i$ is lower than 80.

To recover all the bits of the initial state, we introduce a technique which consists of building chains of keystream bits. The equations for keystream bits z_{17} to z_{79} involve either one bit of the LFSR ($z_i = x_i$ or $z_i = x_i \oplus 1$) or two bits ($z_i = x_i \oplus x_{63+i}$ or $z_i = x_i \oplus x_{63+i} \oplus 1$). An equation involving only one bit allows us to instantly recover the value of the corresponding bit of the initial state. This can be considered as a chain of length 0. On the other hand, an equation involving two bits does not allow this because we do not know the value of x_{63+i} (for $i > 16$).

However, by considering not only the equations for z_i but also all the equation for $z_{k\cdot 63+i}$ for $k \geq 1$, we can cancel the bits we do not know and retrieve the value of x_i. With probability $\frac{1}{2}$, the equation for z_{63+i} involves one single unknown bit. Then it provides the value of x_{63+i} and consequently the value of x_i. Here the chain is of length 1, since we have to consider one extra equation to retrieve x_i. The equation for z_{63+i} can also involve two bits with probability $\frac{1}{2}$. Then we have to consider the equation of $z_{2\cdot 63+i}$, which can also either involve only one bit (we have a chain of length 2) or two bits and we have to consider more equations to solve. Each equation has a probability $\frac{1}{2}$ to involve 1 or 2 bits. Consequently the probability that a chain is of length n is $\frac{1}{2^{n+1}}$ and the probability that a chain is of length strictly larger than n is $\frac{1}{2^{n+1}}$.

We want to recover the values of $x_{17}, \ldots, x_{79}$. We have to build 64 different chains. Let us consider $L = 63 \cdot n$ bits of keystream. The probability that one of the chains is of length larger than n is less than $= 64 \cdot 2^{-n-1}$ and therefore less than 2^{-n+5}. If we want this probability to be bounded by 2^{-10}, then $n > 15$ and $L > 945$ suffices. Consequently a few thousands of keystream bits are required to retrieve the initial state of the NFSR and the complexity of the operation is bounded by $64 \cdot n$.

Since the internal state transition function associated to the special key and IV setup mode is one to one, the key can be efficiently derived from the NFSR and LFSR states at the beginning of the keystream generation by running this function backward.

6 Simulations and Results

To confirm that our cryptanalysis is correct, we ran several experiments. First we checked the bias ϵ of Section 3.1 by running the cipher with a known initial state of both the LFSR and the NLFSR, computing the linear approximations, and counting the number of fulfilled relations for a very large number of relations. For instance we found that one linear approximation is satisfied 19579367 times out of 39060639, which gives an experimental bias of $2^{-9.63}$, to be compared with the theoretical bias $\epsilon = 2^{-9.67}$.

To check the two proposed LFSR reconstruction methods of Section 5, we considered a reduced version of Grain in order to reduce the memory and time required by the attack on a single computer: we shortened the LFSR by a factor of 2. We used an LFSR of size 40 with a primitive feedback polynomial and we reduced by two the distances

for the tap entries of function h: we selected taps number 3, 14, 24, and 33, instead of 3, 25, 46, and 64 for Grain.

The complexity of the first technique for the actual Grain is 2^{55} which is out of reach of a single PC. For our reduced version, the complexity given by the formula of Section 4.2 is only 2^{35}. We exploited the 16 linear approximations to derive relations colliding on the first 11 bits. Consequently the table of the Walsh transform is only of size 2^{29}. We used $15612260 \simeq 2^{23}$ relations, which corresponds to a false alarm probability of 2^{-29}. Our implementation needed around one hour to recover the correct value of the LFSR internal state on a computer with a Intel Xeon processor running at 2.5 GHz with 3 GB of memory. The Walsh transform computation took only a few minutes.

For the actual Grain, the second technique requires only 2^{43} operations which is achievable by a single PC. However it also requires 2^{42} of memory which corresponds to 350 GB of memory. We do not have such an amount of memory but for the reduced version the required memory is only 2^{29}. Since the complexity given by the formula of Section 4.3 is dominated by the required number of relations to detect the bias, our simulation has a complexity close to 2^{43}. In practice, we obtained a result after 4 days of computation on the same computer as above and $2.5 \cdot 10^{12} \simeq 2^{41}$ relations where considered and allowed to recover the correct LFSR initial state.

Finally, we implemented the method of Section 5 to recover the NFSR. Given the correct initial state of the LFSR, and the first thousand keystream bits, our program recovers the initialization of the NFSR in a few seconds for a large number of different initializations of both the known LFSR and unknown NLFSR. We also confirmed the failure probability assessed in Section 5 for this method (which corresponds to the occurrence probability of at least one chain of length larger than 15).

7 Conclusion

We have presented a key-recovery attack against Grain which requires 2^{43} computations, 2^{42} bits of memory, and 2^{38} keystream bits. This attack suggests that the following slight modifications of some of the Grain features might improve its strength:

- Introduce several additional masking variables from the NFSR in the keystream bit computation.
- Replace the nonlinear feedback function g in such a way that the associated function g' be balanced (e.g. replace g by a 2-resilient function). However this is not necessarily sufficient to thwart all similar attacks.
- Modify the filtering function h in order to make it more difficult to approximate.
- Modify the function g and h to increase the number of inputs.

Following recent cryptanalysis of Grain including the key recovery attack reported here and distinguishing attacks based on the same kind of linear approximations as those presented in Section 3 [19] [26], the authors of Grain proposed a tweaked version of their algorithm [12], where the functions g and h' have been modified. This novel version of Grain appears to be much stronger and is immune against the statistical attacks presented in this paper.

We would like to thank Matt Robshaw and Olivier Billet for helpful comments.

References

1. M. Briceno, I. Goldberg, and D. Wagner. A pedagogical implementation of A5/1. Available at `http://jya.com/a51-pi.htm`, Accessed August 18, 2003, 1999.
2. A. Canteaut and M. Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In B. Preneel, editor, *Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 573–588. Springer-Verlag, 2000.
3. V. Chepyzhov and B. Smeets. On a fast correlation attack on certain stream ciphers. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1991.
4. M. W. Dodd. *Applications of the Discrete Fourier Transform in Information Theory and Cryptology*. PhD thesis, University of London, 2003.
5. ECRYPT. eSTREAM: ECRYPT Stream Cipher Project, IST-2002-507932. Available at `http://www.ecrypt.eu.org/stream/`, Accessed September 29, 2005, 2005.
6. P. Ekdahl and T. Johansson. Another attack on A5/1. In *Proceedings of International Symposium on Information Theory*, page 160. IEEE, 2001.
7. P. Ekdahl and T. Johansson. Another attack on A5/1. *IEEE Transactions on Information Theory*, 49(1):284–289, January 2003.
8. H. Englund and T. Johansson. A new simple technique to attack filter generators and related ciphers. In *Selected Areas in Cryptography*, pages 39–53, 2004.
9. H. Gilbert and P. Audoux. Improved fast correlation attacks on stream ciphers using FFT techniques. personnal communication, 2000.
10. J.D. Golić. Cryptanalysis of alleged A5 stream cipher. In W. Fumy, editor, *Advances in Cryptology—EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1997.
11. M. Hell, T. Johansson, and W. Meier. Grain - A Stream Cipher for Constrained Environments. ECRYPT Stream Cipher Project Report 2005/001, 2005. `http://www.ecrypt.eu.org/stream`.
12. M. Hell, T. Johansson, and W. Meier. Grain - A Stream Cipher for Constrained Environments, 2005. `http://www.it.lth.se/grain`.
13. T. Johansson and F. Jönsson. Fast correlation attacks based on turbo code techniques. In *Advances in Cryptology—CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 1999.
14. T. Johansson and F. Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. In *Advances in Cryptology—EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999.
15. F. Jönsson. *Some Results on Fast Correlation Attacks*. PhD thesis, Lund University, Department of Information Technology, P.O. Box 118, SE–221 00, Lund, Sweden, 2002.
16. A. Joux, P. Chose, and M. Mitton. Fast Correlation Attacks: An Algorithmic Point of View. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221. Springer-Verlag, 2002.
17. B. S. Jr. Kaliski and M. J. B. Robshaw. Linear Cryptanalysis Using Multiple Approximations. In Yvo G. Desmedt, editor, *Advances in Cryptology – CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 26–39. Springer-Verlag, 1994.
18. M. Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology – EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1993.
19. A. Maximov. Cryptanalysis of the "Grain" family of stream ciphers. In *ACM Transactions on Information and System Security (TISSEC)*, 2006.
20. W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In C.G. Günter, editor, *Advances in Cryptology—EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 301–316. Springer-Verlag, 1988.

21. W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3):159–176, 1989.
22. W. Meier and O. Staffelbach. The self-shrinking generator. In A. De Santis, editor, *Advances in Cryptology—EUROCRYPT'94*, volume 905 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1994.
23. M. Mihaljevic and J.D. Golić. A fast iterative algorithm for a shift register initial state reconstruction given the noisy output sequence. In J. Seberry and J. Pieprzyk, editors, *Advances in Cryptology—AUSCRYPT'90*, volume 453 of *Lecture Notes in Computer Science*, pages 165–175. Springer-Verlag, 1990.
24. NESSIE. New European Schemes for Signatures, Integrity, and Encryption. Available at `http://www.cryptonessie.org`, Accessed August 18, 2003, 1999.
25. W.T. Penzhorn and G.J. Kühn. Computation of low-weight parity checks for correlation attacks on stream ciphers. In C. Boyd, editor, *Cryptography and Coding - 5th IMA Conference*, volume 1025 of *Lecture Notes in Computer Science*, pages 74–83. Springer-Verlag, 1995.
26. M. Hassanzadeh S. Khazaei and M. Kiaei. Distinguishing attack on grain. ECRYPT Stream Cipher Project Report 2005/001, 2005. `http://www.ecrypt.eu.org/stream`.
27. T. Siegenthaler. Correlation-immunity of non-linear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, 30:776–780, 1984.
28. T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers*, 34:81–85, 1985.

Cryptanalysis of Mir-1, a T-function Based Stream Cipher

Yukiyasu Tsunoo[1], Teruo Saito[2], Hiroyasu Kubo[2], and Maki Shigeri[2]

[1] NEC Corporation
1753 Shimonumabe, Nakahara-Ku, Kawasaki, Kanagawa 211-8666, Japan
tsunoo@BL.jp.nec.com
[2] NEC Software Hokuriku Ltd.
1 Anyoji, Hakusan, Ishikawa 920-2141, Japan
{t-saito@qh, h-kubo@ps, m-shigeri@pb}.jp.nec.com

Abstract. This paper describes the cryptanalysis of Mir-1, a T-function based stream cipher proposed at eSTREAM (the ECRYPT Stream Cipher Project) in 2005. It uses a multiword T-function, with four 64-bit words, as its basic structure. Mir-1 operations process the data in every 64 bits (one word) to generate a keystream.
This paper discusses a distinguishing attack against Mir-1, one that exploits the T-function characteristics and the Mir-1 initialization. With merely three or four IV pairs, this attack can distinguish a Mir-1 output sequence from a true random sequence. In this case, the amount of data theoretically needed for cryptanalysis is only 2^{10} words.
Key words: Mir-1, ECRYPT, eSTREAM, stream cipher, pseudo-random number generator, distinguishing attack

1 Introduction

Over the past two decades, a variety of steam ciphers have been proposed. Many of these use a linear feedback shift register (LFSR) with a non-linear Boolean function to generate a keystream. However, attacks that exploit the linear characteristics of LFSR have been proposed [2, 12, 15]. LFSR-based stream ciphers might be vulnerable to such algebraic cryptanalysis.

In 2003, Klimov and Shamir proposed the T-function as a new primitive that can be used as an alternative to LFSR. The T-function is suitable for software implementation. Though it is a form of non-linear mapping, it uses a combination of operations including ADD, SUB, MUL, XOR, AND, and OR for a single cycle of maximal length. Klimov and Shamir insist that the T-function can be used not only for a stream cipher, but also for a block cipher and a hash function.

Various T-function based stream ciphers were then proposed. In 2005, Hong et al. proposed single-cycle T-functions using the S-box properties, as well as TSC-1/2 which are stream ciphers using these proposed T-functions [3]. They reported that TSC-1 is suitable for hardware implementation while TSC-2 is a stream cipher suitable for software implementation. At the FSE 2005 rump session, though, these ciphers were broken using the T-function properties [5]. In

the same year, Hong et al. proposed TSC-3, a new version of TSC, at eSTREAM [4]. However, shortly afterwards Muller and Peyrin broke this version [14].

At eSTREAM 2005, Maximov proposed Mir-1, a T-function based stream cipher [11]. The cipher uses data updated using a T-function as a key each time, and it generates a keystream through randomization by an S-box whose entries change depending on a secret key. In this paper, we propose a new cryptanalysis that exploits the T-function characteristics and the Mir-1 initialization. This method makes it possible to distinguish the output sequence of Mir-1 from a true random sequence with only three or four initial vector (IV) pairs. The amount of data theoretically needed by the method is only about 2^{10} words. Thus, with a practical amount of computation, this attack could be a threat to Mir-1.

The following section provides an overview of T-functions and recently proposed T-function based stream ciphers. Section 3 describes the structure of Mir-1. Section 4 explains how the output sequence of Mir-1 can be distinguished from a true random sequence through the T-function properties and the Mir-1 initialization. Section 5 concludes this paper.

2 T-function

This section provides a basic explanation of the T-function. The details are provided in the original paper published by Klimov and Shamir.

2.1 T-function Proposed by Klimov and Shamir

In 2002, Klimov and Shamir proposed the T-function as a new class for invertible mapping [6]. Their T-function is a single-word T-function and features single n-bit word mapping. The i-th bit of a single-word T-function output depends only on the 0th through i-th bits of its input. Single-word T-functions include arithmetic operations such as ADD, SUB, and MUL, and logical operations such as OR, AND, and XOR. These operations are referred to as primitive operations, and they are very useful because they can be processed within one clock and one cycle on many kinds of processor.

Klimov and Shamir used various combinations of these operations to design many kinds of T-functions. These T-functions feature a single cycle of maximal length. This kind of function could be used as an alternative to LFSR. However, a single-word T-function is not so useful, because its bit size n is limited to 32 or 64 in today's processors.

In 2004, Klimov and Shamir proposed multiword T-functions, which were expanded versions of single-word T-functions [8]. Multiword T-functions define m n-bit words for mapping, and they offer a single-cycle of maximal length as is offered by single-word T-functions.

The following is a more specific description of multiword T-functions with m n-bit words. If each of the m n-bit words is represented by x_k $(k = 0, \ldots, m-1)$,

the set of m words x is expressed as $\mathbf{x} = (x_k)_{k=0}^{m-1}$. The i-th bit of each word $[x_k]_i$ is then denoted as

$$[x_k] = \sum_{i=0}^{n-1} [x_k]_i 2^i$$

The layer of the i-th bit of word $\mathbf{x}$ is expressed as

$$[\mathbf{x}]_i = \sum_{k=0}^{m-1} [x_k]_i 2^k$$

Figure 1 outlines the multiword T-function defined below, where $m = 4$.

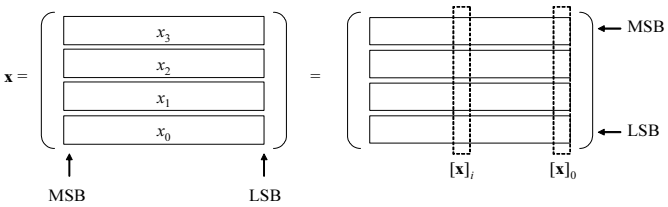


Fig. 1. Multiword T-function, where $m = 4$

Definition 1. *A (multiword) T-function is a map*

$$\mathbf{T} : \begin{cases} (\{0,1\}^n)^m \mapsto (\{0,1\}^n)^m \\ \mathbf{x} \qquad\qquad\quad \mapsto \mathbf{T}(\mathbf{x}) = (T_k(\mathbf{x}))_{k=0}^{m-1} \end{cases}$$

sending an m-tuple of n-bit words to another m-tuple of n-bit words, where each resulting n-bit word is denoted as $T_k(\mathbf{x})$, such that for each $0 \leq i < n$, the i-th bits of the resulting words $[\mathbf{T}(\mathbf{x})]_i$ are functions of just the lower input bits $[\mathbf{x}]_0, [\mathbf{x}]_1, \ldots, [\mathbf{x}]_i$.

Thus, as for multiword T-functions, the i-th bit of any output word depends only on the 0th through i-th bit of each input word.

2.2 T-function Based Stream Ciphers and Their Cryptanalysis

This section introduces T-function based stream ciphers. The paper written by Klimov and Shamir [8] gave some examples of multiword T-functions. However, Mitra and Sarkar reported in 2004 that a stream cipher employing a simple output function can be broken by a time-memory trade-off attack [13].

In 2005, Hong et al. proposed a new single-cycle T-function, which uses the S-box properties, as a T-function based stream cipher. They also proposed TSC-1/2, stream ciphers that use their proposed T-function. Both of these, however,

were broken by Junod et al. at the FSE 2005 rump session. In the same year, Hong et al. proposed TSC-3, an improved algorithm of TSC at eSTREAM. Not long afterwards, though, Muller and Peyrin broke TSC-3.

Mir-1 is a stream cipher proposed at eSTREAM by Maximov, and it basically uses the multiword T-function proposed by Klimov and Shamir. Mir-1 uses a T-function that is intended to reduce the size of internal state.

This paper describes cryptanalysis against Mir-1 that exploits the T-function characteristics and the Mir-1 initialization.

3 Description of Mir-1

This section describes the structure of Mir-1, the T-function based stream cipher proposed by Maximov at eSTREAM 2005. Ciphertexts are computed by exclusive ORing plaintexts with the keystream generated by the cipher. The keystream generation and initialization of Mir-1 is explained below.

3.1 Notation and Definition

In this paper, bit-wise XOR, AND, and OR are represented by $\oplus$, &, and |, respectively. Addition and multiplication on mod 2^{64} are denoted by $+$ and $\cdot$, respectively. $X \lll t$ denotes a t-bit rotating shift to the left of 64-bit word X. The byte unit and bit unit of 64-bit word X are set as follows, where $\|$ represents data concatenation.

$$\begin{aligned} X &= X.byte_7 \,\|\, X.byte_6 \,\|\, \cdots \,\|\, X.byte_0 \\ &= X.bit_{63} \,\|\, X.bit_{62} \,\|\, \cdots \,\|\, X.bit_0 \end{aligned}$$

The a-th through the b-th bits of 64-bit word X are represented by $X[a, b]$. Using the notation described above, we express them as

$$X[a, b] = X.bit_b \,\|\, X.bit_{b-1} \,\|\, \cdots \,\|\, X.bit_a$$

The secret key KEY of Mir-1 is 128-bit and its initial vector IV is 64-bit. They are defined as

$$\begin{aligned} KEY &= k_{15} \,\|\, k_{14} \,\|\, \cdots \,\|\, k_0 \\ IV &= IV_7 \,\|\, IV_6 \,\|\, \cdots \,\|\, IV_0 \end{aligned}$$

3.2 Keystream Generation

This section treats Mir-1's keystream generation, which consists of roughly two parts: the loop state update (LS update) and the automata state update (AS update).

The LS update has four words of 64-bit register $x_i (i = 0, 1, 2, 3)$. Register x_i is updated by a multiword T-function. The LS update function is shown in Fig. 2. It guarantees the maximal length cycle of 2^{256}.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \mapsto \begin{pmatrix} x_0 + (s) & + 2 \cdot x_2 \cdot (x_1 \mid C_1) \\ x_1 + (s \,\&\, x_0) & + 2 \cdot x_2 \cdot (x_3 \mid C_3) \\ x_2 + (s \,\&\, x_0 \,\&\, x_1) & + 2 \cdot x_0 \cdot (x_3 \mid C_3) \\ x_3 + (s \,\&\, x_0 \,\&\, x_1 \,\&\, x_2) & + 2 \cdot x_0 \cdot (x_1 \mid C_1) \end{pmatrix}$$

$$s = (x_0 \& x_1 \& x_2 \& x_3 + C_0) \oplus x_0 \& x_1 \& x_2 \& x_3$$

$$C_0 = \text{0x1248842112488421}$$
$$C_1 = \text{0x1248124812481248}$$
$$C_3 = \text{0x4812481248124812}$$

Fig. 2. Loop state update

The AS update holds two words of 64-bit registers A and B, and it computes A' and B' using the update function shown in Fig. 3. When A' and B' are computed, the register value from the LS update is two 64-bit words obtained by concatenating the upper 32 bits of each of four registers, x_0, x_1, x_2, x_3. Each 64-bit word is denoted as

$$(x_{i+2}[32, 63] \parallel x_i[32, 63]) \quad (i = 0, 1)$$

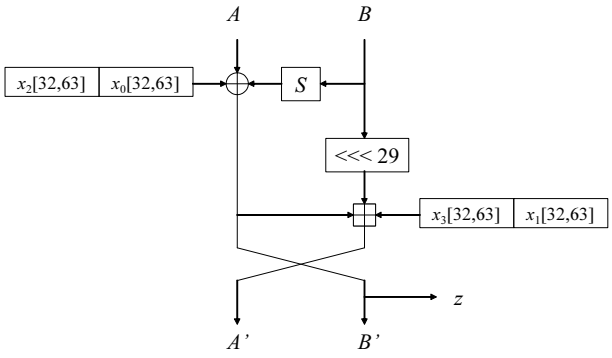


Fig. 3. Automata state update

The keystream generation part of Mir-1 performs the LS update and AS update at each clock, and outputs keystream z; that is, the 64-bit B' computed by the AS update.

3.3 Initialization

This section describes Mir-1's initialization part, which also consists of roughly two parts: the key setup and the IV setup.

The key setup initializes register $x_i (i = 0, 1, 2, 3)$ and registers A and B, using a 128-bit secret key. The key setup function is shown in Fig. 4.

1. Initialise secret S-box
2. $A = x_1 = (k_7 \parallel \ldots \parallel k_0)$
   $B = x_3 = (k_{15} \parallel \ldots \parallel k_8)$
   $x_0 = C_0$
   $x_2 = C_1$
3. Repeat 8 times
   Loop State Update
   Automata State Update

Fig. 4. Key setup

First, the key setup computes an S-box, which varies depending on the secret key value referred to as the secret S-box, using the equation shown below. Here, $SR[\cdot]$ means the S-box of Rijndael. Each entry is computed for $i = 0, \ldots, 255$.

$$S[i] = SR[\cdots SR[SR[i \oplus k_0] \oplus k_1] \oplus \cdots \oplus k_{15}]$$

The IV setup uses a 64-bit initial vector to update register $x_i (i = 0, 1, 2, 3)$ as well as registers A and B. The IV setup function is shown in Fig. 5.

4 Cryptanalysis of Mir-1

This section describes the method to attack the Mir-1 stream cipher. Section 4.1 explains the structural properties of the IV setup and LS update, which are necessarily exploited for the cryptanalysis, and section 4.2 describes the cryptanalysis using these properties. Section 4.3 discusses the results of an experimental attack made on Mir-1.

4.1 Properties of IV Setup and LS Update

This section describes the properties of the IV setup and LS update, on which the key setup has no particular influence.

First, we explain the structural properties of the IV setup. As shown in Fig. 5, the IV setup divides a 64-bit IV into eight 8-bit data, each of which is substituted

1. $x_0.byte_4 = x_0.byte_4 \oplus S[IV_0] \oplus S[IV_1] \oplus S[IV_2]$
   $x_1.byte_4 = x_1.byte_4 \oplus S[IV_0] \oplus S[IV_3] \oplus S[IV_4]$
   $x_2.byte_4 = x_2.byte_4 \oplus S[IV_2] \oplus S[IV_5] \oplus S[IV_7]$
   $x_3.byte_4 = x_3.byte_4 \oplus S[IV_3] \oplus S[IV_6] \oplus S[IV_7]$
2. $x_0.byte_0 = x_0.byte_0 \oplus S[IV_3] \oplus S[IV_5]$
   $x_1.byte_0 = x_1.byte_0 \oplus S[IV_7] \oplus S[IV_6]$
   $x_2.byte_0 = x_2.byte_0 \oplus S[IV_0] \oplus S[IV_1]$
   $x_3.byte_0 = x_3.byte_0 \oplus S[IV_2] \oplus S[IV_4]$
3. $A.byte_0 = A.byte_0 \oplus S[IV_0] \oplus S[IV_5] \oplus S[IV_6]$
   $A.byte_4 = A.byte_4 \oplus S[IV_1] \oplus S[IV_3] \oplus S[IV_5]$
   $B.byte_0 = B.byte_0 \oplus S[IV_1] \oplus S[IV_4] \oplus S[IV_7]$
   $B.byte_4 = B.byte_4 \oplus S[IV_2] \oplus S[IV_4] \oplus S[IV_6]$
4. Repeat 2 times
   Loop State Update
   Automata State Update

Fig. 5. IV setup

in the secret S-box to be XORed with each register. Thus, if entries of the secret S-box are unknowns, then each register is XORed with an unknown. Here, we assume that each byte inputs the same value, $IVa = (a \parallel a \parallel \cdots \parallel a)$ to the IV setup. Then the data XORed with each register in the IV setup are as shown in Fig. 6.

As shown in Fig. 6, the IV has no influence over registers x_0, x_1, x_2, or x_3 at step 2, regardless of the value of a. At steps 1 and 3, all the data XORed with each register becomes $S[a]$. Though entries of the secret S-box are unknown because of their dependence on the secret key, it is apparent that all the registers are XORed with the same value.

Next, we describe the LS update properties. As explained in Section 2.1, as far as multiword T-functions are concerned, the n-th bit of any output word depends only on the 0th through n-th bit of each input word. Thus, if differential Δ_i is given as the initial value of register $x_i (i = 0, 1, 2, 3)$, and if all of the 0th through n-th bits of differential Δ_i are 0, then the differential of the 0th through n-th bits of register x_i is always 0, regardless of the number of times the LS update is performed.

As shown in Fig. 6, if the same IV value, $IVa = (a \parallel a \parallel \cdots \parallel a)$ is input to IV setup, the IV has no influence over the 0th through 31st bits of register $x_i (i = 0, 1, 2, 3)$. Consequently, while the secret key is fixed, no changes are made on the 0th through 31st bits of register x_i, regardless of the number of times the LS update is performed, even if IVa is changed.

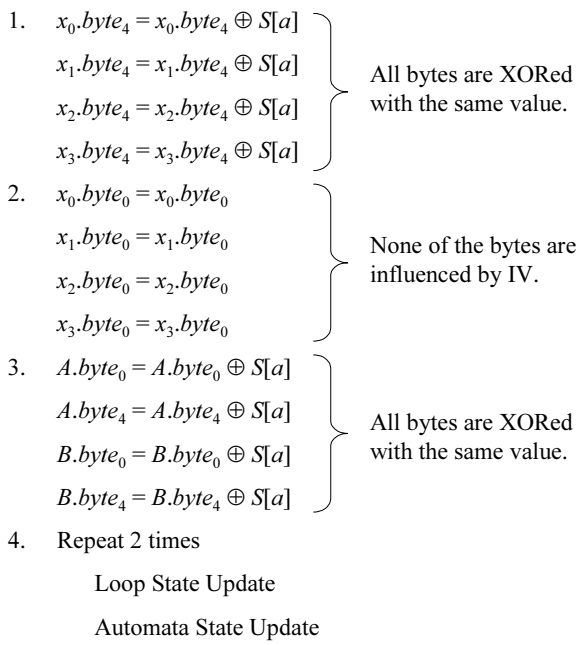


Fig. 6. IV setup where each byte inputs the same IVa

The following section describes the cryptanalysis where these two properties are exploited.

4.2 Attack Method

This section describes an attack method that exploits the two properties described in Section 4.1. For this attack to succeed, the following preconditions must be met.

- The secret key is fixed during the attack.
- Attackers can choose the IV freely.
- Attackers can obtain the keystream generated using the given IV.

First, a pair of $IVa = (a \parallel a \parallel \cdots \parallel a)$ and $IVb = (b \parallel b \parallel \cdots \parallel b)$ is provided for the IV setup. Note that all the bytes of IVa as well as those of IVb have the same value. Because of the structural properties of the IV setup described in Section 4.1, the difference between each of the lower 32 bits of register $x_i (i = 0, 1, 2, 3)$ updated by IVa and the corresponding bits updated by IVb becomes 0. In other words, the equation given below holds true, where xa_i and xb_i respectively represent the register $x_i (i = 0, 1, 2, 3)$ updated by IVa and that updated by IVb.

$$xa_i[0, 31] = xb_i[0, 31] \quad (i = 0, 1, 2, 3) \tag{1}$$

When IVa and IVb are given, byte 4 of register x_i is XORed with $S[a]$ and $S[b]$, respectively. This is expressed by the following equations.

$$xa_i.byte_4 = x_i.byte_4 \oplus S[a] \quad (i = 0, 1, 2, 3)$$
$$xb_i.byte_4 = x_i.byte_4 \oplus S[b] \quad (i = 0, 1, 2, 3)$$

Here, the entries of the secret S-box are unknowns. We can assume, though, that the equation below is satisfied:

$$S[a] \,\&\, 1 = S[b] \,\&\, 1 \tag{2}$$

If the condition of Eq. (2) is met, the relation described in Eq. (3) holds true for bit 32 of register xa_i and bit 32 of register xb_i.

$$xa_i.bit_{32} = xb_i.bit_{32} \quad (i = 0, 1, 2, 3) \tag{3}$$

Thus, if all the bytes of IVa as well as those of IVb have the same value, and if the condition of Eq. (2) is satisfied, the equation below is supported by Eqs. (1) and (3):

$$xa_i[0, 32] = xb_i[0, 32] \quad (i = 0, 1, 2, 3) \tag{4}$$

Consequently, the difference between each of the lower 33 bits of register $x_i (i = 0, 1, 2, 3)$ updated by IVa and the corresponding bit updated by IVb becomes 0. As described in Section 4.1, because of the LS update properties Eq. (4) always holds true for the IV setup and the keystream generation, regardless of the number of times the LS update is performed.

Here, we consider the keystream generation, presuming that the condition of Eq. (2) is met. Figure 7 outlines the AS update for three clocks. Though the AS update uses addition in mod 2^{64}, this operation can be substituted by XOR if only the least significant bit is used for cryptanalysis. In Fig. 7, addition in mod 2^{64} is substituted by XOR, taking only the least significant bit into account. To simplify the explanation given hereafter, the two 64-bit words inserted by the LS update are represented by $x20 = (x_2[32, 63] \parallel x_0[32, 63])$, $x31 = (x_3[32, 63] \parallel x_1[32, 63])$, and data X at time t is denoted as $X^{(t)}$.

Here, we describe the method to create a distinguisher. The keystreams generated by IVa and IVb are denoted as za and zb, respectively. The data inserted by the LS update, when IVa and IVb are given, are represented by $(x20a, x31a)$ and $(x20b, x31b)$, respectively. The least significant bit at the position where the secret S-box is output at time t is then expressed as follows. Note that these equations mean $ROL29(X) = X \lll 29$.

$$\{ROL29(za^{(t-1)}) \oplus za^{(t)} \oplus x31a^{(t-1)} \oplus x20a^{(t)} \oplus za^{(t+1)}\}.bit_0 = S[za^{(t)}] \,\&\, 1$$
$$\{ROL29(zb^{(t-1)}) \oplus zb^{(t)} \oplus x31b^{(t-1)} \oplus x20b^{(t)} \oplus zb^{(t+1)}\}.bit_0 = S[zb^{(t)}] \,\&\, 1$$

Here, Eq. (3) satisfies Eqs. (5) and (6).

$$x20a^{(t)}.bit_0 = x20b^{(t)}.bit_0 \tag{5}$$
$$x31a^{(t-1)}.bit_0 = x31b^{(t-1)}.bit_0 \tag{6}$$

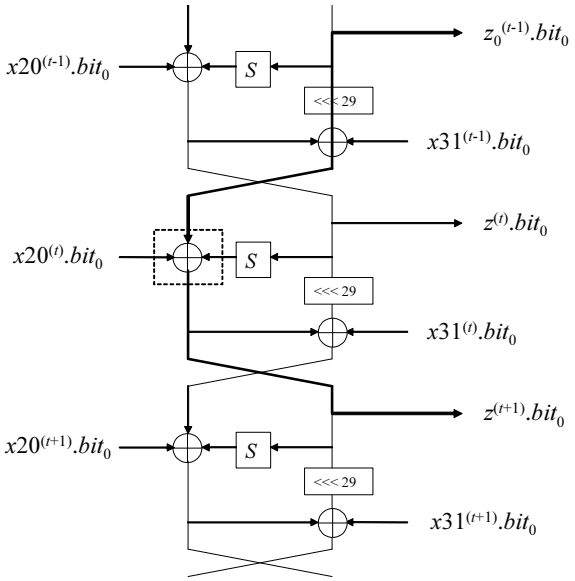


Fig. 7. AS update (LSB) for three clocks

If time t satisfying

$$za^{(t)} = zb^{(t)} \tag{7}$$

is chosen for the keystreams generated by IVa and IVb, the equation described below holds true because the unknown entries of the secret S-box take the same input, resulting in the same secret S-box output.

$$S[za^{(t)}] = S[zb^{(t)}] \tag{8}$$

Thus, Eqs. (5), (6), (7), and (8) support Eq. (9):

$$\{ROL29(za^{(t-1)} \oplus zb^{(t-1)}) \oplus za^{(t+1)} \oplus zb^{(t+1)}\}.bit_0 = 0 \tag{9}$$

In summary, if any given pair of IVa and IVb satisfies Eq. (2), then Eq. (9) always holds true at the time t where Eq. (7) holds. Thus, Eq. (9) can be used as a distinguisher that distinguishes a Mir-1 output sequence from a true random sequence.

Since the probability that Eq. (2) is satisfied is the probability that the least significant bits of two randomly chosen secret S-box entries match each other, it becomes 1/2. The probability that Eq. (7) is satisfied becomes 2^{-8}, assuming that the keystream of Mir-1 is a true random number sequence. Thus, according to Mantin and Shamir [10], when arbitrary IV pairs are chosen, the amount of data T required to distinguish the Mir-1 output sequence from a true random number sequence is theoretically determined by

$$T = (1/2)^{-2} \times 2^8 = 2^{10}$$

4.3 Experimental Results

In this section we discuss the outcome of an experimental attack like the one described in Section 4.2. Preconditions for the experimental attack are defined in Section 4.2, and the steps described below were taken to make the experimental attack.

1. Generate keystreams corresponding to $IV0 = (0 \parallel 0 \parallel \cdots \parallel 0)$ and $IV1 = (1 \parallel 1 \parallel \cdots \parallel 1)$.
2. Find w values of t, where t represents the time at which the least significant byte of the keystream generated by $IV0$ matches that for $IV1$. If we assume that the keystream of Mir-1 is a true random number sequence, the existence probability of t becomes 2^{-8}.
3. Check to see if the distinguisher of Eq. (9) holds true at the w values of t that satisfy the condition described in step 2.
4. If the distinguisher described in step 3 holds true for all w values of t, it is judged to be a Mir-1 keystream sequence. If there is any t for which the distinguisher does not hold true, increment each byte of $IV1$ by 1 and repeat steps 2 and 3.

Given 100 randomly generated secret keys, we made the experimental attack as described above to obtain the number of times IV was changed and the number of secret keys with which the Mir-1 output sequence was distinguished from a true random number sequence, where $w = 128$. [1] Table 1 shows the results of this attack.

Table 1. Number of times IV was changed and the number of distinguishable secret keys with which a Mir-1 output sequence was distinguished from a true random number sequence

IV Changes	Number of Secret Keys Used as a Distinguisher
0	57
1	79
2	91
3	95
4	98
5	100

Satisfying Eq. (2) means that the output sequence of Mir-1 can be distinguished from a true random number sequence. Thus, if an IV pair is given randomly, the Mir-1 output sequence must be distinguished from a true random number sequence at a probability of 1/2. This means that as the number

[1] If $w = 128$, the probability that the distinguisher holds true accidentally is 2^{-128}.

of times the IV is changed is incremented by 1, with one-half of the remaining secret keys, the Mir-1 output sequence must be distinguished from a random number sequence. Thus, we consider the distinguisher to hold true at the probability we expected. Since the entries of the secret S-box are unknown, we cannot say that the distinguisher holds true with any given IV pair. However, as the S-box is a function of bijection, at worst it is apparent that Eq. (2) is necessarily satisfied if the IV is changed 128 times.

The attack proposed in this paper can distinguish a Mir-1 output sequence from a true random number sequence if the chosen IV pairs are provided. With about three or four distinct IV pairs, the distinguisher holds true at a probability of approximately 90%. Under the worst conditions, the distinguisher holds true if 128 distinct IV pairs are provided. We have verified that the proposed attack is applicable to Mir-1 and that it distinguishes a Mir-1 output sequence from a true random number sequence with a very small amount of data.

5 Conclusion

This paper describes the cryptanalysis of Mir-1, a new T-function based stream cipher. The IV used for stream cipher is a parameter that users can choose freely. Thus, an attack using the chosen IVs can be a threat. This paper proposes an effective distinguisher that uses the chosen IVs and the structural properties of the Mir-1 initialization. With a mere three or four chosen IV pairs, the attack method proposed in this paper distinguishes a Mir-1 output sequence from a true random sequence at a high probability. The theoretical amount of data required for the attack is no more than about 2^{10} words.

The attack method proposed in this paper makes effective use of the T-function properties. This is an effective way to attack a T-function based stream cipher. Stream ciphers based on T-functions will probably be used as an alternative to LFSR. However, designers of T-function based stream ciphers should pay attention to this vulnerability to make their ciphers resistant to such attacks.

Note that the attack proposed in this paper has not been developed into a key recovery attack. However, this paper describes the first Mir-1 cryptanalysis. The attack is strong, because it can distinguish a Mir-1 output sequence from a true random number sequence with only small amounts of data and computation needed.

Acknowledgement

The authors would like to thank Alexander Maximov and Masashi Kogiso for their useful comments.

References

1. eSTREAM, the ECRYPT Stream Cipher Project.
   Available at http://www.ecrypt.eu.org/stream/

2. N. Courtois and W. Meier: "Algebraic Attacks on Stream Ciphers with Linear Feedback," *Advances in Cryptology - EUROCRYPT 2003*, LNCS 2656, pp.345-359, Springer Verlag, 2003.
3. J. Hong, D. Lee, Y. Yeom, and D. Han: "A New Class of Single Cycle T-functions," *Fast Software Encryption, FSE 2005*, LNCS 3557, pp.68-82, Springer Verlag, 2005.
4. J. Hong, D. Lee, Y. Yeom, and D. Han: "T-function Based Stream Cipher TSC-3," *ECRYPT Stream Cipher Project*, Report 2005/031, 2005.
5. P. Junod, S. Kunzli, and W. Meier: "Attacks against TSC," Rump Session at FSE 2005. Available at http://crypto.junod.info/rump_session_fse05.pdf
6. A. Klimov and A. Shamir: "A New Class of Invertible Mappings," *Cryptographic Hardware and Embedded Systems, CHES 2002*, LNCS 2523, pp.470-483, Springer Verlag, 2002.
7. A. Klimov and A. Shamir: "Cryptographic Applications of T-functions," *Selected Areas in Cryptography, SAC 2003*, LNCS 3006, pp.248-261, Springer Verlag, 2004.
8. A. Klimov and A. Shamir: "New Cryptographic Primitives Based on Multiword T-functions," *Fast Software Encryption, FSE 2004*, LNCS 3017, pp.1-15, Springer Verlag, 2004.
9. A. Klimov and A. Shamir: "New Applications of T-functions in Block Ciphers and Hash Functions," *Fast Software Encryption, FSE 2005*, LNCS 3557, pp.18-31, Springer Verlag, 2005.
10. I. Mantin and A. Shamir: "A Practical Attack on Broadcast RC4," *Fast Software Encryption, FSE 2001*, LNCS 2355, pp.152-164, Springer Verlag, 2001.
11. A. Maximov: "A New Stream Cipher "Mir-1"," *ECRYPT Stream Cipher Project*, Report 2005/017, 2005.
12. W. Meier and O. Staffelbach: "Fast Correlation Attacks on Certain Stream Ciphers," *Journal of Cryptology*, pp.159-176, Springer Verlag, 1989.
13. J. Mitra and P. Sarkar: "Time-memory Trade-off Attacks on Multiplications and T-functions," *Advances in Cryptology - ASIACRYPT 2004*, LNCS 3329, pp.468-482, Springer Verlag, 2004.
14. F. Muller and T. Peyrin: "Linear Cryptanalysis of TSC Stream Ciphers - Applications to the ECRYPT Proposal TSC-3," *ECRYPT Stream Cipher Project*, Report 2005/042, 2005.
15. T. Siegenthaler: "Decryption a class of stream ciphers using ciphertext only," *IEEE Transactions on Computers*, vol. C-34, no.1, pp.81-85, January 1985.

Truncated differential cryptanalysis of five rounds of Salsa20

Paul Crowley

LShift Ltd, www.lshift.net

Abstract We present an attack on Salsa20 reduced to five of its twenty rounds. This attack uses many clusters of truncated differentials and requires 2^{165} work and 2^6 plaintexts.
Keywords: Salsa20, symmetric cryptanalysis

1 Definition of Salsa20

Salsa20 [1] is a candidate in the eSTREAM project to identify new stream ciphers that might be suitable for widespread adoption. For convenience, we recap here the parameterized family of variants Salsa20-w/r, with w the word size and r the number of rounds; Salsa20 itself is Salsa20-32/20. A **word** is an element of $\mathbb{Z}/2^w\mathbb{Z}$. We omit the precise definitions of word-oriented operations here for brevity; addition ($+$), XOR ($\oplus$) and rotation ($\lll$) are defined in the usual way, and where words are mapped to bytes, a little-endian mapping is used. We define a bijective map S on four-element column vectors of words:

$$S_a((\ y_0\ y_1\ y_2\ y_3\ )^T) = (\ y_1 \oplus ((y_0 + y_3) \lll a)\ y_2\ y_3\ y_0\ )^T$$

and compose it four times to build this bijective map on the same:

$$Q = S_{18} \circ S_{13} \circ S_9 \circ S_7$$

(note that the constants given in the subscripts are appropriate for $w = 32$; different constants might be used for a different w) and compose it with a row and column rotate to get this bijective map on matrices:

$$Q'(m) = \begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & q_1 \\ m_{2,1} & m_{2,2} & m_{2,3} & q_2 \\ m_{3,1} & m_{3,2} & m_{3,3} & q_3 \\ m_{0,1} & m_{0,2} & m_{0,3} & q_0 \end{pmatrix}$$

$$\text{where} \quad q = Q \begin{pmatrix} m_{0,0} \\ m_{1,0} \\ m_{2,0} \\ m_{3,0} \end{pmatrix}, \quad m = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix}$$

from which we build this bijective map on four-by-four square matrices of words:

$$R(m) = (Q'^4(m))^T$$

and from this, we define the Salsa20 "hash function":

$$H(m) = m + R^r(m)$$

Salsa20 maps an eight-word key $k_{0\ldots7}$, a two-word nonce $v_{0\ldots1}$ and a two-word stream position $i_{0\ldots1}$ onto a 16-word output matrix as follows:

$$\text{Salsa20}_k(v,i) = H \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ i_0 & i_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix}$$

where $c_{0\ldots3}$ are constants dependent on the key length and omitted here for brevity. We also omit the (straightforward) definition of the row-wise deserialization of this output matrix, the resulting counter-mode-like stream cipher, and the Salsa20 variants defined for shorter keys. Salsa20's security goal is that the function above be indistinguishable from a random function to a suitably-bounded attacker; from this its security as a stream cipher may be inferred.

2 Cryptanalysis of $r = 5$

We here attack the Salsa20 PRF directly; the resulting attack on the Salsa20 stream cipher follows straightforwardly. Though many techniques of block cipher cryptanalysis are applicable to Salsa20, it has several features to defeat these techniqes. First, the large block size allows for rapid diffusion without penalty of speed. Second, the attacker can control only four words of the sixteen-word input to the block cipher stage. Nevertheless, we can construct an attack based on multiple truncated differentials which breaks five rounds of the cipher.

Where $r = 5$, the output of the PRF is $m + R^5(m)$. Eight of the sixteen cells in m are known to us; the other eight cells contain the key. We can thus straightforwardly infer eight of the sixteen cells in $R^5(m)$. If we correctly guess k_3, this will give us a complete row in $R^5(m)$, to which we can apply Q^{-1} to infer a complete row of $R^4(m)$.

To go further back, we observe that if every input word but the first to Q^{-1} is known, the final output word may be inferred, and if every input but the second is known, the first may be inferred. If we can guess the key words $k_{3\ldots7}$, this allows us to infer these entries of $R^5(m)$ given $H(m)$:

$$\begin{pmatrix} \bullet & ? & ? & ? \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{pmatrix}$$

Applying Q^{-1} to each row except the first allows us to infer these entries in $R^4(m)$:

$$\begin{pmatrix} ? & \bullet & \bullet & \bullet \\ ? & \bullet & \bullet & \bullet \\ ? & \bullet & \bullet & \bullet \\ ? & \bullet & \bullet & \bullet \end{pmatrix}$$

from which we can infer these entries in $R^3(m)$:

$$\begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ \bullet & ? & ? & \bullet \end{pmatrix}$$

Given a sufficiently powerful distinguisher for the function family $f_k(v,i) = (\text{Salsa20}_k(v,i)_{3,0}, \text{Salsa20}_k(v,i)_{3,3})$ we can therefore test our guesses at $k_{3\ldots7}$.

Consider this example of a low-weight (ie high-probability) truncated differential trail suitable for our purposes, identifed using the techniques of [2]. The limitations on the bits under the attacker's control make it difficult to identify trails that start with useful combinations of bits; each word we control is combined with three we do not before the results are combined with each other. Thus, our input difference is simply a single bit in the high word of the stream position, chosen to minimize the nonlinear avalanche. Before round 1:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0x80000000 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

After round 1 (with probability $\frac{1}{2}$):

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0x00201000 & ? & 0x80000000 & 0x00000100 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

After round 2 (with probability 2^{-9}):

$$\begin{pmatrix} ? & 0x00201000 & 0x40200000 & 0x02000800 \\ ? & ? & ? & ? \\ ? & ? & ? & 0x00000040 \\ 0 & 0x00001000 & 0x00200000 & 0x04000080 \end{pmatrix}$$

And after round 3 (with probability 2^{-12}):

$$\begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0x02002802 & ? & ? & ? \end{pmatrix}$$

This trail has sufficiently high probability to act as a suitable distinguisher from which an attack can be built. However, we can do much better. The probability of this difference appearing in the output is much higher than this trail would suggest—in fact, it is closer to 2^{-9}. This is because there are many other

low-weight differential trails that result in this difference in $R^3(m)_{3,0}$. Furthermore, there are many high-probability differentials in this word. By experiment, we have even determined a few differential trails whose probability appears to be twice as high as their weight would suggest—this is presumably because of problems with the independence assumption, and suggests that there may be trails which are less probable than their weight would suggest.

By considering many trails, we can build a far more effective attack. Many tradeoffs are possible; we give one example here. We have experimentally determined a set of 1024 possible differences in $R^3(m)_{3,0}$ from this one input difference such that the probability of one of them being right appears to be roughly 30%. With 32 output pairs, the probability that 5 or more of these pairs show a difference in the set is greater than $1 - 2^{-3}$, while the probability of this threshold being met or exceeded by chance is less than 2^{-99}. We try all 2^{160} possible values of $k_{3\ldots7}$; for each that meets the threshold, we try to determine $k_{0\ldots2}$ by simple brute-force search. The true key will be among these values with probability $1 - 2^{-3}$ as noted, and we can expect $2^{160-99} = 2^{61}$ false positives; the cost of the brute-force search stage will thus be roughly $2^{96+61} = 2^{157}$, much less than the cost of determining our candidates for $k_{3\ldots7}$.

3 Conclusions and open questions

It is clear that a naive attack of this type cannot be extended to more than a handful of rounds; this has no negative implications for the security of the full Salsa20-32/20 presented to eSTREAM.

Nonetheless, the degree of clustering exhibited by these differential characteristics is surprising; it is more usual for a single differential trail to dominate. It is also striking to find differential trails whose overall probability is so greatly mispredicted by the products of the probabilities of its components, marking a violation of the independence assumption usual in differential cryptanalysis. In both instances, it would bear investigation whether other ciphers that rely heavily on addition mod 2^n to introduce nonlinearity in $GF(2)$ would also show these properties in differential cryptanalysis, or related properties in other forms of cryptanalysis.

References

1. Daniel J. Bernstein. Salsa20 specification, 2005.
2. Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In Mitsuru Matsui, editor, *Fast Software Encryption 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2001.

http://www.ciphergoth.org/crypto/salsa20/

A Anomalous differential trails

We give here examples of differential trails whose observed frequency is markedly different from that predicted by the simplifying assumptions of differential cryptanalysis. The trails below should appear with frequency 2^{-9}, but in 2^{26} trials appeared not the expected 131072 times, but 262018 and 262412 times respectively. Both trails start

$$\begin{pmatrix} 0 & 0 & 0\ 0 \\ 0 & 0 & 0\ 0 \\ 0 & 0x80000000 & 0\ 0 \\ 0 & 0 & 0\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0x00601000 & ? & 0x80000000 & 0x00000100 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

One goes on thus:

$$\begin{pmatrix} ? & 0x00601000 & 0x40200000 & 0x02000800 \\ ? & ? & ? & ? \\ ? & ? & ? & 0x00000040 \\ 0 & 0x00000100 & ? & ? \end{pmatrix}$$

and the other thus:

$$\begin{pmatrix} ? & 0x00601000 & 0x40200000 & 0x02001800 \\ ? & ? & ? & ? \\ ? & ? & ? & 0x00000040 \\ 0 & 0x00000100 & ? & ? \end{pmatrix}$$

TRIVIUM
A Stream Cipher Construction Inspired by Block Cipher Design Principles*

Christophe De Cannière[1] and Bart Preneel[2]

[1] IAIK Krypto Group, Graz University of Technology
Inffeldgasse 16A, A–8010 Graz, Austria
christophe.decanniere@iaik.tugraz.at
[2] Katholieke Universiteit Leuven, Dept. ESAT/SCD-COSIC,
Kasteelpark Arenberg 10, B–3001 Heverlee, Belgium
bart.preneel@esat.kuleuven.be

Abstract. In this paper, we propose a new stream cipher construction based on block cipher design principles. The main idea is to replace the building blocks used in block ciphers by equivalent stream cipher components. In order to illustrate this approach, we construct a very simple synchronous stream cipher which provides a lot of flexibility for hardware implementations, and seems to have a number of desirable cryptographic properties.

1 Introduction

In the last few years, widely used stream ciphers have started to be systematically replaced by block ciphers. An example is the A5/1 stream cipher used in the GSM standard. Its successor, A5/3, is a block cipher. A similar shift took place with wireless network standards. The security mechanism specified in the original IEEE 802.11 standard (called 'wired equivalent privacy' or WEP) was based on the stream cipher RC4; the newest standard, IEEE 802.11i, makes use of the block cipher AES.

The declining popularity of stream ciphers can be explained by different factors. The first is the fact that the security of block ciphers seems to be better understood. Over the last decades, cryptographers have developed a rather clear vision of what the internal structure of a secure block cipher should look like. This is much less the case for stream ciphers. Stream ciphers proposed in the past have been based on very different principles, and many of them have shown weaknesses. A second explanation is that efficiency, which has been the traditional motivation for choosing a stream cipher over a block cipher, has ceased to be a decisive factor in many applications: not only is the cost of computing power rapidly decreasing, today's block ciphers are also significantly more efficient than their predecessors.

Still, it seems that stream ciphers could continue to play an important role in those applications where high througput remains critical and/or where resources are very restricted. This poses two challenges for the cryptographic community: first, restoring the confidence in stream ciphers, e.g., by developing simple and

* The work described in this paper has been partly supported by the European Commission under contract IST-2002-507932 (ECRYPT), by the Fund for Scientific Research – Flanders (FWO), and by the Austrian Science Fund (FWF project P18138).

reliable design criteria; secondly, increasing the efficiency advantage of stream ciphers compared to block ciphers.

In this paper, we try to explore both problems. The first part of the article reviews some concepts which lie at the base of today's block ciphers (Sect. 3), and studies how these could be mapped to stream ciphers (Sects. 4–5). The design criteria derived this way are then used as a guideline to construct a simple and flexible hardware-oriented stream cipher in the second part (Sect. 6).

2 Security and Efficiency Considerations

Before devising a design strategy for a stream cipher, it is useful to first clearly specify what we expect from it. Our aim in this paper is to design a hardware-oriented binary additive stream cipher which is both efficient and secure. The following sections briefly discuss what this implies.

2.1 Security

The additive stream cipher which we intend to construct takes as input a k-bit secret key K and an n-bit IV. The cipher is then requested to generate up to 2^d bits of key stream $z_t = S_K(IV, t)$, $0 \leq t < 2^d$, and a bitwise exclusive OR of this key stream with the plaintext produces the ciphertext. The security of this additive stream cipher is determined by the extent to which it mimics a one-time pad, i.e., it should be hard for an adversary, who does not know the key, to distinguish the key stream generated by the cipher from a truly random sequence. In fact, we would like this to be as hard as we can possibly ask from a cipher with given parameters k, n, and d. This leads to a criterion called K-security [1], which can be formulated as follows:

Definition 1. *An additive stream cipher is called K-secure if any attack against this scheme would not have been significantly more difficult if the cipher had been replaced by a set of 2^k functions $S_K\colon \{0,1\}^n \times \{0,\ldots,2^d-1\} \to \{0,1\}$, uniformly selected from the set of all possible functions.*

The definition assumes that the adversary has access to arbitrary amounts of key stream, that he knows or can choose the a priory distribution of the secret key, that he can impose relations between different secret keys, etc.

Attacks against stream ciphers can be classified into two categories, depending on what they intend to achieve:

- *Key recovery attacks*, which try to deduce information about the secret key by observing the key stream.
- *Distinguishing attacks*, the goal of which is merely to detect that the key stream bits are not completely unpredictable.

Owing to their weaker objective, distinguishing attacks are often much easier to apply, and consequently harder to protect against. Features of the key stream that can be exploited by such attacks include periodicity, dependencies between bits at different positions, non-uniformity of distributions of bits or words, etc. In this paper we will focus in particular on linear correlations, as it appeared to be the weakest aspect in a number of recent stream cipher proposals such as SOBER-t*w* [2] and SNOW 1.0 [3]. Our first design objective will be to keep the largest correlations below safe bounds. Other important properties, such as

a sufficiently long period, are only considered afterwards. Note that this approach differs from the way LFSR or T-function based schemes are constructed. The latter are typically designed by maximizing the period first, and only then imposing additional requirements.

2.2 Efficiency

In order for a stream cipher to be an attractive alternative to block ciphers, it must be efficient. In this paper, we will be targeting hardware applications, and a good measure for the efficiency of a stream cipher in this environment is the number of key stream bits generated per cycle per gate.

There are two ways to obtain an efficient scheme according to this measure. The first approach is illustrated by A5/1, and consists in minimizing the number of gates. A5/1 is extremely compact in hardware, but it cannot generate more than one bit per cycle. The other approach, which was chosen by the designers of PANAMA [4], is to dramatically increase the number of bits per cycle. This allows to reduce the clock frequency (and potentially also the power consumption) at the cost of an increased gate count. As a result, PANAMA is not suited for environments with very tight area constraints. Similarly, designs such as A5/1 will not perform very well in systems which require fast encryption at a low clock frequency. One of the objectives of this paper is to design a flexible scheme which performs reasonably well in both situations.

3 How Block Ciphers are Designed

As explained above, the first requirement we impose on the construction is that it generates key streams without exploitable linear correlations. This problem is very similar to the one faced by block cipher designers. Hence, it is natural to attempt to borrow some of the techniques used in the block cipher world. The ideas relevant to stream ciphers are briefly reviewed in the following sections.

3.1 Block Ciphers and Linear Characteristics

An important problem in the case of block ciphers is that of restricting linear correlations between input and output bits in order to thwart linear cryptanalysis [5]. More precisely, let P be any plaintext block and C the corresponding ciphertext under a fixed secret key, then any linear combination of bits

$$\Gamma_P^{\mathsf{T}} \cdot P + \Gamma_C^{\mathsf{T}} \cdot C \,,$$

where the column vectors Γ_P and Γ_C are called *linear masks*, should be as balanced as possible. That is, the correlation

$$c = 2 \cdot \frac{|\{P \mid \Gamma_P^{\mathsf{T}} \cdot P = \Gamma_C^{\mathsf{T}} \cdot C\}|}{|\{P\}|} - 1$$

has to be close to 0 for any Γ_P and Γ_C. The well-established way to achieve this consists in alternating two operations. The first splits blocks into smaller words which are independently fed into nonlinear substitution boxes (S-boxes); the second step recombines the outputs of the S-boxes in a linear way in order to 'diffuse' the nonlinearity. The result, called a substitution-permutation network, is depicted in Fig. 1.

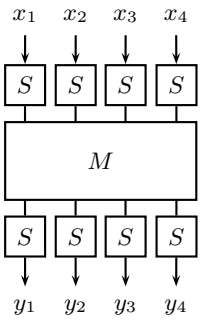


Fig. 1. Three layers of a block cipher

In order to estimate the strength of a block cipher against linear cryptanalysis, one will typically compute bounds on the correlation of *linear characteristics*. A linear characteristic describes a possible path over which a correlation might propagate through the block cipher. It is a chain of linear masks, starting with a plaintext mask and ending with a ciphertext mask, such that every two successive masks correspond to a nonzero correlation between consecutive intermediate values in the cipher. The total correlation of the characteristic is then estimated by multiplying the correlations of all separate steps (as dictated by the so-called Piling-up Lemma).

3.2 Branch Number

Linear diffusion layers, which can be represented by a matrix multiplication $Y = M \cdot X$, do not by themselves contribute in reducing the correlation of a characteristic. Clearly, it suffices to choose $\Gamma_X = M^\mathsf{T} \cdot \Gamma_Y$, where M^T denotes the transpose of M, in order to obtain perfectly correlating linear combinations of X and Y:

$$\Gamma_Y^\mathsf{T} \cdot Y = \Gamma_Y^\mathsf{T} \cdot MX = (M^\mathsf{T}\Gamma_Y)^\mathsf{T} \cdot X = \Gamma_X^\mathsf{T} \cdot X\,.$$

However, diffusion layers play an important indirect role by forcing characteristics to take into account a large number of nonlinear S-boxes in the neighboring layers (called *active* S-boxes). A useful metric in this context is the *branch number* of M.

Definition 2. *The branch number of a linear transformation M is defined as*

$$\mathcal{B} = \min_{\Gamma_Y \neq 0} [\mathrm{w_h}(\Gamma_Y) + \mathrm{w_h}(M^\mathsf{T}\Gamma_Y)]\,,$$

where $\mathrm{w_h}(\Gamma)$ *represents the number of nonzero words in the linear mask* Γ*.*

The definition above implies that any linear characteristic traversing the structure shown in Fig. 1 activates at least $\mathcal{B}$ S-boxes. The total number of active S-boxes throughout the cipher multiplied by the maximal correlation over a single S-box gives an upper bound for the correlation of the characteristic.

The straightforward way to minimize this upper bound is to maximize the branch number $\mathcal{B}$. It is easy to see that $\mathcal{B}$ cannot exceed $m + 1$, with m the number of words per block. Matrices M that satisfy this bound (known as the Singleton bound) can be derived from the generator matrices of maximum distance separable (MDS) block codes.

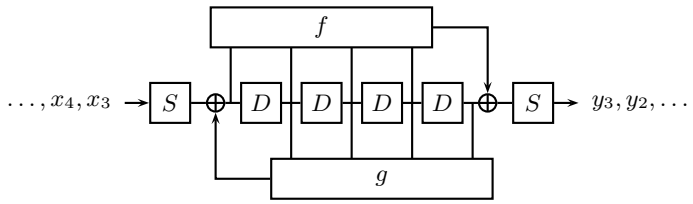


Fig. 2. Stream equivalent of Fig. 1

Large MDS matrices are expensive to implement, though. Therefore, it is often more efficient to use smaller matrices, with a relatively low branch number, and to connect them in such a way that linear patterns with a small number of active S-boxes cannot be chained together to cover the complete cipher. This was the approach taken by the designers of Rijndael [6].

4 From Blocks to Streams

In this section, we try to adapt the concepts described above to a system where the data is not processed in blocks, but rather as a stream.

Since data enters the system one word at a time, each layer of S-boxes in Fig. 1 can be replaced by a single S-box which substitutes individual words as they arrive. A general mth-order linear filter can take over the task of the diffusion matrix. The new system is represented in Fig. 2, where D denotes the delay operator (usually written as z^{-1} in signal processing literature), and f and g are linear functions.

4.1 Polynomial Notation

Before analyzing the properties of this construction, we introduce some notations. First, we adopt the common convention to represent streams of words $x_0, x_1, x_2, \ldots$ as polynomials with coefficients in the finite field:

$$x(D) = x_0 + x_1 D + x_2 D^2 + \ldots .$$

The rationale for this representation is that it simplifies the expression for the input/output relation of the linear filter, as shown in the following equation:

$$y(D) = \frac{f(D)}{g(D)} \cdot \big[x(D) + x^0(D)\big] + y^0(D)\,. \tag{1}$$

The polynomials f and g describe the feedforward and feedback connections of the filter. They can be written as

$$\begin{aligned} f(D) &= D^m \cdot \left(f_m D^{-m} + \cdots + f_1 D^{-1} + 1\right)\,, \\ g(D) &= 1 + g_1 D + g_2 D^2 + \cdots + g_m D^m\,. \end{aligned}$$

The Laurent polynomials x^0 and y^0 represent the influence of the initial state s^0, and are given by $x^0 = D^{-m} \cdot \left(s^0 \cdot g \bmod D^m\right)$ and $y^0 = D^{-m} \cdot \left(s^0 \cdot f \bmod D^m\right)$.

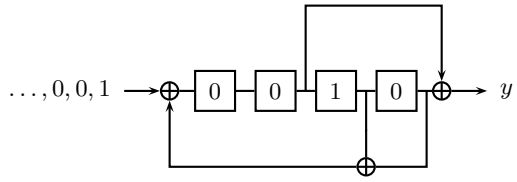


Fig. 3. A 4th-order linear filter

Example 1. The 4th-order linear filter depicted in Fig. 3 is specified by the polynomials $f(D) = D^4 \cdot (D^{-2} + 1)$ and $g(D) = 1 + D^3 + D^4$. Suppose that the delay elements are initialized as shown in the figure, i.e., $s^0(D) = D$. Knowing s^0, we can compute $x^0(D) = D^{-3}$ and $y^0(D) = D^{-1}$. Finally, using (1), we find the output stream corresponding to an input consisting, for example, of a single 1 followed by 0's (i.e., $x(D) = 1$):

$$\begin{aligned} y(D) &= \frac{D^{-1} + D + D^2 + D^4}{1 + D^3 + D^4} + D^{-1} \\ &= D + D^3 + D^5 + D^6 + D^7 + D^8 + D^{12} + D^{15} + D^{16} + D^{18} + \dots \end{aligned}$$

4.2 Linear Correlations

In order to study correlations in a stream-oriented system we need a suitable way to manipulate linear combinations of bits in a stream. It will prove convenient to represent them as follows:

$$\mathrm{Tr}\left[[\gamma_x(D^{-1}) \cdot x(D)]_0\right] .$$

The operator $[\cdot]_0$ returns the constant term of a polynomial, and $\mathrm{Tr}(\cdot)$ denotes the trace to GF(2). The coefficients of γ_x, called *selection polynomial*, specify which words of x are involved in the linear combination. In order to simplify expressions later on we also introduce the notation $\gamma^*(D) = \gamma(D^{-1})$. The polynomial γ^* is called the *reciprocal* polynomial of γ.

As before, the correlation between x and y for a given pair of selection polynomials is defined as

$$c = 2 \cdot \frac{|\{(x, s^0) \mid \mathrm{Tr}[[\gamma_x^* \cdot x]_0] = \mathrm{Tr}[[\gamma_y^* \cdot y]_0]\}|}{|\{(x, s^0)\}|} - 1 .$$

4.3 Propagation of Selection Polynomials

Let us now analyze how correlations propagate through the linear filter. For each selection polynomial γ_x at the input, we would like to determine a polynomial γ_y at the output (if it exists) such that the corresponding linear combinations are perfectly correlated, i.e.,

$$\mathrm{Tr}[[\gamma_x^* \cdot x]_0] = \mathrm{Tr}[[\gamma_y^* \cdot y]_0], \quad \forall x, s^0 .$$

If this equation is satisfied, then this is still be the case after replacing x by $x' = x + x^0$ and y by $y' = y + y^0$, since x^0 and y^0 only consist of negative powers, none of which can be selected by γ_x or γ_y. Substituting (1), we find

$$\mathrm{Tr}[[\gamma_x^* \cdot x']_0] = \mathrm{Tr}[[\gamma_y^* \cdot f/g \cdot x']_0], \quad \forall x, s^0 ,$$

which implies that $\gamma_x^* = \gamma_y^* \cdot f/g$. In order to get rid of negative powers, we define $f^\star = D^m \cdot f^*$ and $g^\star = D^m \cdot g^*$ (note the subtle difference between both stars), and obtain the equivalent relation

$$\gamma_y = g^\star / f^\star \cdot \gamma_x \,. \tag{2}$$

Note that neither of the selection polynomials γ_x and γ_y can have an infinite number of nonzero coefficients (if it were the case, the linear combinations would be undefined). Hence, they have to be of the form

$$\gamma_x = q \cdot f^\star / \gcd(f^\star, g^\star) \quad \text{and} \quad \gamma_y = q \cdot g^\star / \gcd(f^\star, g^\star) \,, \tag{3}$$

with $q(D)$ an arbitrary polynomial.

Example 2. For the linear filter in Fig. 3, we have that $f^\star(D) = 1 + D^2$ and $g^\star(D) = D^4 \cdot (D^{-4} + D^{-3} + 1)$. In this case, $f^\star$ and $g^\star$ are coprime, i.e., $\gcd(f^\star, g^\star) = 1$. If we arbitrarily choose $q(D) = 1 + D$, we obtain a pair of selection polynomials

$$\gamma_x(D) = 1 + D + D^2 + D^3 \quad \text{and} \quad \gamma_y(D) = 1 + D^2 + D^4 + D^5 \,.$$

By construction, the corresponding linear combinations of input and output bits satisfy the relation

$$\mathrm{Tr}(x_0 + x_1 + x_2 + x_3) = \mathrm{Tr}(y_0 + y_2 + y_4 + y_5), \quad \forall x, s^0 \,.$$

4.4 Branch Number

The purpose of the linear filter, just as the diffusion layer of a block cipher, will be to force linear characteristics to pass through as many active S-boxes as possible. Hence, it makes sense to define a branch number here as well.

Definition 3. *The branch number of a linear filter specified by the polynomials f and g is defined as*

$$\begin{aligned} \mathcal{B} &= \min_{\gamma_x \neq 0} [\mathrm{w_h}(\gamma_x) + \mathrm{w_h}(g^\star / f^\star \cdot \gamma_x)] \\ &= \min_{q \neq 0} [\mathrm{w_h}(q \cdot f^\star / \gcd(f^\star, g^\star)) + \mathrm{w_h}(q \cdot g^\star / \gcd(f^\star, g^\star))] \,, \end{aligned}$$

where $\mathrm{w_h}(\gamma)$ represents the number of nonzero coefficients in the selection polynomial γ.

From this definition we immediately obtain the following upper bound on the branch number

$$\mathcal{B} \leq \mathrm{w_h}(f^\star) + \mathrm{w_h}(g^\star) \leq 2 \cdot (m+1) \,. \tag{4}$$

Filters for which this bound is attained can be derived from MDS convolutional $(2, 1, m)$-codes [7]. For example, one can verify that the 4th-order linear filter over GF(2^8) with

$$\begin{aligned} f(D) &= D^4 \cdot \left(\mathtt{02}_x D^{-4} + D^{-3} + D^{-2} + \mathtt{02}_x D^{-1} + 1 \right) \,, \\ g(D) &= 1 + \mathtt{03}_x D + \mathtt{03}_x D^2 + D^3 + D^4 \,, \end{aligned}$$

has a branch number of 10. Note that this example uses the same field polynomial as Rijndael, i.e., $x^8 + x^4 + x^3 + x + 1$.

5 Constructing a Key Stream Generator

In the previous section, we introduced S-boxes and linear filters as building blocks, and presented some tools to analyze how they interact. Our next task is to determine how these components can be combined into a key stream generator. Again, block ciphers will serve as a source of inspiration.

5.1 Basic Construction

A well-known way to construct a key stream generator from a block cipher is to use the cipher in output feedback (OFB) mode. This mode of operation takes as input an initial data block (called initial value or IV), passes it through the block cipher, and feeds the result back to the input. This process is iterated and the consecutive values of the data block are used as key stream. We recall that the block cipher itself typically consists of a sequence of rounds, each comprising a layer of S-boxes and a linear diffusion transformation.

By taking the very same approach, but this time using the stream cipher components presented in Sect. 4, we obtain a construction which, in its simplest form, might look like Fig. 4(a). The figure represents a key stream generator

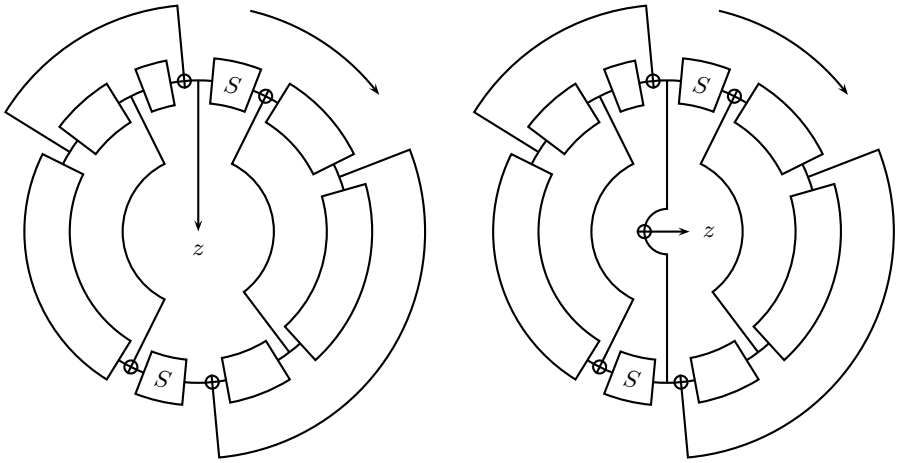


Fig. 4. Two-round key stream generators

consisting of two 'rounds', where each round consists of an S-box followed by a very simple linear filter. Data words traverse the structure in clockwise direction, and the output of the second round, which also serves as key stream, is fed back to the input of the first round.

While the scheme proposed above has some interesting structural similarities with a block cipher in OFB mode, there are important differences as well. The most fundamental difference comes from the fact that linear filters, as opposed to diffusion matrices, have an internal state. Hence if the algorithm manages to keep this state (or at least parts of it) secret, then this eliminates the need for a separate key addition layer (another important block cipher component, which we have tacitly ignored so far).

5.2 Analysis of Linear Characteristics

As stated before, the primary goal in this paper is to construct a scheme which generates a stream of seemingly uncorrelated bits. More specifically, we would like the adversary to be unable to detect any correlation between linear combinations of bits at different positions in the key stream. In the following sections, we will see that the study of linear characteristics provides some guidance on how to design the components of our scheme in order to reduce the magnitude of these correlations.

Applying the tools from Sect. 4 to the construction in Fig. 4(a), we can easily derive some results on the existence of low-weight linear characteristics. The term 'low-weight' in this context refers to a small number of active S-boxes. Since we are interested in correlations which can be detected by an adversary, we need both ends of the characteristic to be accessible from the key stream. In order to construct such characteristics, we start with a selection polynomial γ_u at the input of the first round, and analyze how it might propagate through the cipher.

First, the characteristic needs to cross an S-box. The S-box preserves the positions of the non-zero coefficients of γ_u, but might modify their values. For now, however, let us only consider characteristics for which the values are preserved as well. Under this assumption and using (2), we can compute the selection polynomials γ_v and γ_w at the input and the output of the second round:

$$\gamma_v = g_1^\star / f_1^\star \cdot \gamma_u \quad \text{and} \quad \gamma_w = g_2^\star / f_2^\star \cdot \gamma_v \,.$$

Since all three polynomials γ_u, γ_v, and γ_w need to be finite, we have that

$$\gamma_u = q \cdot f_1^\star f_2^\star / d \,, \quad \gamma_v = q \cdot g_1^\star f_2^\star / d \,, \quad \text{and} \quad \gamma_w = q \cdot g_1^\star g_2^\star / d \,,$$

with $d = \gcd(f_1^\star f_2^\star, g_1^\star f_2^\star, g_1^\star g_2^\star)$ and q an arbitrary polynomial. Note that since both γ_u and γ_w select bits from the key stream z, they can be combined into a single polynomial $\gamma_z = \gamma_u + \gamma_w$.

The number of S-boxes activated by a characteristic of this form is given by $\mathcal{W} = \mathrm{w_h}(\gamma_u) + \mathrm{w_h}(\gamma_v)$. The minimum number of active S-boxes over this set of characteristics can be computed with the formula

$$\mathcal{W}_{\min} = \min_{q \neq 0} [\mathrm{w_h}(q \cdot f_1^\star f_2^\star / d) + \mathrm{w_h}(q \cdot g_1^\star f_2^\star / d)] \,,$$

from which we derive that

$$\mathcal{W}_{\min} \leq \mathrm{w_h}(f_1^\star f_2^\star) + \mathrm{w_h}(g_1^\star f_2^\star) \leq \mathrm{w_h}(f_1^\star) \cdot \mathrm{w_h}(f_2^\star) + \mathrm{w_h}(g_1^\star) \cdot \mathrm{w_h}(f_2^\star) \,.$$

Applying this bound to the specific example of Fig. 4(a), where $\mathrm{w_h}(f_i^\star) = \mathrm{w_h}(g_i^\star) = 2$, we conclude that there will always exist characteristics with at most 8 active S-boxes, no matter where the taps of the linear filters are positioned.

5.3 An Improvement

We will now show that this bound can potentially be doubled by making the small modification shown in Fig. 4(b). This time, each non-zero coefficient in the selection polynomial at the output of the key stream generator needs to propagate to both the upper and the lower part of the scheme. By constructing

linear characteristics in the same way as before, we obtain the following selection polynomials:

$$\gamma_u = q \cdot \frac{f_1^\star f_2^\star + f_1^\star g_2^\star}{d}\,, \quad \gamma_v = q \cdot \frac{f_1^\star f_2^\star + g_1^\star f_2^\star}{d}\,, \quad \text{and} \quad \gamma_z = q \cdot \frac{f_1^\star f_2^\star + g_1^\star g_2^\star}{d}\,,$$

with $d = \gcd(f_1^\star f_2^\star + f_1^\star g_2^\star, f_1^\star f_2^\star + g_1^\star f_2^\star, f_1^\star f_2^\star + g_1^\star g_2^\star)$. The new upper bounds on the minimum number of active S-boxes are given by

$$\begin{aligned}\mathcal{W}_{\min} &\le \mathrm{w_h}(f_1^\star f_2^\star + f_1^\star g_2^\star) + \mathrm{w_h}(f_1^\star f_2^\star + g_1^\star f_2^\star) \\ &\le 2 \cdot \mathrm{w_h}(f_1^\star) \cdot \mathrm{w_h}(f_2^\star) + \mathrm{w_h}(f_1^\star) \cdot \mathrm{w_h}(g_2^\star) + \mathrm{w_h}(g_1^\star) \cdot \mathrm{w_h}(f_2^\star)\,,\end{aligned}$$

or, in the case of Fig. 4(b), $\mathcal{W}_{\min} \le 16$. In general, if we consider extensions of this scheme with r rounds and $\mathrm{w_h}(f_i^\star) = \mathrm{w_h}(g_i^\star) = w$, then the bound takes the form:

$$\mathcal{W}_{\min} \le r^2 \cdot w^r\,. \tag{5}$$

This result suggests that it might not be necessary to use a large number of rounds, or complicated linear filters, to ensure that the number of active S-boxes in all characteristics is sufficiently large. For example, if we take $w = 2$ as before, but add one more round, the bound jumps to 72.

Of course, since the bound we just derived is an upper bound, the minimal number of active S-boxes might as well be much smaller. First, some of the product terms in $f_1^\star f_2^\star + f_1^\star g_2^\star$ or $f_1^\star f_2^\star + g_1^\star f_2^\star$ might cancel out, or there might exist a $q \neq d$ for which $\mathrm{w_h}(\gamma_u) + \mathrm{w_h}(\gamma_v)$ suddenly drops. These cases are rather easy to detect, though, and can be avoided during the design. A more important problem is that we have limited ourselves to a special set of characteristics, which might not necessarily include the one with the minimal number of active S-boxes. However, if the feedback and feedforward functions are sparse, and the linear filters sufficiently large, then the bound is increasingly likely to be tight. On the other hand, if the state of the generator is sufficiently small, then we can perform an efficient search for the lowest-weight characteristic without making any additional assumption.

This last approach allows to show, for example, that the smallest instance of the scheme in Fig. 4(b) for which the bound of 16 is actually attained, consists of two 11th-order linear filters with

$$\begin{aligned} f_1^\star(D) &= 1 + D^{10}\,, & g_1^\star(D) &= D^{11} \cdot (D^{-3} + 1)\,, \\ f_2^\star(D) &= 1 + D^{9}\,, & g_2^\star(D) &= D^{11} \cdot (D^{-8} + 1)\,. \end{aligned}$$

5.4 Linear Characteristics and Correlations

In the sections above, we have tried to increase the number of active S-boxes of linear characteristics. We now briefly discuss how this number affects the correlation of key stream bits. This problem is treated in several papers in the context of block ciphers (see, e.g., [6]).

We start with the observation that the minimum number of active S-boxes $\mathcal{W}_{\min}$ imposes a bound on the correlation c_c of a linear characteristic:

$$c_c^2 \le (c_s^2)^{\mathcal{W}_{\min}}\,,$$

where c_s is the largest correlation (in absolute value) between the input and the output values of the S-box. The squares c_c^2 and c_s^2 are often referred to as *linear*

probability, or also *correlation potential*. The inverse of this quantity is a good measure for the amount of data that the attacker needs to observe in order to detect a correlation.

What makes the analysis more complicated, however, is that many linear characteristics can contribute to the correlation of the same combination of key stream bits. This occurs in particular when the scheme operates on words, in which case there are typically many possible choices for the coefficients of the intermediate selection polynomials describing the characteristic (this effect is called *clustering*). The different contributions add up or cancel out, depending on the signs of c_c. If we now assume that these signs are randomly distributed, then we can use the approach of [6, Appendix B] to derive a bound on the expected correlation potential of the key stream bits:

$$E(c^2) \leq (c_s^2)^{\mathcal{W}_{\min}-n}. \tag{6}$$

The parameter n in this inequality represents the number of degrees of freedom in the choice for the coefficients of the intermediate selection polynomials.

For the characteristics propagating through the construction presented in Sect. 5.3, one will find, in non-degenerate cases, that the values of $n = r \cdot (r-1) \cdot w^{r-1}$ non-zero coefficients can be chosen independently. Hence, for example, if we construct a scheme with $w = 2$ and $r = 3$, and if we assume that it attains the bound given in (5), then we expect the largest correlation potential to be at most $c_s^{2 \cdot 48}$. Note that this bound is orders of magnitude higher than the contribution of a single characteristic, which has a correlation potential of at most $c_s^{2 \cdot 72}$.

Remark 1. In order to derive (6), we replaced the signs of the contributing linear characteristics by random variables. This is a natural approach in the case of block ciphers, where the signs depend on the value of the secret key. In our case, however, the signs are fixed for a particular scheme, and hence they might, for some special designs, take on very peculiar values. This happens for example when $r = 2$, w is even, and all non-zero coefficients of f_i and g_i equal 1 (as in the example at the end of the previous section). In this case, all signs will be positive, and we obtain a significantly worse bound:

$$c^2 \leq (c_s^2)^{\mathcal{W}_{\min}-2 \cdot n}.$$

6 Trivium

In this final section, we present an experimental cipher based on the approach outlined above. Because of space restrictions, we limit ourselves to a very rough sketch of some basic design ideas behind the scheme. The complete specifications of the cipher, which was submitted to the eSTREAM Stream Cipher Project under the name Trivium, can be found at `http://www.ecrypt.eu.org/stream/` [8].

6.1 A Bit-Oriented Design

The main idea of Trivium's design is to turn the general scheme of Sect. 5.3 into a bit-oriented stream cipher. The first motivation is that bit-oriented schemes are typically more compact in hardware. A second reason is that, by reducing the word-size to a single bit, we may hope to get rid of the clustering phenomenon which, as seen in the previous section, has a significant effect on the correlation.

Of course, if we simply apply the previous scheme to bits instead of words, we run into the problem that the only two existing 1×1-bit S-boxes are both linear. In order to solve this problem, we replace the S-boxes by a component which, from the point of view of our correlation analysis, behaves in the same way: an exclusive OR with an external stream of unrelated but biased random bits. Assuming that these random bits equal 0 with probability $(1 + c_s)/2$, we will find as before that the output correlates with the input with correlation coefficient c_s.

The introduction of this artificial 1×1-bit S-box greatly simplifies the correlation analysis, mainly because of the fact that the selection polynomial at the output of an S-box is now uniquely determined by the input. Thanks to this lack of freedom, we neither need to make special assumptions about the values of the non-zero coefficients, nor to consider the effect of clustering: the maximum correlation in the key stream is simply given by the relation

$$c_{\max} = c_s^{\mathcal{W}_{\min}} . \tag{7}$$

The obvious drawback, however, is that the construction now relies on external streams of random bits, which have to be generated somehow. TRIVIUM attempts to achieve this by interleaving three identical key stream generators, where each generator obtains streams of biased bits (with $c_s = 1/2$) by ANDing together state bits of the two other generators. The result is shown in Fig. 5.

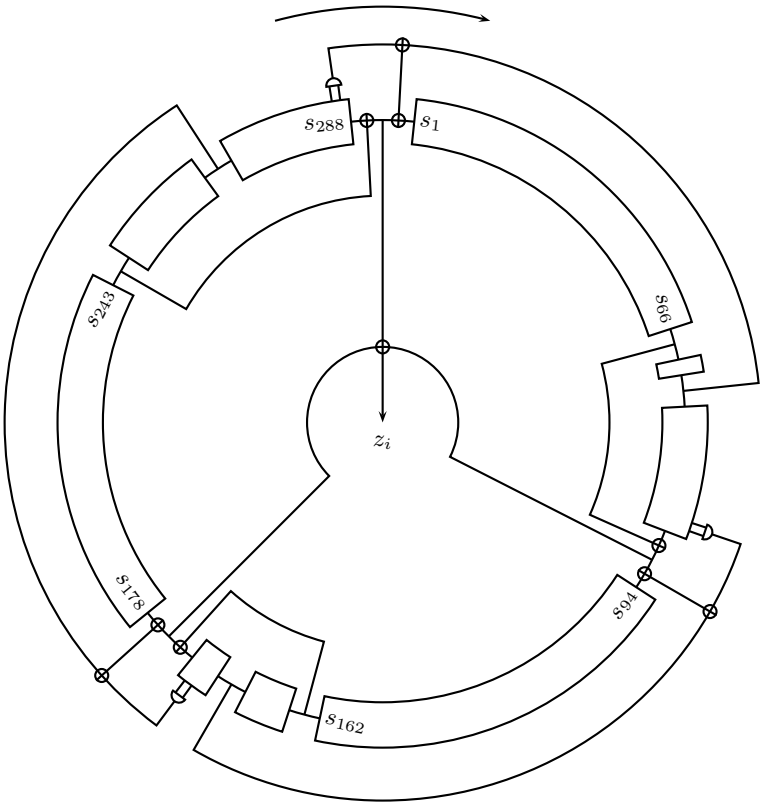


Fig. 5. TRIVIUM

References

1. Daemen, J.: Cipher and hash function design. Strategies based on linear and differential cryptanalysis. PhD thesis, Katholieke Universiteit Leuven (1995)
2. Hawkes, P., Rose, G.G.: Primitive specification and supporting documentation for SOBER-t*w* submission to NESSIE. In: Proceedings of the First NESSIE Workshop, NESSIE (2000)
3. Ekdahl, P., Johansson, T.: SNOW – A new stream cipher. In: Proceedings of the First NESSIE Workshop, NESSIE (2000)
4. Daemen, J., Clapp, C.S.K.: Fast hashing and stream encryption with PANAMA. In Vaudenay, S., ed.: Fast Software Encryption, FSE'98. Volume 1372 of Lecture Notes in Computer Science., Springer-Verlag (1998) 60–74
5. Matsui, M.: Linear cryptanalysis method for DES cipher. In Helleseth, T., ed.: Advances in Cryptology – EUROCRYPT'93. Volume 765 of Lecture Notes in Computer Science., Springer-Verlag (1993) 386–397
6. Daemen, J., Rijmen, V.: The Design of Rijndael: AES — The Advanced Encryption Standard. Springer-Verlag (2002)
7. Rosenthal, J., Smarandache, R.: Maximum distance separable convolutional codes. Applicable Algebra in Engineering, Communication and Computing **10** (1999) 15–32
8. De Cannière, C., Preneel, B.: TRIVIUM — Specifications. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030 (2005) `http://www.ecrypt.eu.org/stream`.

On periods of Edon-$(2m, 2k)$ Family of Stream Ciphers

Danilo Gligoroski[1,2], Smile Markovski[2], and Svein Johan Knapskog[1]

[1] Centre for Quantifiable Quality of Service in Communication Systems, Norwegian University of Science and Technology, O.S.Bragstads plass 2E, N-7491 Trondheim, NORWAY

[2] "Ss Cyril and Methodius" University
Faculty of Natural Sciences and Mathematics, Institute of Informatics
P.O.Box 162, 1000 Skopje,
Republic of MACEDONIA

gligoroski@yahoo.com, smile@ii.edu.mk, Svein.J.Knapskog@Q2S.ntnu.no

Abstract. Modularity of the design of Edon80 stream cipher allows us to define a family of stream ciphers Edon-$(2m, 2k)$ where the value $2m$ is the number of internal quasigroup transformations and $2k$ is the bit size of the key. That allows us further to derive the distribution of the periods of the keystreams produced by every stream cipher in that family. We show that the obtained distribution is LogNormal when $m \to \infty$. Having a formula for that distribution, we can compute the parameter m for every combination of key and IV sizes such that Edon-$(2m, 2k)$ will meet any predetermined security criteria. [3]

Key words: hardware, synchronous stream cipher, Latin square, quasigroup, quasigroup string processing

1 Introduction

In this paper we derive a formula for the distribution of the periods of the proposed stream cipher Edon80 as well as for a family of Edon-$(2m, 2k)$ stream ciphers to which Edon80 belongs. We have initially announced this result in our response [1] to the remarks of Hong given in [2]. Here we give a precise analysis and a precise formula for computing the distribution of the periods of the keystreams for the Edon-$(2m, 2k)$ family.

Although all stream ciphers proposed for the eSTREAM project have given the expected periods of their keystreams, very few of them have precise analysis and strong mathematical claims for the produced keystream periods. Beside the security scalability that does not influence the speed performance of the Edon80

[3] This work was carried out during the tenure of an ERCIM fellowship of D. Gligoroski visiting Q2S - Centre for Quantifiable Quality of Service in Communication Systems at Norwegian University of Science and Technology - Trondheim, Norway.

(when realized in hardware), we think that having such a precise mathematical description of the periods of its keystreams is one of the strongest points compared to the other eSTREAM submissions.

The paper is organized as follows: In Section 2 we derive a precise mathematical model and precise mathematical expressions for the probabilities of the keystream periods, in Section 3 we discuss two security criteria and how Edon80 or Edon-$(2m, 2k)$ can meet them, and in Section 4 we give the conclusions.

2 Probabilistic model for the periods produced by Edon-$(2m, 2k)$ stream ciphers

Here we will give a brief description of Edon80. For a detailed description see [3]. Edon80 uses 4 quasigroups of order 4 (shown in Table 1) that process the initial string consisting of letters "0 1 2 3 0 1 2 3 0 ..." in 80 steps and output every second letter that forms the keystream of the stream cipher (see Table 2). The processing in every step is done by a quasigroup $*_i$ and a leader a_i, $i = 0, \ldots, 79$ chosen in the IVSetup process that have the property to map the initial 80-bit key (40 2-bit letters) and initial 64-bit IV (32 2-bit letters) equiprobable in the space $\{0, 1, 2, 3\}^{80}$.

$\bullet_0$	0	1	2	3
0	0	2	1	3
1	2	1	3	0
2	1	3	0	2
3	3	0	2	1

$\bullet_1$	0	1	2	3
0	1	3	0	2
1	0	1	2	3
2	2	0	3	1
3	3	2	1	0

$\bullet_2$	0	1	2	3
0	2	1	0	3
1	1	2	3	0
2	3	0	2	1
3	0	3	1	2

$\bullet_3$	0	1	2	3
0	3	2	1	0
1	1	0	3	2
2	0	3	2	1
3	2	1	0	3

Table 1. Quaigroups used for the design of *Edon80*

$*_i$		0	1	2	3	0	1	2	3	0	. .
$*_0$	a_0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$	$a_{0,8}$	. .
$*_1$	a_1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$	$a_{1,8}$	. .
.	.	.	.	.	.	.	.	.	.	.	. .
$*_{79}$	a_{79}	$a_{79,0}$	$a_{79,1}$	$a_{79,2}$	$a_{79,3}$	$a_{79,4}$	$a_{79,5}$	$a_{79,6}$	$a_{79,7}$	$a_{79,8}$	. .

Table 2. Representation of quasigroup string e-transformations of *Edon80* during the *Keystream* mode

In what follows we will describe the mathematical probabilistic model that explains the distribution of the periods obtained by quasigroup string transformations like those used in Edon80.

For that purpose we need the following definitions:

Definition 1. (*Quasigroup*) *A quasigroup is a groupoid* $(Q, *)$ *satisfying the laws*

$$(\forall u, v \in Q)(\exists x, y \in Q)(u * x = v,\ y * u = v),$$

$$x * y = x * z \implies y = z,\ y * x = z * x \implies y = z.$$

Definition 2. (*Quasigroup String Transformations*) *For a finite set* Q *let us denote by* Q^+ *the set of all nonempty words* (*i.e. finite strings*) *formed by the elements of* Q. *Let the elements of* Q^+ *be denoted by* $\alpha = a_1 a_2 \dots a_n$ *where* $a_i \in Q$. *Let* $*$ *be a quasigroup operation on the set* Q. *For each* $l \in Q$ *the function* $e_{l,*} : Q^+ \to Q^+$, *called the e-transformation based on the operation* $*$ *with leader* l, *is defined as follows:*

$$e_{l,*}(\alpha) = b_1 \dots b_n \iff b_{i+1} = b_i * a_{i+1} \tag{1}$$

for each $i = 0, 1, \dots, n-1$, *where* $b_0 = l$.

Definition 3. (*Period of a string*) *The string* $\alpha = a_1 a_2 \dots a_n \in Q^+$, *where* $a_i \in Q$, *has a period* p *if* p *is the smallest positive integer such that* $a_{i+1} a_{i+2} \dots a_{i+p} = a_{i+p+1} a_{i+p+2} \dots \ \dots a_{i+2p}$ *for each* $i \geq 0$.

Definition 4. (*Edon-*$(2m, 2k)$) *Let* Key *be a* $2k$ *bit string represented as a string of* k 2*-bit letters, i.e.* $Key = K_0 K_1 \cdots K_{k-1}$. *For every* $m \in \mathbb{N},\ m \geq k$, *let* $q = 2m - k$ *be the length of the string* $Const$, *i.e.* $Const = c_0 c_1 \cdots c_{q-1}$. *Let* $S_0 = s_0 s_1 \dots s_{2m-1}$ *be a concatenation of the strings* Key *and* $Const$ *i.e.* $S_0 = Key || Const$.

Let us assign $2m$ *working quasigroups by the following formula:*

$$(Q, *_i) \leftarrow (Q, \bullet_{K_{i \bmod k}}),\quad 0 \leq i \leq 2m-1,$$

and assign $2m$ *leaders by the following formula:*

$$t_i = s_{2m-1-i},\quad 0 \leq i \leq 2m-1.$$

Let us perform $2m$ *e-transformations on the string* S_0 *with quasigroups* $*_i$ *and leaders* t_i, $0 \leq i < 2m$, *i.e.*

$$S_{i+1} = e_{*_i, t_i}(S_i),\quad 0 \leq i \leq 2m-1,$$

and let $S_{2m} = a_0 a_1 \dots a_{2m-1}$.

Let us denote by $\Gamma_0 =$ "0 1 2 3 0 1 2 3 0 ..." *the infinite sting consisting of infinite concatenations of the substrings* "0 1 2 3".

Let us perform $2m$ *e-transformations on the string* Γ_0 *with quasigroups* $*_i$ *and leaders* a_i, $0 \leq i < 2m$, *i.e.*

$$\Gamma_{i+1} = e_{*_i, a_i}(\Gamma_i),\quad 0 \leq i \leq 2m-1,$$

and let $Keystream = \Gamma_{2m}|_{2i}$ *where operator* $|_{2i}$ *means that* $Keystream$ *consists of every second letter of* Γ_{2m}.

The particular definition of Edon80 will be equivalent to our definition of Edon-$(80, 80)$ if we put $m = 40$, $k = 40$ and thus $q = 40$, and in the string $Const = c_0c_1 \cdots c_{39}$, we put $IV = c_0c_1 \ldots c_{31}$ and we fix $c_{32}c_{33} \ldots c_{39} \equiv 3\ 2\ 1\ 0\ 0\ 1\ 2\ 3$.

Definition 5. (*Keystream periods of Edon-*$(2m, 2k)$ *stream ciphers seen as stochastic process*) *Let* Ξ *be a stochastic process* $\{X_i\}$ *defined as a family of random variables indexed by a parameter* i*. Further, let every* X_i *have its own distribution over the sample space* Ω *where the values of* Ω *denote how many times the period* p_{Γ_i} *of the string* Γ_i *is larger then the period* $p_{\Gamma_{i-1}}$ *of the string* Γ_{i-1}*.*

Theorem 1. (*Ever non-decreasing periodicity of quasigroup string transformations*) *Let* $(Q, *)$ *be a quasigroup of order* r*, let* $\Gamma \in Q^*$ *be an infinite string with period* p *and let* $\Gamma' = e_{*,l}(\Gamma)$ *have a period* p'*. Then* $p'/p \in \{1, 2, \ldots, r\}$*.*

The proof of the Theorem 1 is given in the appendix of FSE 2005 paper [4].

Simple exhaustive investigation of all choices for all of the four quasigroups and for each case an investigation of all four possibilities for choosing the leader gives the following distribution of X_1.

Lemma 1. *The distribution of* X_1 *is* $\begin{pmatrix} 1 & 2 & 3 & 4 \\ \frac{1}{8} & \frac{1}{2} & \frac{3}{8} & 0 \end{pmatrix}$. □

Further, we will assume stationarity of the defined stochastic process by the following assumption:

Assumption 1 *The stochastic process* $\Xi \equiv \{X_i\}, i = 1, 2, 3, \ldots$ *of discrete random variables* X_i *converge to a stationary distribution* $X = \begin{pmatrix} 1 & 2 & 3 & 4 \\ \frac{1}{4} & \frac{1}{4} & \frac{11}{32} & \frac{5}{32} \end{pmatrix}$.

It is easy to verify that $\mu = E(X) = \frac{77}{32}$, $\sigma^2 = Var(X) = \frac{1079}{1024}$.

Theorem 2. *If* Y_{2m} *is a random variable describing the period of Edon-*$(2m, 2k)$ *then, when* $m \to \infty$*, its cumulative density function can be approximated by the continuous function:*

$$F_{Y_{2m}}(y) = \frac{1}{2}\left(1 + \texttt{erf}\left(\frac{1.00777\ (\ln(2y) - 1.535086\ m)}{\sqrt{m}}\right)\right), \quad 0 < y < \infty,$$

with expectation

$$E(Y_{2m}) = \frac{1}{2}e^{1.78125\ m}$$

and variance

$$Var(Y_{2m}) = \frac{1}{4}e^{3.5625\ m}(e^{0.492324\ m} - 1).$$

Proof. As a consequence from Theorem 1 it follows that every application of an e-transformation in a cipher like Edon-$(2m, 2k)$ can be seen as a random variable receiving values from the set $\{1, 2, 3, 4\}$. Since Edon-$(2m, 2k)$ has $2m$ e-transformations, we have $2m$ random variables $X_1, X_2, \ldots, X_{2m}$ (that can be treated as statistically independent under the assumption that one-way *IVSetup* procedure is well defined and maps the initial $2k$ bits of the *Key* and $2q$ bits of the string *Const* without bias into $4m$ bits, i.e. into $2m$ 2-bit letters).

Let us first compute the distribution of the periods of the string Γ_{2m}. If we denote by Z_{2m} the random variable that describes the periods of the string Γ_{2m}, then Z_{2m} can be seen as a product of $2m$ independent random variables X_i, i.e. $Z_{2m} = X_1 X_2 \cdots X_{2m}$. The most important task is to find the distribution of the variables $X_i,\ i = 1, \ldots, 2m$. If we take into account the Assumption 1 then we can assume that (although there is a transition period for the distribution of the first several $X_i, 1 \leq i \leq 16$), if the number of applied transformations $2m$ is large (for example $2m > 40$) then we can compute the distribution of the multiplication of $2m$ i.i.d. r.v. and that distribution will be close to the actual distribution of Z_{2m}.

After numerous numerical experiments of performing e-transformations on the strings obtained in Edon-$(2m, 2k)$ stream ciphers, we have found the numerical values for the distributions of the random variables $X_i, i = 1, 2, \ldots, 16$, which are clearly supporting Assumption 1 and they are shown in Table 3.

i	X_i	i	X_i
1	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ \frac{1}{8} & \frac{1}{2} & \frac{3}{8} & 0 \end{pmatrix}$	9	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2505 & 0.2510 & 0.3416 & 0.1569 \end{pmatrix}$
2	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.1485 & 0.1875 & 0.3522 & 0.3118 \end{pmatrix}$	10	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2503 & 0.2536 & 0.3397 & 0.1564 \end{pmatrix}$
3	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2369 & 0.3355 & 0.2539 & 0.1738 \end{pmatrix}$	11	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2502 & 0.2510 & 0.3407 & 0.1581 \end{pmatrix}$
4	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2536 & 0.2661 & 0.3115 & 0.1688 \end{pmatrix}$	12	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2516 & 0.2461 & 0.3445 & 0.1577 \end{pmatrix}$
5	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2457 & 0.2512 & 0.3448 & 0.1584 \end{pmatrix}$	13	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2479 & 0.2524 & 0.3429 & 0.1568 \end{pmatrix}$
6	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2498 & 0.2484 & 0.3457 & 0.1561 \end{pmatrix}$	14	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2500 & 0.2502 & 0.3421 & 0.1577 \end{pmatrix}$
7	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2474 & 0.2518 & 0.3432 & 0.1576 \end{pmatrix}$	15	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2538 & 0.2515 & 0.3378 & 0.1569 \end{pmatrix}$
8	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0.2488 & 0.2493 & 0.3451 & 0.1568 \end{pmatrix}$	16	$\begin{pmatrix} 1 & 2 & 3 & 4 \\ \frac{1}{4} & \frac{1}{4} & \frac{11}{32} & \frac{5}{32} \end{pmatrix}$

Table 3. The distribution of the random variables X_i for the first 16 values of i.

Since

$$Z_{2m} = X_1 X_2 \cdots X_{2m}$$

we can apply ln on both sides and obtain:

$$\ln(Z_{2m}) = \ln(X_1) + \ln(X_2) + \cdots \ln(X_{2m}).$$

If we assume that all X_i has the same distribution as the discrete random variable X (Assumption 1), then they have the same mean $\mu_X = \frac{77}{32}$ and the same variance $\sigma_X^2 = \frac{1079}{1024}$. Then, the random variable $W = \ln(X)$ has a mean $\mu_W = E(W) \approx 0.767543$ and a variance $\sigma_W^2 = Var(W) \approx 0.246162$. Thus, the sum of $2m$ random variables $S_{2m} = \sum_{i=1}^{2m} \ln(X_i) = \sum_{i=1}^{2m} W_i$, as a consequence of the Central Limit Theorem, will have a normal distribution with mean $\mu_{S_{2m}} \approx 2m\mu_W \approx 1.535086\ m$ and $\sigma_{S_{2m}}^2 \approx 2m\sigma_W^2 \approx 0.492324\ m$. Now, having $Z_{2m} = e^{S_{2m}}$ and S_{2m} being the normal distribution $\mathcal{N}(1.535086\ m, 0.492324\ m)$ we can compute the pdf of Z_{2m} (the so called LogNormal Distribution) by the following formula (found in many introductory probability textbook - see for example [5]):

$$f_{Z_{2m}}(z) = \frac{1}{z\sqrt{0.492324\ m}\sqrt{2\pi}} \exp\left(-\frac{(\ln(z) - 1.535086\ m)^2}{2 \times 0.492324\ m}\right), \quad 0 < z < \infty,$$

that by a little simplification will take the form:

$$f_{Z_{2m}}(z) = \frac{1}{0.701658\ z\sqrt{2\pi m}} \exp\left(-\frac{(\ln(z) - 1.535086\ m)^2}{0.984648\ m}\right), \quad 0 < z < \infty.$$

The formulas for computing the mean $E(Z_{2m})$ and the variance $Var(Z_{2m})$ can be found also in [5]:

$$E(Z_{2m}) = e^{1.78125\ m}, \quad Var(Z_{2m}) = e^{3.5625\ m}(e^{0.492324\ m} - 1).$$

If we bear in mind that $Y_{2m} = \frac{1}{2}Z_{2m}$ (because the keystream of Edon-$(2m, 2k)$ consists of every second letter from the string Γ_{2m}) we have that pdf, mean and variance for Y_{2m} can be computed as $f_{Y_{2m}}(y) = 2f_{Z_{2m}}(2y)$, $E(Y_{2m}) = \frac{1}{2}E(Z_{2m})$ and $Var(Y_{2m}) = \frac{1}{4}Var(Z_{2m})$, i.e.

$$f_{Y_{2m}}(y) = \frac{1}{1.40332\ y\sqrt{2\pi m}} \exp\left(-\frac{(\ln(2y) - 1.535086\ m)^2}{0.984648\ m}\right), \quad 0 < y < \infty, \tag{2}$$

$$E(Y_{2m}) = \frac{1}{2}e^{1.78125\ m}, \quad Var(Y_{2m}) = \frac{1}{4}e^{3.5625\ m}(e^{0.492324\ m} - 1).$$

From the obtained pdf for Y_{2m} we can easily compute the cumulative density function as:

$$F_{Y_{2m}}(y) = \frac{1}{2}\left(1 + \texttt{erf}\left(\frac{1.00777\ (\ln(2y) - 1.535086\ m)}{\sqrt{m}}\right)\right), \quad 0 < y < \infty. \tag{3}$$

□

We have derived equation (3) as a useful tool when designing Edon-$(2m, 2k)$ stream ciphers that will satisfy different security requirements as we will see in the next section. However, we have to note that, since we have approximated a discrete random variable Y_{2m} by a continuous function in (3), it makes no sense to use a continuous pdf equation (2) for computing probabilities for obtaining a specific period. For example, Edon-$(2m, 2k)$ does not produce periods of length $2^{16} + 1$ and so the actual probability for obtaining such a period in the discrete case is 0, but the pdf equation (2) gives some positive probability. On the other hand, the approximations made by (3) are satisfactory and in fact are guaranteed by the Central Limit Theorem. In Figure 1 we show the results of our simulation for Edon-$(16, 16)$. The red line is obtained by equation (3) and the green one is obtained by making exhaustive search changing all 2^{16} values for the Key.

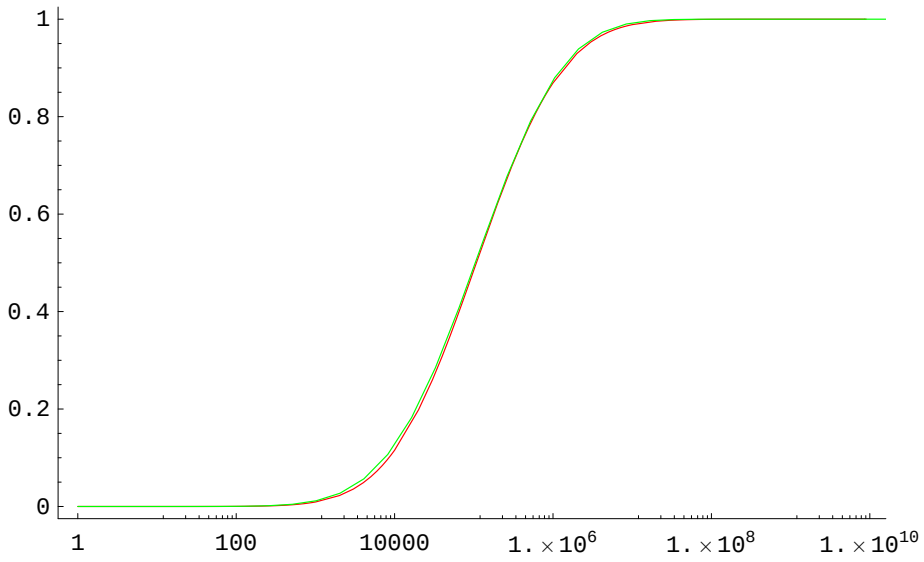


Fig. 1. Comparison between our mathematical model and concrete experimental results for the periods of Edon-(16,16). The red line represents values from the model and green line represents obtained results after exhaustive search for all 2^{16} keys for Edon-$(16, 16)$.

It is a relatively simple iterative procedure to numerically obtain the cdf and pdf for a concrete discrete random variable Y_{2m} (without approximation by continuous functions) and in Figure 2 and Figure 3 we show our experimentally obtained cdf's compared with cdf's that are obtained by equation (3) for Edon-$(80, 80)$ and Edon-$(160, 80)$.

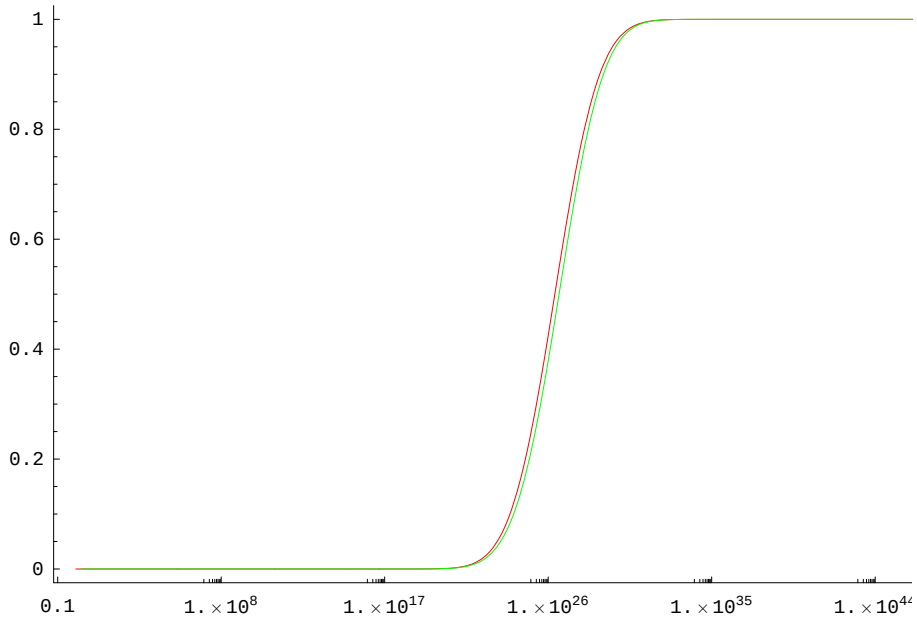


Fig. 2. Comparison between our mathematical model and concrete experimental results for the cumulative distribution of the periods of Edon-(80,80). The red line represents values from the model and green line represents experimentally obtained discrete distribution.

3 How Edon-$(2m, 2k)$ meets different security criteria

In this section we would like to give an answer to the questions: "Is there a weak key attack on Edon80?" and "Is the design of Edon-$(2m, 2k)$ adaptable to more demanding security criteria?". For that purpose let us recall briefly the security criteria that were posted for the eSTREAM - ECRYPT Stream Cipher Project. During the initial phase of the project the security criteria for hardware and software stream ciphers were set and announced formally as follows:

- *Any key-recovery attack (including time-memory-data tradeoff attacks) should be at least as difficult as exhaustive search.*
- *Also, distinguishing attacks are likely to be of interest to the cryptographic community. However the relative importance of high complexity distinguishing attacks may become an issue for wider discussion.*
- *Clarity of design is likely to be an important consideration.*

Special attention to the time-memory-data tradeoff attacks has been payed since the publication of Hong-Sarkar paper [6], which resulted in an update of the initial requirements for the size of the key and IV in eSTREAM call for participation (the rationale can be found in Cannière, Lano and Preneel's com-

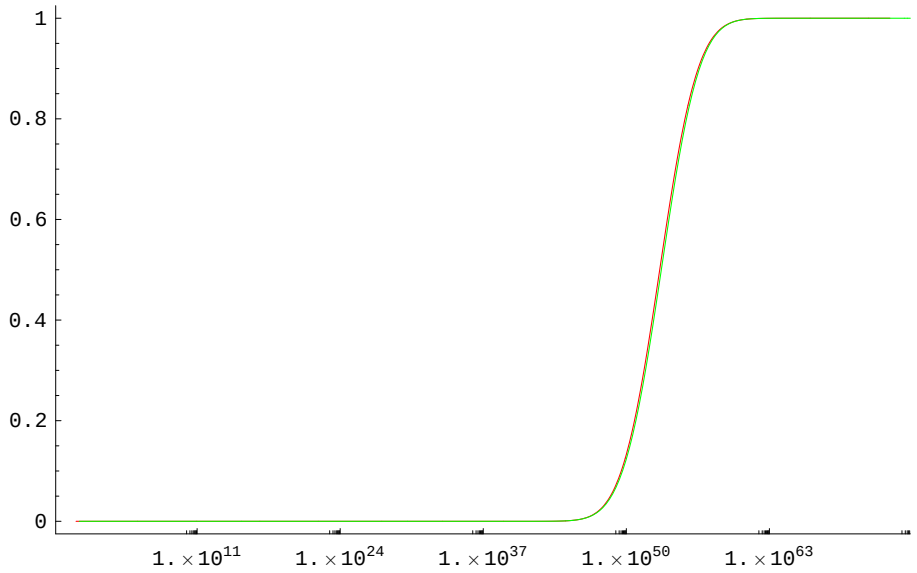


Fig. 3. Comparison between our mathematical model and concrete experimental results for the cumulative distribution of the periods of Edon-(160,80). The red line represents values from the model and green line represents experimentally obtained discrete distribution.

ments to TMD attacks in [7]). However, from many comments on the eSTREAM forum (as well as from the comments of Hong in [2]) it can be concluded that sometimes the workloads that are equivalent to the amount of work of a simple exhaustive key search are not satisfactory as a security criterion (at least as intuitive perception). In particular, that can be said about the distribution of the lengths of the keystream periods.

Since the length of the keystream in Edon-$(2m, 2k)$ stream ciphers depends on the choice of the (Key, IV) pair, we can say that an attack on the cipher when the key stream has short period can be treated as weak key attack. Weak key attacks were successful cryptanalytic tools against IDEA and Lucifer (see for example [8–10]). The basic idea is that if the key consists of $2k$ bits and so the exhaustive search needs 2^{2k} operations, if there is a set of weak keys with volume of $V = 2^f$ and the membership testing procedure whether a key is weak needs 2^w operations, then the complexity of the weak key attack is 2^{2k-f+w}. So if $w - f < 0$ i.e. if $w < f$ then a weak key attack can break the cipher with complexity less then the exhaustive key search.

In the situation for IDEA and Lucifer, the testing procedure was based on differential cryptanalysis and it needed 2^4 and 2^{36} operations respectably. For Edon-$(2m, 2k)$ the membership test whether the length of the keystream is 2^w needs 2^w computations. For Edon-$(2m, 2k)$ we have a "controllable" part in the design that will prevent weak key attack from being effective. This is the value of m in the formula (3). More precisely, we can state the following:

Lemma 2. *For any predetermined and fixed key size $2k$, the minimum number of necessary $2m$ e-transformations in Edon-$(2m, 2k)$ to make weak key attack ineffective can be computed by the following expression:*

$$\min_m \left(\frac{y}{F_{Y_{2m}}(y)} \geq 2^{2m}, \quad \forall y > 0 \right).$$

Proof. The probability that a keystream has a period less than $y = 2^w$ can be expressed as power of 2, i.e. let us denote $F_{Y_{2m}}(y) = 2^{-f}$. Since that probability can be interpreted as a ratio between the number of weak (Key, IV) pairs and the total number of (Key, IV) pairs of size 2^{2m} i.e. $F_{Y_{2m}}(y) = 2^{-f} = V/2^{2m}$ the volume of the weak (Key, IV) pairs can be computed as $V = 2^{2m-f}$. Since the membership test needs $y = 2^w$ operations, the cipher is resistant against a weak key attack if $2^{2k-(2m-f)+w} \geq 2^{2k}$ i.e. if $2^{w+f} \geq 2^{2m}$ which is equivalent with the expression $\frac{y}{F_{Y_{2m}}(y)} \geq 2^{2m}$. □

We have tested whether Edon-(80,80) is vulnerable to a weak key attack and the findings are presented in Figure 4. Edon-(80,80) is not totally resistant against a weak key attack since the minimum of the function $\frac{y}{F_{Y_{80}}(y)}$ is obtained for $y \approx 2^{60.55}$ and the value is $2^{76.89}$ i.e. $60.55 + 76.89 = 137.44$ which is less than 144. Here we want to stress the fact that the search space has the size of 2^{144} and not 2^{160} since in the design of Edon80 we have 16 fixed bits.

A simple tweak with $2m = 84$ e-transformations will result in full resistance against a weak key attack since the minimum of the function $\frac{y}{F_{Y_{84}}(y)}$ is obtained for $y \approx 2^{63.5676}$ and the value is $2^{80.6428}$ i.e. $63.5676 + 80.6428 = 144.21$.

As we mentioned in the beginning of this section, an intuitive requirement for security of a certain stream cipher primitive is as follows:

> *The stream cipher has to have the property that finding a (Key, IV) pair that gives period less then 2^{2k} has probability less then 2^{-2k}. More specifically, the security criterion in this case is:*
>
> $$\forall p < 2k, \quad P[Keystream\ period < 2^p] < 2^{-2k}. \tag{4}$$

For the latest criterion we have the following Lemma:

Lemma 3. *For any predetermined and fixed key size $2k$, the minimum number of necessary $2m$ e-transformations to meet the requirements of the criterion expressed in formula (4) can be computed by the following expression:*

$$F_{Y_{2m}}(y) \leq 2^{-2k}, \quad \forall y \leq 2^{2k}. \qquad \square$$

From the results of the analysis of Edon-(80,80), it is clear that it does not comply with the requirements of security criterion (4).

Although the practical value of the criterion (4) is disputable, since the total workload for practical attacks that will use the noncompliance with it is much

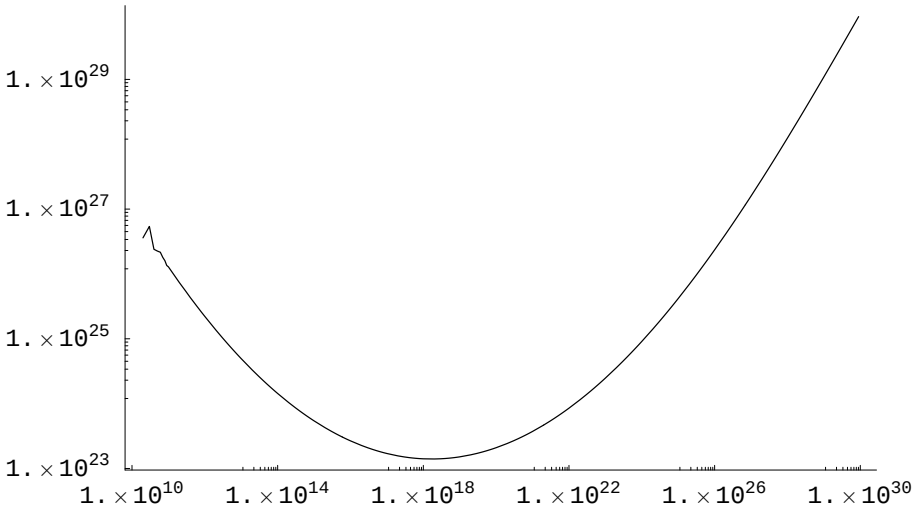


Fig. 4. The log–log plot of the function $\frac{y}{F_{Y_{80}}(y)}$. The minimum is obtained for the periods $y \approx 2^{60.55}$ and the value is $2^{76.89}$.

bigger then exhaustive search, Edon-$(2m, 2k)$ stream ciphers can comply with that criterion. For example, for the key size of 80 bits, Edon-(160,80) meets the requirements of criterion (4) and the probability of obtaining a keystream with period less then 2^{80} is $2^{-86.1351}$.

4 Conclusions

We have built a mathematical probabilistic model by which the periods produced by the family of stream ciphers Edon-$(2m, 2k)$ (where Edon80 belongs) can be modelled. The Edon-$(2m, 2k)$ stream ciphers are based on a solid mathematical background and by increasing the number of rounds we can increase some security aspects of the primitive in a controllable manner. Further, by using the mathematical model we have developed and described in this paper we can build distinct types of stream ciphers with any size of the *key* and IV, that will comply with different types of security requirements, without any loss of operating speed of the cipher.

ACKNOWLEDGMENT

We would like to thank the two anonymous reviewers that gave as very useful comments that improved the quality of the paper - especially Section 3.

References

1. D. Gligoroski, S. Markovski, L. Kocarev, and M. Gušev: Understanding Periods in Edon80, ECRYPT database, July 2005.
2. J. Hong: Remarks on the Period of Edon80, ECRYPT database, June 2005.
3. D. Gligoroski, S. Markovski, L. Kocarev, and M. Gušev: Edon80 - Hardware synchoronous stream cipher. Symmetric Key Encryption Workshop, Århus, Denmark, May, 2005.
4. S. Markovski, D. Gligoroski, and L. Kocarev: Unbiased Random Sequences from Quasigroup String Transformations, in Fast Software Encryption 2005, H. Gilbert and H. Handschuh (Eds.), LNCS 3557, pp. 163-180, 2005.
5. D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, John Wiley & Sons, Inc., ISBN 0-471-20454-4, 2003.
6. J. Hong and P. Sarkar: Rediscovery of Time Memory Tradeoffs, Cryptology ePrint Archive, Report 2005/090.
7. C. De Cannière, J. Lano, and B. Preneel: Comments on the rediscovery of time memory data tradeoffs, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/040, 2005.
8. P. Hawkes: Differential-linear weak key classes of IDEA, in Proceedings of Eurocrypt98 (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 112 126, Springer-Verlag, 1998.
9. I. Ben-Aroya and E. Biham: Differential cryptanalysis of Lucifer, in Advances in Cryptology CRYPTO93 (D. R. Stinson, ed.), vol. 773 of Lecture Notes in Computer Science, pp. 187199, Springer-Verlag, 1993. see also Journal of Cryptology, Vol. 9, No. 1, pp. 2134, 1996.
10. A. Biryukov, web page: http://homes.esat.kuleuven.be/~abiryuko/Enc/c.pdf

CRYPTANALYSIS OF CRYPTMT: EFFECT OF HUGE PRIME PERIOD AND MULTIPLICATIVE FILTER

MAKOTO MATSUMOTO, MUTSUO SAITO, TAKUJI NISHIMURA,
AND MARIKO HAGITA

Abstract. CryptMT (Cryptographic Mersenne Twister) is an 8-bit pseudorandom integer generator for a stream cipher. It combines an $\mathbb{F}_2$-linear generator of period $2^{19937}-1$ and a multiplicative filter with 31-bit memory. We analyze its security against some standard cryptanalytic attacks for filter generators. It is proved that CryptMT has strong resistance against them: CryptMT has a period of $2^{19937}-1$, the correlations among the consecutive 624-bytes of outputs are of order 2^{-19937}, the algebraic degree of the output bits with respect to the bits in Key and IV is expected to be near to the size of Key and IV. The Key size and IV size are variable, up to 2048-bit for each. We claim that CryptMT has the same security level with the minimum of the key size and the IV size. CryptMT is 1.5–2.0 times faster than the optimized AES CTR mode with 256-bit security level.

1. Introduction

In the previous article[16], we proposed an 8-bit-integer pseudorandom number generator Cryptographic Mersenne Twister (CryptMT) for a stream cipher, and FUBUKI block/stream cipher. In this article, we explain the design rationale of CryptMT and analyze its resistance against some standard attacks.

2. Design rationale of CryptMT

CryptMT is a variant of classical filter generators. Conventional method is to use LFSR as a mother generator[1] and to transform its outputs by a nonlinear Boolean function (i.e. without memory) called a filter.

CryptMT adopted Mersenne Twister(MT) as the mother generator and a multiplicative filter with memory, as explained below. Properties of MT stated here are proved in [14].

MT generates a pseudorandom 32-bit integer sequence by the $\mathbb{F}_2$-linear recursion

$$x_{624+i} = x_{397+i} \oplus ((x_i \& \texttt{0x80000000}) | (x_{1+i} \& \texttt{0x7fffffff}))A \quad (i = 0, 1, 2, \ldots).$$

Date: January 23, 2006.

Key words and phrases. Cryptographic Mersenne Twister, CryptMT, SNOW, stream cipher, multiplicative filter, algebraic attack, algebraic degree, correlation attack.

CryptMT stream cipher analyzed in this manuscript was proposed to eSTREAM Stream Cipher Proposal `http://www.ecrypt.eu.org/stream/`. The reference codes are available there. The first author was supported in part by JSPS Grant-In-Aid #16204002, and Hiroshima University President's Discretion Fund '05.

[1]It seems there is no standard terminology for the source generator in a filtered generator: in many articles it is referred merely as the LFSR. We shall refer the source generator as the "mother generator" in this article.

Here x_i $(i = 0, 1, 2, \ldots)$ are 32-bit integers, each of which is considered as a 32-dimensional row vector over the two element field $\mathbb{F}_2$. The binary operator $\oplus$ denotes the bitwise exclusive-or, i.e., addition as a vector. The C-like hexadecimal notation 0x80000000 denotes the vector whose components are all zero except for the left most 1, and & denotes the bitwise AND operator. Thus,

$$((w_i\&\texttt{0x80000000})|(w_{1+i}\&\texttt{0x7fffffff}))$$

is the row vector obtained by concatenating the MSB of w_i and all bits but the MSB of w_{1+i}. To this vector a constant 32×32 matrix A is multiplied from the right, which is defined and computed by

$$xA = \begin{cases} \text{shiftright}(x) & \text{(if the LSB of } x \text{ is 0)} \\ \text{shiftright}(x) \oplus a & \text{(if the LSB of } x \text{ is 1),} \end{cases}$$

where a is a constant vector $a = (a_{31}, a_{30}, \ldots, a_0) = \texttt{0x9908B0DF}$. Let us fix a j, $1 \le j \le 32$. If we look at the j-th bit of x_i for $i = 0, 1, 2, \ldots$, they constitute a linear recurring sequence over $\mathbb{F}_2$ with order 19937 with 135 terms. Its period is $P := 2^{19937} - 1$, and 623-dimensional tuples $(x_i, x_{i+1}, \ldots, x_{i+622})$ assume every possible (there are $2^{623 \cdot 32}$) bit pattern twice, except for the all 0 pattern which occurs once, in a whole period $0 \le i \le P - 1$.

We then consider x_i as 32-bit integers modulo 2^{32}, i.e. as elements of $\mathbb{Z}/2^{32}$, and pass them to the following simple filter with memory. We set y_1 to an odd integer (chosen to be 1 in CryptMT), and generate a sequence of 32-bit integers y_i by

$$y_{i+1} := (x_i|1) \times y_i \mod 2^{32},$$

where $(x_i|1)$ denotes x_i with its LSB set to 1. We use the most significant 8 bits of y_i as the output. In the implementation, we prepare a variable `accum` of 32-bit integer, and substitute it iteratively by

$$\texttt{accum} = \texttt{(output\_of\_MT() | 1)} \times \texttt{accum} \mod 2^{32}.$$

We call such type of filter with memory, based on the multiplication and the use of MSBs, a *multiplicative filter*.

The resynchronization (initialization) scheme will not be discussed in this article; it is described in [16] (and we propose a new faster version [17]).

To explain the design rationale, we compare CryptMT with SNOW2.0 (or its original version SNOW1.0) [7][8], which is also a linear generator with a filter with memory. The mother generator of SNOW is a LFSR of order 16 over $\mathbb{F}_{2^{32}}$, and its filter has two memories of 32-bit word size. The transition function of the filter is a combination of an integer addition and an exclusive-or, with nonlinearity introduced by 4 copies of an 8-bit S-box (based on the 7th-power operation in $\mathbb{F}_{2^8}$ in SNOW1.0, and on the inverse operation in $\mathbb{F}_{2^8}$ in SNOW2.0).

Design of CryptMT comes from the two observations: (1) we may use a huge state in a software, (2) we may use integer multiplication instead of S-box. We shall discuss on these two.

2.1. **Use a linear generator with huge (19937-bit) internal state space.** Many attacks depend on the size of the internal state, and become infeasible when the size is large. Typical filter generators have 128–512 bits of internal state. However, we may use more memory in a software implementation. In addition, in many platforms, the generation speed is even faster, when the internal state is larger (if the number of operations to generate one word is independent of the size, such as

in the case of MT), due to the cache memory and pipeline-processing. Thus, we propose to use a large-state generator such as MT.

There is a trade-off between the memory size and security. Our claim is that, there should be some needs for a cipher with astronomical resistance, at the cost of 625 words (2.5KB) of memory. We may also argue that a fast software implementation of AES consumes roughly four times memory than MT [1], due to a large look-up table. The memory size of CryptMT seems not a big issue in a software.

2.2. **Use of filter with memory, based on multiplication.** A most conventional design is a linear generator with memoryless filter. However, the (fast) algebraic attacks are always threats to such generators, see for example an attack[6] to Sfinks[2] (this attack seems not practical, but shows some potential weakness). According to a claim in [6], such attacks show the necessity of big margins for the security in such stream ciphers.

In a recent study [5], N. Courtois shows that some fast algebraic attack is also applicable for filter with memory, but if the memory size is large (say, more than 4 bits) then it becomes infeasible. Thanks to the filter with 64-bit memory, SNOW2.0 seems safe at present.

A difference on the filter between CryptMT and SNOW2.0 is that CryptMT utilizes multiplication in $\mathbb{Z}/2^{32}$ to introduce non-linearity, whereas SNOW utilizes four copies of one same S-box of 8-bit size, based on arithmetic operations in $\mathbb{F}_{2^8}$. In a fast implementation, SNOW uses a large size of look-up table (depending on the implementations: 2^8 words to 2^{16} words). However, recent studies [19][3] warn about the possibility of cache-timing attacks for ciphers using a large look-up table. CryptMT is safe with respect to this attack.

Moreover, recent trend shows that modern CPUs tend to have a faster multiplication instruction, so the cost of the multiplication would probably become even smaller in near future.

One may feel that the integer multiplication is simpler and hence vulnerable, compared to S-boxes based on operations in the finite field. We feel converse: since the mother generator is based on the finite fields over $\mathbb{F}_2$, operations not from such finite fields would be preferable in the filter. A toy model of CryptMT shows high algebraic degrees and nonlinearity for the multiplicative filter, which supports its effectiveness. See §4.7 and §4.8.

3. Advantages of CryptMT

An advantage of CryptMT over other ciphers is that the key size and IV size are variable and can be specified by the users, both up to 2048 bits (up to 64 words of 32-bit integers), thanks to the 19937+32-1 bits of internal state (the memory of the filter being odd, hence $32 - 1$).

Because of the progress of attacks (such as the new kind[12] of time-memory-tradeoff attacks, which claims that every stream cipher has security level less than its key length), it may perhaps become necessary to consider a larger key than 256 bits, in future. Even if that occurs, CryptMT can be used with no change.

Another advantage is that its period is $2^{19937}-1$ (see Theorem A.1 for $\geq 2^{19937}-1$, and the appendix of [16] for the equality). This is in contrast to most generators with non-linear recursion, which have the danger of short period cycles.

4. Resistance to standard attacks

We shall use the letter ℓ to denote the size of the internal state of the mother generator ($\ell = 19937$ for MT case), and w to denote the size of the memory in the multiplicative filter ($w = 32$ for CryptMT).

4.1. **Time-Memory-Tradeoff attacks.** A naive time-memory-tradeoff attack consumes the computation time of roughly the square root of the size of the state space, which is $O(\sqrt{2^{\ell+w-1}}) = O(2^{9984})$ for CryptMT.

The new class of time-memory-tradeoff attacks introduced in [12] is independent of the state size, and depending only on the key size. It is applicable to any stream ciphers. We will not discuss on the resistance of CryptMT against this attack here. Still, we note that in CryptMT both the key size and the IV size are up to 2048 bits, which will allow the users to choose a security level against such attacks.

4.2. **An abstract description of CryptMT.** CryptMT can be considered as an automaton with no inputs, with the state space ${\mathbb{F}_2}^{\ell} \times (\mathbb{Z}/2^w)^{\times}$, where $^{\times}$ denotes the set of invertible elements. In the following analysis, it is convenient to fix a model for generators using a filter with memory.

Definition 4.1. (Mother generator + filter with memory.) Let S be the state space of the mother generator, $h : S \to S$ its state transition function, and $o : S \to X$ its output function (X: output symbols). Let Y be the state space of the filtering automaton, and

$$f : X \times Y \to Y$$

be the state transition function, where X is now considered as the set of input symbols. The output function of the filtering automaton is $g : Y \to B$, where B is the output symbols of the filtering automaton.

The *composed generator* C is an automaton, with the state space $S \times Y$, the transition function

$$(s, y) \mapsto (h(s), f(o(s), y)),$$

and the output function

$$(s, y) \mapsto g(y) \in B.$$

4.3. **A cheating argument on a modified generator.** Before going into an analysis on correlation attacks, we would like to prepare a cheating argument.

Fix an initial state s_0 of the mother generator from now on. Consider an initial state (s_0, y_0) of the composed generator. We *assume* that the transition function h of the mother generator is bijective. Let P be its period (for the initial state s_0). After P times transitions, the state of the composed generator will be (s_0, y_0') for a unique $y_0' \in Y$ determined by y_0, which gives a mapping (for fixed s_0)

$$\phi : Y \to Y, \quad y_0 \mapsto y_0'.$$

We *assume* that the transition function of the filter $f(x, y)$ is a bijection for any fixed x, that is, for any $x \in X$,

$$f(x, -) : Y \to Y, \quad y \mapsto f(x, y)$$

is bijective. These assumptions assure that the transition function of the composed generator is bijective. Hence, $\phi : y_0 \mapsto y_0'$ is bijective and Y is partitioned into some orbits of ϕ. Suppose that there are k orbits. We choose representatives

$y_0, y_1, \ldots, y_{k-1}$ from each orbit, and construct a new automaton C'. The state space and the output function are the same with C, and the state transition is

$$(s, y) \mapsto \begin{cases} (h(s), f(o(s), y)) & \text{if } (h(s), f(o(s), y)) \neq (s_0, y_j) \text{ for any } j \\ (s_0, y_{(j+1 \bmod k)}) & \text{if } h(s) = s_0 \text{ and } f(o(s), y) = y_j. \end{cases}$$

This transition is chosen to have a maximally long orbit, as follows. The outputs of C and C' are identical before C returns to the initial state. Immediately before C returns to the initial state, C' changes its state to the next orbit specified by the representative y_1, and works in the same way with C, until the state returns to (s_0, y_1). Just one step before to reach to (s_0, y_1), C' changes the state to (s_0, y_2). This assures the following.

Proposition 4.2. For any $s \in S$ in the orbit of the mother generator started from s_0, and for any $y \in Y$, the state (s, y) occurs exactly once in the orbit of C' starting from (s_0, y_0). The period of the state transition is $P \times \#(Y)$.

Proof. By the construction of C' by patching the orbits, the period is $P \times \#(Y)$. Since this coincides with the number of possible (s, y), each of these must appear in the orbit exactly once. □

Our cheating argument is

Assumption 4.3. If the period of the mother generator P is large enough, then in practice our consumptions of the outputs of C can not reach to P. Hence, we do not need to distinguish C and C'. We assume that the statistical analysis on C' for full period will give a good approximation to that on C.

This last assumption may seem to be cheating, but this level of "dishonesty" is hidden in many arguments, such as the statistical analysis on LFSRs [9], where the distribution property and the correlation are computed under the assumption that the full-period is used, but in reality a small fraction of the period is used. In this regard, the identification of C and C' seems just as sinful as such standard arguments. We use C' in the following statistical analysis, instead of C. Another way to justify such an assumption is to choose y_0 randomly at each synchronization.

4.4. **n-dimensional distribution.** A sequence of X with period P is said to be n-dimensionally equidistributed with defect d and multiplicity M, if its outputs $x_0, x_1, \ldots$ satisfy the following. Let

$$O_n := \{(x_i, x_{i+1}, \ldots, x_{i+n-1}) \mid 0 \leq i \leq P - 1\}$$

be the multi-set of the n-tuples for one period, counted with multiplicity. Then,

$$\#((MX^n) \setminus O_n) = d$$

holds, where MX^n denotes the multiset which contains every element of X^n with multiplicity M, $\setminus$ denotes the difference, and the cardinality is computed with counting the multiplicity.

MT as a 32-bit integer generator has this property with $n = 623$, $M = 2$, and $d = 1$, see [14]. The difference comes from the zero state.

Proposition 4.4. We keep the set-up of Definition 4.1. Assume that h is bijective, that f is bijective at both variables, namely, $f(-, y) : X \to Y,\ x \mapsto f(x, y)$ is bijective for any fixed y, and so is $f(x, -) : X \to Y,\ y \mapsto f(x, y)$ for any fixed x. Assume that the output function $g : Y \to B$ is uniformly N to 1 (i.e. $\#(g^{-1}(b)) =$

N for any $b \in B$). Take an initial state (s_0, y_0). Suppose that the mother generator is n-dimensionally equidistributed with multiplicity M with defect d. Then, the modified generator C' is $(n+1)$-dimensionally equidistributed with defect $d\#(Y)$.

Proof. We may replace S with the orbit starting from s_0. Then, replace S with its quotient set where two states are identified if the output sequences from them are identical. Thus, we may assume $\#(S) = P$.

Consider the n-tuple output function of the mother generator $o_n : S \to X^n$, which maps a state s to the consecutive n outputs from the state s. Then, the equidistribution property is equivalent to

$$o_n(S) = MX^n \setminus D,$$

where $D \subset X^n$ is a multiset of cardinality d corresponding to the defect. The $(n+1)$-tuple output function $O_{C'}$ of the modified generator C' is the composite

$$O_{C'} : S \times Y \overset{o_n \times \mathrm{id}_Y}{\to} X^n \times Y \overset{\mu}{\to} Y^{n+1} \overset{g^{n+1}}{\to} B^{n+1},$$

where the second map μ is given by

$$\mu : ((x_n, x_{n-1}, \dots, x_1), y_1) \mapsto (y_{n+1}, y_n, \dots, y_1)$$

where y_i's are inductively defined by $y_{i+1} := f(x_i, y_i) \quad (i = 1, 2, \dots, n)$. The assumption on f implies the bijectivity of μ. The third map is uniformly N^{n+1} to 1. By taking the image of $S \times Y$, we have

$$O_{C'}(S \times Y) = N^{n+1} M B^{n+1} \setminus g^{n+1} \circ \mu(D \times Y),$$

which shows $(n+1)$-dimensional equidistribution of the output of C' with defect $\#(g^{n+1} \circ \mu(D \times Y)) = d\#(Y)$. □

Corollary 4.5. The modified CryptMT in the sense of §4.3 is 624-dimensionally equidistributed with defect 2^{31}.

Proof. MT is 623-dimensionally equidistributed with defect 1 [14]. This is true even when the LSB of the output is set to 1. Now X is the set of 32-bit *odd* integers, and $Y = X$. Then the multiplication $X \times Y \to Y$ is bijective at the both variables. Thus the assumptions in Proposition 4.4 are satisfied. The output function $g : Y \to {\mathbb{F}_2}^8$ taking 8 MSBs is uniform. □

4.5. **Correlation attacks and distinguish attacks.**

Proposition 4.6. Let F be any real-valued function whose inputs are (less than or equal to) $(n+1)$ elements of B. Let $E_{C'}(F)$ be the average value of F applied to the consecutive $(n+1)$ outputs of the modified generator C' stated in Proposition 4.4 for a full period, where all conditions of the proposition are assumed. Then the error term is bounded by

$$|E_{C'}(F) - E(F)| \le 2d||F||/(P+d),$$

where $E(F)$ is the expectation of F when the $(n+1)$ variables are independently and uniformly randomly chosen from B, and $||F||$ is the maximum of the absolute value of F.

Proof. By Proposition 4.4, C' is $(n+1)$-dimensionally equidistributed with some multiplicity N' and defect $d\#(Y)$, that is

$$O' := O_{C'}(S \times Y) = (N' B^{n+1}) \setminus T$$

for $\#(T) = d\#(Y)$, and hence $\#(N'B^{n+1}) = \#(S \times Y) + \#(T) = (P+d)\#(Y)$. By definition

$$E_{C'}(F) = \sum_{\mathbf{b} \in O'} F(\mathbf{b})/\#(O'), \quad E(F) = \sum_{\mathbf{b} \in N'B^{n+1}} F(\mathbf{b})/\#(N'B^{n+1}).$$

Then we have

$$\begin{aligned}
&|E_{C'}(F) - E(F)| \\
&= \frac{|\#(N'B^{n+1}) \sum_{\mathbf{b} \in O'} F(\mathbf{b}) - \#(O') \sum_{\mathbf{b} \in N'B^{n+1}} F(\mathbf{b})|}{\#(O')\#(N'B^{n+1})} \\
&\leq \frac{|\#(T) \sum_{\mathbf{b} \in O'} F(\mathbf{b}) - \#(O') \sum_{\mathbf{b} \in T} F(\mathbf{b})|}{\#(O')\#(N'B^{n+1})} \\
&\leq \frac{\#(T)}{\#(N'B^{n+1})} \left(\frac{|\sum_{\mathbf{b} \in O'} F(\mathbf{b})|}{\#(O')} + \frac{|\sum_{\mathbf{b} \in T} F(\mathbf{b})|}{\#(T)} \right) \leq 2d||F||/(P+d).
\end{aligned}$$

□

Corollary 4.7. We mean by a *simple distinguishing attack of order N* to choose a function F (with up to N variables) and to detect the deviation of the values of F applied to the consecutive N-outputs. Then, its deviation is bounded by $2d||F||/(P+d)$, and hence we need $O((P/d)^2)$ samples to detect it statistically.

Corollary 4.8. The security level of CryptMT to such attacks for $N \leq 624$ is $2^{19937 \times 2}$.

By this reason, it seems very difficult to apply a correlation attack to CryptMT. One needs to observe the correlation of outputs with the lag more than 624. Because of the high nonlinearity of the multiplicative filter discussed below, we guess this is infeasible.

One might think that MT would be weak since its recurrence is sparse and we can easily find many three-term relations between bits among the consecutive 624 words of MT. However, the digits of the dependent bits differ [14]. Suppose that the output word-sequence (x_j) satisfy a linear relation of type

$$x_{N+j} = \sum_{i=0}^{N-1} a_i x_{i+j}, \quad (a_i \in \mathbb{F}_2)$$

where each word is considered as a vector in ${\mathbb{F}_2}^{32}$. If the number of nonzero coefficients (including that of x_{N+j}) is t, then we call the above relation as an *N-th order t-term linear word-relation*. The smallest order linear word-relation is of order 19937 with 135 terms for MT (see [14]).

This invalidates improved correlation attacks such as the attack [18] to LILI-128 or the attack to SNOW1.0 [11][8], both of which depend on a few-term linear word-relation of the mother generator. In the former case, the attackers found a four-term linear word-relation $x_i + x_{i+j_1} + x_{i+j_2} + x_{i+j_3} = 0$. In LILI-128, a filtering Boolean function F *without* memory is used. An analysis in [18] showed that

$$Prob(F(x_i) + F(x_{i+j_1}) + F(x_{i+j_2}) + F(x_{i+j_3}) = 0) \geq \frac{1}{2} + \frac{1}{2(2^w - 1)},$$

where w is the number of tapping positions from the mother generator to the filter function. They gave a more exact value using the Walsh spectrum of F, and since it is significantly greater than $1/2$ a distinguishing attack is possible.

This attack is not feasible to CryptMT, at least as is, since the filter of CryptMT has memory. Even if we consider a memoryless filter + MT, this attack is infeasible, because even the fast algorithm [20] to find a four-term relation requires the runtime complexity $O(N \log N)$ and memory complexity $O(N)$, where $N = 2^{\ell/3} = 2^{6645.7}$ for MT.

4.6. **Advantage of a Mersenne exponent extension, over LFSRs with coefficients in $\mathbb{F}_{2^{32}}$.** One weakness of SNOW1.0 utilized in the guess and determine attack [11][8] is that its mother generator is a LFSR on $\mathbb{F}_{2^{32}}$, with the recursion polynomial being for an $\alpha \in \mathbb{F}_{2^{32}}$

$$p(x) = x^{16} + x^{13} + x^7 + \alpha^{-1} \in \mathbb{F}_{2^{32}}[x].$$

Since the 2^{32}-th power operation is the identity on $\mathbb{F}_{2^{32}}$, we have a multiple of $p(x)$

$$p(x)^{2^{32}} = x^{16 \cdot 2^{32}} + x^{13 \cdot 2^{32}} + x^{7 \cdot 2^{32}} + \alpha^{-1} \in \mathbb{F}_{2^{32}}[x],$$

and by eliminating α^{-1} from these two equations, we obtain a linear relation between 6 words, with coefficients equal to 1.

In SNOW2.0, several improvements are introduced. One of them is to replace $p(x)$ with

$$\pi(x) = \alpha x^{16} + x^{14} + \alpha' x^5 + 1$$

for an element $\alpha' \in \mathbb{F}_{2^{32}}$ (actually it is α^{-1}). This would be practically enough, but we can eliminate α, α' from three equations

$$\pi(x) = 0, \pi(x)^{2^{32}} = 0, \pi(x)^{2^{64}} = 0.$$

The result is

$$\det \begin{pmatrix} x^{16} & x^5 & x^{14}+1 \\ x^{16N} & x^{5N} & x^{14N}+1 \\ x^{16N^2} & x^{5N^2} & x^{14N^2}+1 \end{pmatrix} = 0,$$

having 24 terms (here $N = 2^{32}$). This would be useless to design an attack, but still gives a slight negative flavor to the recursion.

Perhaps, to choose a linear recursion over a non-prime field (such as $\mathbb{F}_{2^{32}}$) may be not a best idea. In the case of Mersenne Twister, the characteristic polynomial of the state transition has degree 19937, which is a prime. Hence, no intermediate field exists, and it seems impossible to apply the above trick.

Moreover, since $2^{19937} - 1$ is a prime number, it seems difficult to obtain any information from decimation techniques.

4.7. **A proposition on the algebraic degree of integer products.** To discuss about algebraic attacks, we prepare a lemma on the algebraic degree. Let $f(c_1, c_2, \ldots, c_n)$ be a boolean function, i.e., c_i's are variables each of which assumes 0 or 1, and the value of f is 0 or 1. Then, f can be represented by an n-variable polynomial function with coefficients in $\mathbb{F}_2$, namely as a function

$$f = \sum_{T \subset \{1,2,\ldots,n\}} a_T c_T$$

holds, where $a_T \in \mathbb{F}_2$ and $c_T = \prod_{t \in T} c_t$. This representation is unique, and called the *algebraic normal form*. Its degree is called the algebraic degree of f.

The following lemma is well-known.

Lemma 4.9. It holds that $a_T = \sum_{U \subset T} f(U)$, where $f(U) := f(c_1, \ldots, c_n)$ with $c_i = 0, 1$ according to $i \notin U$, $\in U$, respectively.

Definition 4.10. Let us define a boolean function $m_{s,N}$ of $(s-1)N$ variables, as follows. Consider N of s-bit integer variables $x_1, \ldots, x_N$. Let

$$c_{s-1,i} c_{s-2,i} \cdots c_{0,i}$$

be the 2-adic representation of x_i, hence $c_{j,i} = 0, 1$. We fix $c_{0,i} = 1$ for all $i = 1, \ldots, N$, i.e. assuming x_i odd. The boolean function $m_{s,N}$ has variables $c_{j,i}$ ($j = 1, 2, \ldots, s-1, i = 1, 2, \ldots, N$), and whose value is the s-th digit (from the LSB) of the 2-adic expansion of the product $x_1 x_2 \cdots x_N$ as an integer.

Proposition 4.11. Assume that $N, s \geq 2$. The algebraic degree of $m_{s,N}$ is bounded from below by

$$\min\{2^{s-2}, 2^{\lfloor \log_2 N \rfloor}\}.$$

Proof. For $s = 2$, the claim is easy to check. We assume $s \geq 3$.

Case 1. $s - 2 \leq \log_2 N$. In this case, it suffices to prove that the algebraic degree is at least 2^{s-2}. Take a subset T of size 2^{s-2} from $\{1, 2, \ldots, N\}$, say $T = \{1, 2, \ldots, 2^{s-2}\}$. Then, we choose $c_{1,1}, c_{1,2}, \ldots, c_{1,2^{s-2}}$ as the $\#T$ variables "activated" in Lemma 4.9, and consequently, the coefficient of $c_{1,1} c_{1,2} \cdots c_{1,2^{s-2}}$ in the algebraic normal form of $m_{s,N}$ is given by the sum in $\mathbb{F}_2$:

$$a_T := \sum_{U \subset T} (s\text{-th bit of } x_1 \cdots x_n, \text{ where } c_{j,i} = 1 \text{ if and only if } j = 1 \text{ and } i \in U).$$

Note that $c_{0,i} = 1$. It suffices to prove $a_T = 1$. Now, each term in the right summation is the s-th bit of the integer $3^{\#U}$, so the right hand side equals to

$$\sum_{m=0}^{2^{s-2}} \left[\binom{2^{s-2}}{m} \times \text{ the } s\text{-th bit of } 3^m \right].$$

However, the well-known formula

$$(x+y)^{2^{s-2}} \equiv x^{2^{s-2}} + y^{2^{s-2}} \mod 2$$

implies that the binary coefficients are even except for the both end, so the summation is equal to the s-th bit of $3^{2^{s-2}}$.

A well-known lemma says that if $x \equiv 1 \mod 2^i$ and $x \not\equiv 1 \mod 2^{i+1}$ for $i \geq 2$, then $x^2 \equiv 1 \mod 2^{i+1}$ and $x^2 \not\equiv 1 \mod 2^{i+2}$. By applying this lemma inductively, we know that

$$3^{2^{s-2}} = (1+8)^{2^{s-3}} \equiv 1 \mod 2^s, \quad \not\equiv 1 \mod 2^{s+1}.$$

This means that s-th bit of $3^{2^{s-2}}$ is 1, and the proposition is proved.

Case 2. $s - 2 > \lfloor \log_2(N) \rfloor$. In this case, we put $t := \lfloor \log_2(N) \rfloor + 2$, and hence $s > t$ and $2^{t-2} \leq N$. We apply the above arguments for $T = \{1, 2, \ldots, 2^{t-2}\}$, but this time instead of $c_{1,i}$, we activate

$$\{c_{s-t+2,i} \mid i \in T\}.$$

The same argument as above reduces the non-vanishing of the coefficient of the term $c_{s-t+2,1} \cdots c_{s-t+2,2^{t-2}}$ to the non-vanishing of

$$\sum_{m=0}^{2^{t-2}} \left[\binom{2^{t-2}}{m} \times \text{the } s\text{-th bit of } (1+2^{s-t+2})^m \right].$$

Again, only the both ends $m = 0$ and $m = 2^{t-2}$ can survive, and the above summation is the s-th bit of $(1+2^{s-t+2})^{t-2}$. Since $s-t+2 \geq 2$, the lemma mentioned above implies that

$$(1+2^{s-t+2})^{2^{t-2}} \equiv 1 \bmod 2^s, \quad \not\equiv 1 \bmod 2^{s+1},$$

which implies that its s-th bit is 1. □

4.8. **Simulation by toy models.** Since the filter has a memory, it is not clear how to define the algebraic degree or non-linearity of the filter. Instead, if we consider all bits in the initial state as variables, then each bit of the outputs is a boolean function of these variables, and algebraic degree and non-linearity are defined.

However, it seems difficult to compute them explicitly for CryptMT, because of the size. So we made a toy model and obtained experimental results. Its mother generator is a linear generator with 16-bit internal state, and generates a 16-bit integer sequence defined by

$$\mathbf{x}_{j+1} := (\mathbf{x}_j >> 1) \oplus ((\mathbf{x}_j \& 1) \cdot \mathbf{a}),$$

where $>> 1$ denotes the one-bit shift-right, $(\mathbf{x}_j \& 1)$ denotes the LSB of $\mathbf{x}_j$, $\mathbf{a} = 1010001001111000$ is a constant 16-bit integer, and $(\mathbf{x}_j \& 1) \cdot \mathbf{a}$ denotes the product of the scaler $(\mathbf{x}_j \& 1) \in \mathbb{F}_2$ and the vector $\mathbf{a}$.

Then it is filtered by

$$y_{j+1} = (\mathbf{x}_j | 1) \times y_j \mod 2^{16},$$

where $(\mathbf{x}_j|1)$ denotes $\mathbf{x}_j$ with LSB set to 1, as defined previously. We put $y_0 = 1$, and compute the algebraic degree of each of the 16 bits in the outputs $y_1 \sim y_{16}$, each regarded as a polynomial function with 16 variables being the bits in $\mathbf{x}_0$. The result is listed in Table 1. The lower six bits of the table clearly show the pattern $0, 1, 1, 2, 4, 8$, which suggests that the lower bound 2^{s-2} for $s \geq 2$ given in Proposition 4.11 would be tight, when the iterations are many enough. On the other hand, eighth bit and higher are "saturated" to the upper bound 16, after 12 generations.

We expect that the same will occur for the CryptMT case. So, if we consider each bit of the internal state of MT as a variable, then the algebraic degree of the 8 MSBs of y_i will be near to $\ell = 19937$, after some steps of generations.

Also, we computed the non-linearity of the MSB of each y_i $(i = 1, 2, \ldots, 8)$ of this toy model. The result is listed in Table 2, and each value is near to 2^{16-1}. This suggests that there would be no good linear approximation of CryptMT.

4.9. **Algebraic attacks.** Assume that the filter by multiplication is used for free variable inputs. Then, as proved in Proposition 4.11, the algebraic degree of the s-th bit increases at least up to 2^{s-2} in the long run. In the case of CryptMT, the 32nd to 24th bits are used, and their degrees would be 2^{30} to 2^{22}, respectively. This is huge when compared to the ordinary memoryless filters with limited number of input-bits, say, 16.

TABLE 1. Table of the algebraic degrees of output bits of a toy model.

y_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
y_2	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	0
y_3	15	15	14	13	12	11	10	9	8	6	4	3	2	1	1	0
y_4	15	16	15	14	13	12	11	10	9	7	5	4	2	1	1	0
y_5	16	16	15	15	14	13	12	11	10	7	5	4	2	1	1	0
y_6	16	16	15	15	15	14	13	11	10	9	7	4	2	1	1	0
y_7	16	15	16	16	15	15	14	13	12	9	7	4	2	1	1	0
y_8	15	15	15	16	16	15	15	14	13	10	8	4	2	1	1	0
y_9	16	15	16	15	15	16	15	15	13	10	8	4	2	1	1	0
y_{10}	15	16	16	16	16	16	15	15	14	12	8	4	2	1	1	0
y_{11}	15	16	16	15	15	15	16	15	15	12	8	4	2	1	1	0
y_{12}	15	16	16	16	16	15	16	16	15	13	8	4	2	1	1	0
y_{13}	16	15	15	15	15	15	16	15	16	13	8	4	2	1	1	0
y_{14}	15	15	16	15	15	16	16	15	16	15	8	4	2	1	1	0
y_{15}	15	16	16	16	15	16	16	16	15	14	8	4	2	1	1	0
y_{16}	16	15	16	15	15	15	15	15	16	14	8	4	2	1	1	0

TABLE 2. The non-linearity of the MSB of each output of a toy model.

output	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
nonlinearity	0	32112	32204	32238	32201	32211	32208	32170	32235

By these arguments and from the above experiments with the toy-model, we expect that the algebraic degree of the outputs of CryptMT with respect to the bits of the initial state would be close to the upper bound $\ell = 19937$ after sufficiently many steps.

This is in contrast to a filter without memory, where the algebraic degree of each output bit is bounded by the algebraic degree of the filter function since all output bits of a linear mother generator have algebraic degree one. For example, Sfinks stream cipher [2] has a memoryless filter of algebraic degree 15, but [6] utilized a degree-reduction technique which reduces the algebraic degree to 7. Such reduction seems very difficult for a filter with 31-bit memory.

4.10. **Berlekamp-Massey attacks.** The linear complexity (LC) of an $\mathbb{F}_2$-linear generator with ℓ-bits of the internal state with memoryless filter with algebraic degree d is expected to be approximately $\binom{\ell}{d}$, and the Berlekamp-Massey attack requires $2 \cdot LC$ data and $(LC)^2$ computational complexity. CryptMT has a filter with memory, so such estimation can not be applied. A heuristic guess is that d would be rather high if it is appropriately extended to the case of filter with memory. The size $\ell = 19937$ seems to make these attacks infeasible, too.

5. CONCLUSION

CryptMT has a huge period of $2^{19937} - 1$. Because of the size 19937+31 of the internal state and the multiplicative filter with 31-bit memory and 8-bit output, CryptMT puts two large margins for the security on both the mother generator and the filter.

By a tricky argument, we showed that the generated key stream can be regarded to have negligible (in the order of 2^{-19937}) correlation between the consecutive 624 outputs, so standard correlation attacks are very hard to apply.

We proved a proposition giving a lower bound of the algebraic degree of the multiplicative filter. The result, together with the experiments through a toy model, shows the tendency that the algebraic degree of the outputs with respect to the initial state of the mother generator increases after each step, until they become saturated near the upper bound 19937. The toy-model also suggests that the nonlinearity with respect to the key and the initial value would be close to the upper bound.

CryptMT admits variable key-size and IV-size, upto 2048 bits for each. We claim that its security level is at least the minimum of the key size and the IV size.

Differently from the fast implementations of AES, CryptMT uses no look-up tables, so it has resistance against cache-timing attacks. It is 1.5–2.0 times faster than AES CTR mode with 256-bit security level (depending on the platform, if the CPU is slow at multiplication, then it is slower than AES).

6. Tweaks

6.1. **Resynchronization scheme.** The present resynchronization scheme in [16] is redundant and slow, since it was designed for a large scale Monte Carlo simulation where the initialization speed is not so important. We propose a much faster resynchronization scheme [17].

6.2. **MT replaced with other generators.** We reported a new version of MT [10], pulmonary MT, with better bit-mixing property. We propose to replace MT with this [17].

6.3. **Change of the filter.** The simple choice $f(x, y) = x \times y \mod 2^{32}$ and outputting the most significant 8 bits would have enough resistance against attacks, but still the adversary can get some information. For example, if the 8 MSBs of y_i and y_{i+1} do not coincide, then we know that $x_i \neq 1$. Similarly, we can know that $x_i \neq 3, 5, \dots 255$ nor their multiplicative inverses in $\mathbb{Z}/2^{32}$, for some pairs of the 8 MSBs. Since the multiplication is associative, we can get similar information on $x_i x_{i+1} \cdots x_{i+j-1}$ from the 8 MSBs of y_i and those of y_{i+j}.

We may change f to address the above point. Theorem A.1 assure that the period is no less than $2^{19937} - 1$, as far as f is bijective at the both variables.

Appendix A. A theorem on the period

Theorem A.1. Consider a combined generator C as in Definition 4.1. Assume that the mother generator is purely periodic for an initial state s_0 with period $P = Qq$ for a prime Q and an integer q, S is an orbit (by replacing S if necessary), and that $o_n : S \to X^n$ mapping the state to the next n outputs of the mother generator is surjective. Suppose that f is bijective at both variables as in Proposition 4.4. Let $y_0, y_1, \dots \in Y$ be the state transition of the filter of C. Let r be the ratio of the size of the maximum inverse image of $g : Y \to B$ in Y, namely

$$r = \max_{b \in B} \{\#(g^{-1}(b))\}/\#(Y).$$

If

$$r^{-(n+1)} > q(\#(Y))^2,$$

then the period of the output sequence $g(y_0), g(y_1), \ldots$ of C is a nonzero multiple of Q.

Proof. We may assume that $\#(S) = P$ as in the proof of Proposition 4.4.

In this proof, we do not consider multi-sets. Consider the mappings

$$O_C : S \times Y \overset{o_n \times id_Y}{\to} X^n \times Y \overset{\mu}{\to} Y^{n+1} \overset{g^{n+1}}{\to} B^{n+1}$$

defined in the proof of Proposition 4.4. (The difference between C and C' does not matter in this proof.) Since o_n is surjective and μ is bijective, the image $I \subset Y^{n+1}$ of $S \times \{y_0\}$ by $\mu \circ (o_n \times id_Y)$ has the cardinality $\#(X)^n$. By the assumption of the pure periodicity of x_i and the bijectivity of f, the output sequence $g(y_i)$ $(i = 0, 1, 2, \ldots)$ is purely periodic. Let p be the period. Then, $g^{n+1}(I) \subset B^{n+1}$ can have at most p elements. Thus, by the assumption on g and the definition of r,

$$\#(I) \le p(r\#(Y))^{n+1}.$$

Since $\#(X)^n = \#(I)$ and $\#(X) = \#(Y)$, we have an inequality

$$r^{-(n+1)} \le p\#(Y).$$

The period P' of the state transition of C is a multiple of $P = Qq$. Since the state size of C is $P \times \#(Y)$, $P' = Qm$ holds for some $m \le q\#(Y)$. Consequently, p is a divisor of Qm. If p is not a multiple of Q, then p divides m and $p \le q\#(Y)$. Thus we have

$$r^{-(n+1)} \le q\#(Y)^2,$$

contradicting to the assumption. □

Corollary A.2. Each bit of the output of CryptMT has period at least $2^{19937} - 1$. This is true even if we replace f with any function which is bijective at both variables.

Proof. Let S be the set of the nonzero states. Let $g : Y \to B = \mathbb{F}_2$ be the observed bit of the state y of the filter. Then $r = 1/2$, and

$$2^{(623+1)} > 1 \cdot (\#(Y))^2 = 2^{62}.$$

□

References

[1] AES lounge: http://www.iaik.tu-graz.ac.at/research/krypto/AES/

[2] Braeken, A., Lano, J., Mentens, N., Preneel, B., and Verbauwhede, I. SFINKS: A Synchronous Stream Cipher for Restricted Hardware Environments. Submitted to eSTREAM stream cipher proposals, `http://www.ecrypt.eu.org/stream/`.

[3] Bernstein, D. J. Cache-timing attack on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] Biryukov, A., Shamir, A. and Wagner, D. Real time cryptanalysis of A5/1 on a PC. In B. Schneier, editor, Fast Software Encryption, FSE 2000, LNCS 1978 1–18. Springer-Verlag, 2000.

[5] Courtois, N. Algebraic Attacks on Combiners with Memory and Several Outputs, to appear in ICISC 2004, LNCS, Springer. The extended and recently updated version of this paper is available at eprint.iacr.org/2003/125/.

[6] Courtois, N. Cryptanalysis of Sfinks, http://eprint.iacr.org/2005/243.

[7] Ekdahl, P. and Johansson, T. SNOW - a new stream cipher, `http://www.it.lth.se/cryptology/snow/snow10.pdf`

[8] Ekdahl, P. and Johansson, T. A new version of the stream cipher SNOW, `http://www.it.lth.se/cryptology/snow/snow20.pdf`

[9] Golomb, S. Shift Register Sequences. Aegean Park Press, 1982.

[10] Haramoto, H., Panneton, F., Nishimura, T., and Matsumoto, M. Hearty Twister: a new random number generator, a talk in Fifth IMACS seminar on Monte-Carlo Method MCM2005, 2005 May at Florida State University.
[11] Hawkes, P. and Rose, G. Guess-and-determine attacks on SNOW, Preproceedings of Selected Areas in Cryptography (SAC), August 2002, St John's, Newfoundland, Canada.
[12] Hong, J. and Sarkar, P. Rediscovery of time memory tradeoffs. Cryptology ePrint Archive, Report 2005/090, 2005. http://eprint.iacr.org/.
[13] Knuth, D. E. The Art of Computer Programming. Vol. 2. Seminumerical Algorithms 3rd Ed. Addison-Wesley, Reading, Mass., (1997).
[14] Matsumoto, M. and Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 8 (1998) 3–30.
[15] Matsumoto, M. and Nishimura, T. Mersenne Twister Homepage. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/emt.html
[16] Matsumoto, M., Nishimura, T., Saito, M. and Hagita, M. Cryptographic Mersenne Twister and Fubuki stream/block cipher, `http://eprint.iacr.org/2005/165`.
This is an extended version of "Mersenne Twister and Fubuki stream/block cipher" submitted for eSTREAM proposal `http://www.ecrypt.eu.org/stream/`.
[17] Matsumoto, M., Saito, M., Nishimura, T. and Hagita, M. CryptMT Version 2.0: a large state generator with faster initialization, to appear in the conference volume of SASC2006 `http://www.ecrypt.eu.org/stream/`.
[18] Molland, H. and Helleseth, T. An improved correlation attack against irregular clocked and filtered keystream generators. In Matthew Franklin, editor, Advances in Cryptology CRYPTO 2004, LNCS 3152, 373–389. Springer- Verlag, 2004.
[19] Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. Cryptanalysis of DES implemented on computers with cache, in Cryptographic hardware and embedded systems–CHES 2003, Springer-Verlag, Berlin (2003), 62–76.
[20] Wagner, D. A generalized birthday problem. In Advances in cryptology-CRYPTO 2002, LNCS 2442, 288-303, 2002.

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: `m-mat@math.sci.hiroshima-u.ac.jp`

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: `saito@math.sci.hiroshima-u.ac.jp`

Department of Mathematics, Yamagata University, Yamagata Japan
E-mail address: `nisimura@sci.kj.yamagata-u.ac.jp`

Department of Information Science, Ochanomizu University, Tokyo Japan
E-mail address: `hagita@is.ocha.ac.jp`

CRYPTMT VERSION 2.0: A LARGE STATE GENERATOR WITH FASTER INITIALIZATION

MAKOTO MATSUMOTO, MUTSUO SAITO, TAKUJI NISHIMURA,
AND MARIKO HAGITA

Abstract. As a pseudorandom number generator (PRNG) for a stream cipher, we propose a combination of (1) an $\mathbb{F}_2$-linear generator of a wordsize-integer sequence with huge state space, and (2) a filter with one wordsize memory, based on the accumulative integer multiplication and extracting some most significant bits from the memory. We proposed CryptMT as an example. Merits of this type of generators are (1) the strength against various attacks assured by the huge state, (2) assurance on the period and the distribution, and (3) high algebraic degree and nonlinearity obtained by the integer multiplication.

One problem of such configuration is the cost at the initialization required to set the huge state. In this article, we introduce a method to avoid this cost by means of a booting PRNG with small state space. We propose CryptMT Ver.2.0 with this quick initialization. In addition, an improved $\mathbb{F}_2$-linear generator, Pulmonary Mersenne Twister, is used as the mother generator. The result is: almost same speed in the stream generation, and 15 times faster in the initial value setup than the original version of CryptMT.

1. Introduction

In this article, we discuss on pseudorandom number generators (PRNGs) for stream ciphers. We denote by w the computer's word size, and assume that $w = 32$ as the default value. We consider implementations in software only. Our proposal is to combine a huge state generator M (called the mother generator) and a filter based on integer-multiplication as follows.

(1) The mother generator M should have very long period and high dimensional equidistribution property. Our proposal for M is an $\mathbb{F}_2$-linear generator with a huge (say more than 200 words of) state space. The outputs $x_0, x_1, x_2, \ldots$ of M is a w-bit integer sequence.

(2) Put these integers into a filter with one word-size memory. Let `accum` (accumulator) be a w-bit integer variable. In the initialization, we set `accum` to some initial value, as well as initializing M. Then, at the i-th step, we assign

$$\texttt{accum} := f(\texttt{accum}, x_i)$$

and output $g(x_i)$, where f is a function based on the integer multiplication (modulo 2^w), and $g(x_i)$ is to take some fixed bits of x_i.

Date: January 23, 2006.

Key words and phrases. Cryptographic Mersenne Twister, CryptMT, Pulmonary Mersenne Twister, stream cipher, booter.

CryptMT is proposed to eSTREAM Proposal `http://www.ecrypt.eu.org/stream/`. The first author was supported in part by JSPS Grant-In-Aid #16204002.

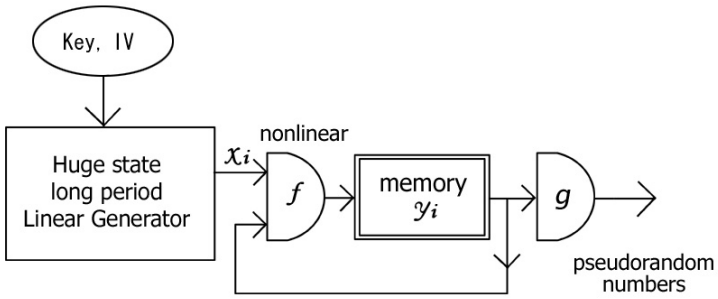


FIGURE 1. Combined generator = linear generator + filter with memory.

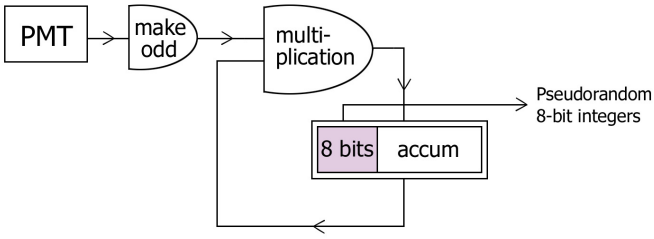


FIGURE 2. CryptMT Version 2.0: MT is replaced with Pulmonary MT. The new initialization is not described here.

Pictorial description is in Figure 1. We call this configuration *the combined generator* in this article. Note that this filter is nothing but a finite state automaton. We proposed CryptMT [4][5] as an example, where the mother generator is Mersenne Twister (MT) 32-bit integer generator [3] with 19937-bit internal state with period $2^{19937} - 1$, and the filter is given by

$$f(y, x) := y \times (x|1) \bmod 2^{32},\ g(y) := 8 \text{ MSBs of } y \tag{1}$$

where $(x|1)$ denotes x with LSB set to 1, and 8 MSBs mean the most significant 8 bits of w-bit integer y. CryptMT is proved to have period $2^{19937} - 1$ and to be very strong to standard attacks in [5]. CryptMT has also assurance of the high dimensional equidistribution property, namely, the consecutive 624 bytes are uniformly equidistributed [5, Corollary 4.5, Proposition 4.6]. These are inherited from the mother generator. Also, the high nonlinearity introduced by the integer multiplication would imply high algebraic degree and high nonlinearity (a lower bound on the algebraic degree of most significant bits of accumulated products [5, Proposition 4.11], together with experiments by toy models [5, Tables 1 and 2], supports this). The security margin obtained by discarding 3/4 of each 32-bit integer raises the hardness to break.

On the other hand, a demerit of such configurations is the high cost at the initialization, necessary to fill the huge state space of the mother generator. In this article,

we propose a cheating solution to this problem, by using another random number generator called a *booter*, which has smaller state space, until the initialization of the mother generator is done.

We also introduce a new mother generator, Pulmonary Mersenne Twister (PMT), for faster generation and improved linear dependencies from MT.

2. A fast initialization of a large state space

2.1. **A cheating method: use a smaller generator for a while.** Let X be the set of w-bit integers. Let $x_i \in X$ $(i = 0, 1, 2, \ldots)$ be a sequence generated by a recursion

$$x_{N+i} := F(x_{N-1+i}, x_{N-2+i}, \ldots, x_{1+i}, x_i),$$

for some $F : X^N \to X$. Suppose that this recursion is used as the mother generator, and hence N is large (e.g. $N = 624$ for MT). A software implementation of such a recursion is: to prepare an array of elements of X with size N, and to use pointers and a cyclic array. It is inevitable to give $x_0, x_1, \ldots, x_{N-1}$ as the initial state, in other words, to fill up the state array, before generation. Thus, we need to generate N of pseudorandom numbers in the initialization.

However, if one wants to encrypt a much shorter message than N, then this is not efficient. A possible solution is to use a PRNG with relatively small state space (called *the booter*) which can be quickly initialized, and use it to generate $x_0, x_1, \ldots, x_{N-1}$ from the key and the initial value (IV). If the message length is smaller than N, then the mother generator is never used: only the booter is used for the necessary times. This seems a little cheating. However, the difference is merely to use $x_0, x_1, \ldots$ (the output of the booter for the first N steps) or $x_N, x_{N+1}, \ldots$ (involving the mother generator). Also, the attacks to the booter is rather limited, since at most N outputs are used. A large period is not necessary. Attacks based on long outputs, such as time-memory-trade-off attacks or Berlekamp-Massey LFSR synthesis attacks, are not applicable to the booter. On the other hand, the booter must have resistance against the attacks designed for the block cipher, since the role of the booter is to "encrypt" IV into a block of N wordsize integers by using the key, without leaking any information on the key even for chosen IVs. This situation is closer to the block ciphers than stream ciphers. A typical attack is the differential attack with respect to the IV.

2.2. **The key, IV, and the Booter.** Here we consider the following situation.

(1) The algorithm is implemented in a software, where we have enough memory and fast integer multiplication.
(2) The user gives the key in the array KEYARRAY of w-bit integers with length KEYSIZE, and the IV in the array IVARRAY of w-bit integers with length IVSIZE.
(3) The key setup does not occur frequently, so the speed does not matter.
(4) The IV setup occurs frequently, so the speed does matter.
(5) Every IV is known to and can be chosen by the adversary.

The booter's role is to expand the key and IV to N wordsize integers. Since the first outputs of the booter are used as the outputs of the combined generator after filtered, the booter should have enough strength against chosen IV attacks. We choose the following strategy.

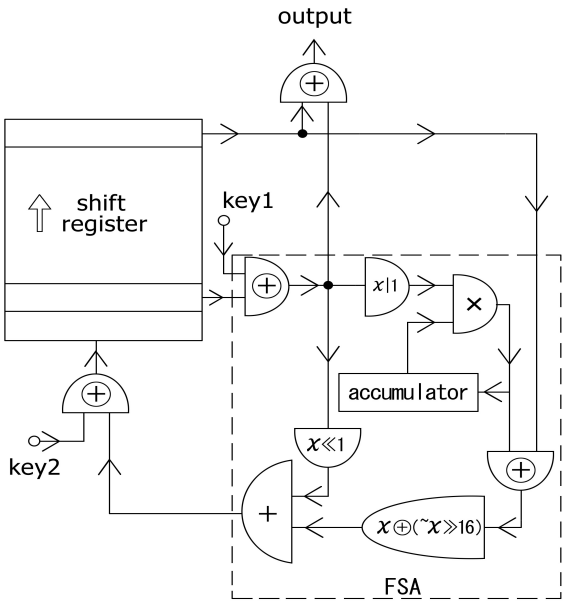


FIGURE 3. The PRNG for the booter

(1) Since the key setup stage is allowed to be slow, we expand the key to two long extended keys, namely two arrays KEY1 and KEY2, by some expanding function.
(2) The booter's inputs are KEY1, KEY2 and IVARRAY. The state space of the booter consists of one cyclic array (a shift register of words) of IVSIZE integers, together with one wordsize memory called the `accumulator`.
(3) In the IV setup, we setup the state space of the booter, and the accumulator of the filter. This is done by copying IVARRAY to the state array of the booter, copying IVARRAY[1] to the accumulator with LSB set to 1, and then by running the booter 2×IVSIZE times without outputting (for idling), except for the IVSIZE-th output which is copied to `accum`, the accumulator of the multiplicative filter, with LSB set to 1.
(4) When the encryption starts, the booter is called to generate one word. The word is used to fill the first member of the state array of the mother generator, as well as the input to the filter. This is iterated N times, namely, until the state space of the mother generator is initialized.
(5) After N steps, the generation by the mother generator starts.

The PRNG used as the booter is described in Figure 3. Every line in the figure denotes w=32-bit data. The bit-wise EOR is denoted by $\oplus$, the integer multiplication (summation) modulo 2^{32} is denoted by $\times$ (+), respectively. The right bottom $x \oplus (\tilde{}x>>16)$ means the following: $\tilde{}x$ denotes the bit-wise inversion of x, $>> 16$ is the shift to the right by 16 bits. Thus, the formula denotes a function mapping x to $x \oplus (\tilde{}x>>16)$, which is bijective because it is inverse to itself. The purpose of the right-shift is to feedback the MSBs of the product, which gather the information of

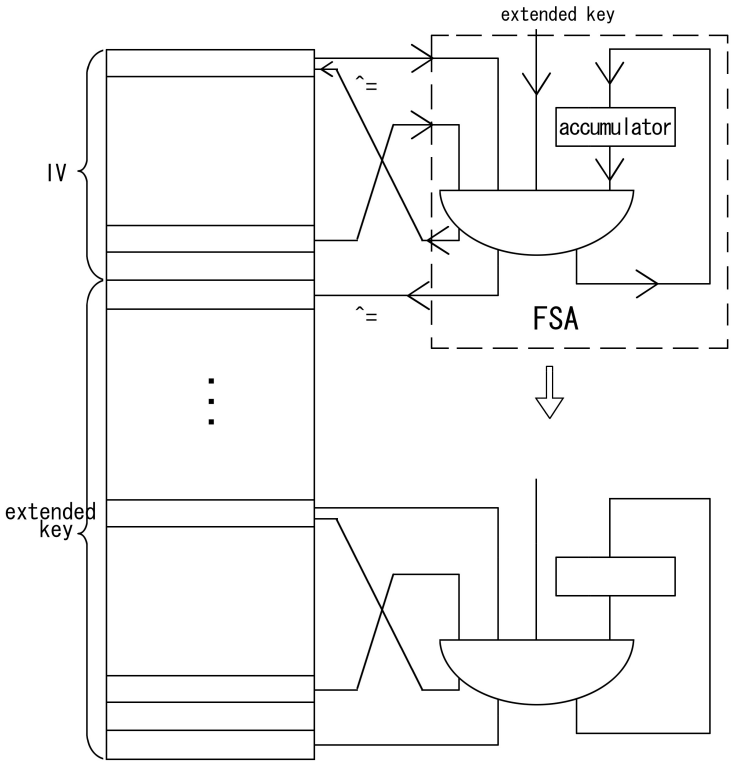


FIGURE 4. The booter generating N words

all bits, to the LSBs, where the information of the higher bits would not be reflected otherwise. The left-shift one-bit function ($x << 1$) below the accumulator in the figure is to pick up the LSB of the middle tap. Without this, the information of LSBs is not well circulated since the LSBs are neglected by the multiplier. The state transition is chosen to be bijective.

The idea of the accumulator comes from the following observation. In a software implementation, we need wordsize variables to compute intermediate results in the computation of the recursion. Usually, the variables are reset by some part of the shift register at every generation. However, we may use the variable as a part of the state space, with paying little cost at the generation stage.

An actual implementation of the booter is pictorially described in Figure 4. It has a shape similar to the Turing machine. The finite state automaton (FSA) at the right-top in Figure 4, having three inputs and two outputs, is the right-bottom box in Figure 3. The IV is copied to the top of the array at the left of Figure 4, and KEY2 is copied below it, while KEY1 is input to the FSA one by one. The output of Figure 3 is written in the same array. The FSA is moved one-step below for each

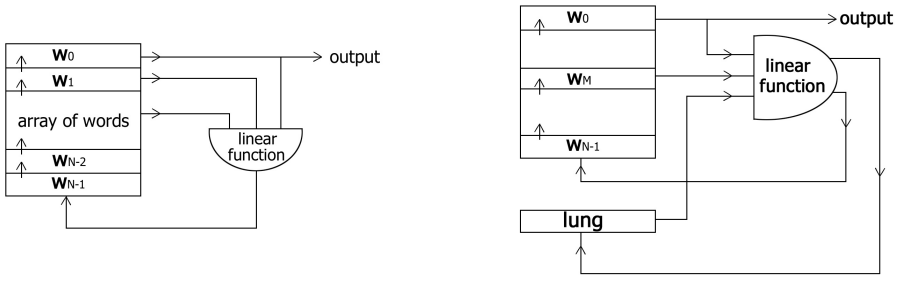


FIGURE 5. Left: a standard LFSR. Right: a pulmonary LFSR

generation. The KEY2 is already copied to the array, so no need to input to FSA: in the C-like notation, `^=` suffices. At the IV setup, we run the booter 2×IVSIZE times for discarding first outputs. Then, the booter's output is used for the first N steps of encryptions. This configuration automatically records the outputs of the booter in the array. Thus, to initialize the mother generator, it suffices to copy N words from the array to the state array of the mother generator (or, we may put a pointer to the array, to use it as the state array of the mother generator.) Because of the idling for 2×IVSIZE steps, it is necessary to prepare $N+2\times$IVSIZE of extended keys in each of KEY1 and KEY2.

The key extension is done by the same method. The same FSA in Figure 3 is used, where the size of the shift register in Figure 3 is KEYSIZE. As for the two inputs, KEY2 is set to all zeroes and KEY1[j]:=j+IVSIZE−2, for $j = 0, 1, \ldots$. In the key setup, the KEYARRAY is copied to the shiftregister, and the KEYARRAY[1] is copied to the accumulator of the booter with LSB set to 1. Then we generate and discard the first 2×KEYSIZE outputs. Then we generate (N+2×IVSIZE) outputs and copy to KEY1, and again generate (N+2×IVSIZE) outputs and copy to KEY2.

3. An improved mother generator PMT

In a typical filtered generator, the mother generator is chosen to be a linear feedbacked shift register (LFSR) described in the left of Figure 5. Here each word is regarded as a w-dimensional vector over $\mathbb{F}_2$, and the feedback is a linear function. MT is one of these.

In [2], we introduced the pulmonary LFSR, described in the right half of Figure 5 (its name was Hearty Twister: we changed the name according to a suggestion by Art Owen). The difference is the existence of one variable `lung` as a component in the state space. This introduces a short length feedback, and improves the dependency on the initial state. The name of "lung" comes from the blood circulating systems of fish and Amphibia. Regard the linear function as the heart, and the array as the body. Then, the standard LFSR has a single loop similarly to the fish, and the pulmonary LFSR has two feedback loops similarly to the Amphibia.

Suppose that the feedback function is a sparse linear function. If the bits in the array contain too many 0's and only small number of 1's, that is, the (Hamming) weight of the array is too small, (like anoxia: 1's are considered as oxygen), then the tendency continues for long in the standard LFSR. The recovery is faster in the

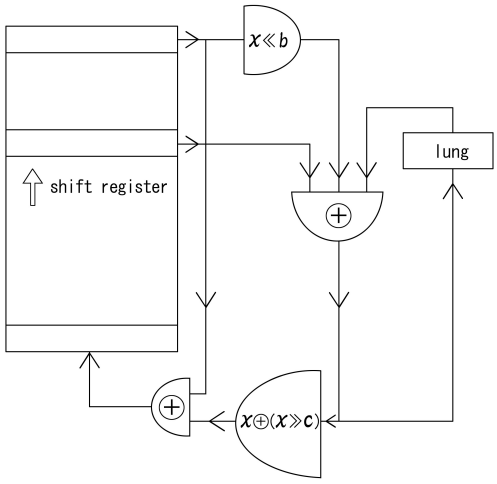


FIGURE 6. Pulmonary Mersenne Twister: Light Version

pulmonary LFSR because of the short cycle containing the lung, which recovers the weight of the lung quickly.

The standard LFSR can be described by a single recursion of order N, but the pulmonary LFSR requires two recursions. The example in Figure 5 is given by

$$u_{i+1} := F_1(x_{i+M}, x_i, u_i)$$
$$x_{i+N} := F_2(x_{i+M}, x_i, u_i),$$

where x_i denotes the content of the i-th member of the array and u_i denotes the content of the lung. We propose to use Pulmonary Mersenne Twister-Light-19937 (PMTL19937), whose recursion is given by

$$u_{i+1} := (x_i << b) \oplus x_{i+M} \oplus u_i;$$
$$x_{i+N} := x_i \oplus R_c(u_{i+1}),$$

where $R_c(x) := x \oplus (x >> c)$ with parameters specified by $N = 623$, $M = 609$, $b = 7$ and $c = 3$. Pictorial description is in Figure 6.

We checked the following by using a computer and mathematical algorithms based on the Berlekamp-Massey method and Lenstra's lattice method. For the detail, we plan to write a paper on PMT.

Proposition 3.1. PMTL19937 is an automaton with $19968 = 32 \times 624$ bits of state space S, which consists of an array of 623 words and a 32-bit memory *lung*.

(1) The transition function h of PMTL19937 is an $\mathbb{F}_2$-linear bijection, whose characteristic polynomial is factorized as

$$\chi_h(t) = \chi_{19937}(t) \times \chi_{31}(t),$$

where $\chi_{19937}(t)$ is a primitive polynomial of degree 19937 and $\chi_{31}(t)$ is a polynomial of degree 31.

(2) The state S is uniquely decomposed into a direct sum of h-invariant subspaces of degrees 19937 and 31

$$S = V_{19937} + V_{31},$$

where the characteristic polynomial of h restricted to V_{19937} is $\chi_{19937}(t)$.

(3) From any initial state s_0 not contained in V_{31}, the period P of the state transition is a multiple of the 24th Mersenne Prime $2^{19937}-1$, namely $P=(2^{19937}-1)q$ holds for some $1 \le q \le 2^{31}-1$ (q may depend on s_0). The period of the output sequence is also P.

In this case, in addition, the sequence of the most significant 31 bits of each output integer is 624-dimensionally equidistributed with defect q in the sense of [5, §4.4] (one dimension larger than MT).

(4) There is a 32-dimensional constant vector v such that if the lung-part of s_0 coincides with v, then $s_0 \notin V_{31}$. We set the lung to this value at the initialization.

(5) $\chi_h(t)$ has 205 nonzero terms (which is larger than 135 of MT), and $\chi_{19937}(t)$ has 9945 nonzero terms.

There are a few more advantages of PMT over MT. Firstly, because of the simplicity of the recursion, the generation speed is a little faster than MT. Secondly, one can eliminate u_i from the recursion to obtain

$$x_{i+N} = x_i + x_{i+1} + x_{i+N-1} + R_c(x_i<<b) + R_c x_{i+M},$$

which shows that there are 5-bit relations among consecutive 624 outputs of this PMT (in the case of MT, there are 3-bit relations).

By the way, the above choice of the recursion is to keep the high speed, and is not the best one from the viewpoint of random number generation for MonteCarlo purpose. We will explain this in a forthcoming paper.

4. Resistance of CryptMTV2 to Standard Attacks

CryptMTV2.0 (CryptMT Version 2.0) is the above modified generator obtained from CryptMT by changing the initialization and the mother generator. The cryptanalysis developed in §4 in [5] for CryptMT is equally valid to CryptMTV2.0, which we briefly recall.

Time-memory-trade-off attack. A naive time-memory-tradeoff attack consumes the computation time of roughly the square root of the size of the state space, which is $O(\sqrt{2^{19968+31}}) = O(2^{9999.5})$ for CryptMTV2.0.

Dimension of Equidistribution. As stated in Proposition 3.1, PMTL19937 satisfies all conditions in §4.2–§4.3 of loc. cit., with period $P=(2^{19937}-1)q$, $n=624$-dimensional equidistribution with defect $d=q$. Proposition 4.4 (loc. cit.) implies that CryptMTV2.0 (more precisely, its indistinguishable modification stated in Assumption 4.3 there) is 625-dimensionally equidistributed with defect $q \cdot 2^{31} < 2^{62}$.

Correlation attacks and distinguishing attack. By Corollary 4.7 (loc. cit.), if we consider a simple distinguishing attack to CryptMTV2.0 of order $N \le 625$, then its security level is $2^{19937\times 2}$, since $P/d = 2^{19937}-1$.

Correlation attacks based on a four-term relation is infeasible, since the computational complexity to find such a relation is of order of $O(N \log N)$, where $N \ge 2^{19937/3}$ for CryptMTV2.0.

Algebraic degree of the filter. Proposition 4.11 (loc. cit.) is about the multiplicative filter, so it is valid for CryptMTV2.0 as it is. This gives a supportive evidence to that each bit of the output of CryptMTV2.0 would have high algebraic degree, close to the upper bound coming from the number of variables. The experimental results by the toy models stated in the next section also support this, so algebraic attacks and Berlekamp-Massey attacks would be infeasible, by the same reasons stated in §4.9 and §4.10 of loc. cit.

5. Simulation by toy models

We consider all bits in the initial state as variables, and then each bit of the outputs is a boolean function of these variables, so algebraic degree and non-linearity are defined. However, they are hard to compute because of the size of the state space. Similarly to §4.8 of loc. cit., we made a toy model and obtained experimental results. Since the mother generator of CryptMTV2.0 is a PMT, we made a toy model of 16-bit state space, which generates a 16-bit integer sequence defined by

$$\begin{aligned}\mathbf{t} &:= \mathbf{x}_j \oplus (\mathbf{x}_j << 7)\\ \mathbf{x}_{j+1} &:= \mathbf{t} \oplus (\mathbf{t} >> 3)\end{aligned}$$

where $\mathbf{t}$ is a temporary 16-bit variable and $\mathbf{x}_j$ is a 16-bit integer, and then it is filtered by

$$y_{j+1} = (\mathbf{x}_j|1) \times y_j \mod 2^{16}.$$

We put $y_0 = 1$, and compute the algebraic degree of each of the 16 bits in the outputs $y_1 \sim y_{16}$, each regarded as a polynomial function with 16 variables being the bits in $\mathbf{x}_0$. The result is listed in Table 1. The lower six bits of the table clearly show the pattern $0, 1, 1, 2, 4, 8$, whereas the eighth bit and higher are "saturated" to the upper bound 16, after 8 generations, which is slightly better than 12 generations for the toymodel of CryptMT, see Table 1, loc. cit.

We expect that the same will occur for CryptMTV2.0. So, if we consider each bit of the internal state of MT as a variable, then the algebraic degree of the 8 MSBs of y_i will be near to 19968, after some steps of generations.

Also, we computed the non-linearity of the MSB of each y_i $(i = 1, 2, \ldots, 8)$ of this toy model. The result is listed in Table 2, and each value is near to 2^{16-1}. This suggests that there would be no good linear approximation of CryptMTV2.0, similarly to CryptMT.

6. Differential attacks on IV and Key

So far, we do not argue on the attacks at the resynchronization. Since the first 623 outputs of CryptMTV2.0 is the filtered output of the booter, we need to discuss on the resistance of the booter with multiplicative filter.

As a first step to the cryptanalysis of the booter, we conducted a statistical test based on a naive differential attack. We set the extended keys KEY1 and KEY2 both to all zeroes. Then we consider the booter as functions $B_n(\mathrm{IV})$, which maps the IV to the n-th output of the booter initialized by that IV. We fix a 256-bit (8-word) IV. Then, we compute

$$\Delta(\mathrm{IV}, i) := B_n(\mathrm{IV} \oplus E_i) \oplus B_n(\mathrm{IV})$$

for $E_1, \ldots, E_{256}$ being the 256-dimensional unit vectors (i.e., of Hamming weight one). The Hamming weight of $\Delta(\mathrm{IV}, i)$ should conform to the binomial distribution

TABLE 1. Table of the algebraic degrees of output bits of a toy model.

y_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
y_2	15	15	15	14	13	12	10	8	7	6	4	3	2	1	1	0
y_3	16	16	15	16	15	15	13	11	9	7	5	3	2	1	1	0
y_4	15	16	16	15	15	15	15	13	12	9	6	4	2	1	1	0
y_5	15	16	16	16	15	16	16	16	13	9	6	4	2	1	1	0
y_6	16	15	15	15	16	15	15	16	15	11	7	4	2	1	1	0
y_7	16	15	15	15	15	15	16	16	15	11	7	4	2	1	1	0
y_8	16	16	16	15	16	16	15	15	16	12	8	4	2	1	1	0
y_9	15	15	16	16	15	15	16	16	15	12	8	4	2	1	1	0
y_{10}	16	15	15	16	15	15	15	16	16	12	8	4	2	1	1	0
y_{11}	15	16	15	16	15	16	16	16	15	14	8	4	2	1	1	0
y_{12}	15	15	16	15	16	15	16	15	16	13	8	4	2	1	1	0
y_{13}	16	15	16	15	16	16	16	15	16	14	8	4	2	1	1	0
y_{14}	15	16	16	15	16	15	15	16	15	15	8	4	2	1	1	0
y_{15}	16	15	16	15	16	16	16	15	16	14	8	4	2	1	1	0
y_{16}	16	15	15	16	15	16	15	16	15	16	8	4	2	1	1	0

TABLE 2. The non-linearity of the MSB of each output of a toy model.

output	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
nonlinearity	0	32118	32246	32206	32218	32165	32233	32103	32213

$B(32, 1/2)$ for an ideal booter. We have 256 samples of the Hamming weights for $i = 1, 2, \ldots, 256$. We choose 1000 random samples of IV, and thus 256000 samples of Hamming weights, for each $1 \leq n \leq 24$. We separate 33 weights into 9 categories

$$\{0...12\}, \{13\}, \{14\}, \{15\}, \{16\}, \{17\}, \{18\}, \{19\}, \{20...32\}$$

and conduct χ^2-tests. The corresponding p-values are listed in Table 3. We iterated this five times. The p-values show that the first 9 outputs are deviated, but the 10th and after seem to be O.K. In the initialization, the booter discards 2×IVSIZE=16 outputs, which seem to be enough.

7. Performance comparison

We used the performance testing tool from eSTREAM [1] to see the speed of the IV setup with the platform Pentium-M 1.4GHz. The original version consumes 31113 cycles for IV setup, while CryptMTV2.0 consumes 2145 cycles, namely, speed-up by a factor of 15. Accordingly, the cycles per byte to encrypt 40 bytes is reduced from 806 to 74. However, the key-setup time is increased from 34 cycles to 22487 cycles. Also, the column STREAM (measuring the time for long stream without IV setup) shows 2% slow-down compared to the original version. Probably this is because the first block is ciphered by the booter, which is slower than PMT.

8. conclusion

We introduced a method to initialize a huge state space with little cost, by using a booter, a smaller PRNG. This solves the slowness in the IV setup of the first version of CryptMT. However, we need to test the resistance of the booter, too.

TABLE 3. The p-values of the Hamming weight test of the n-th output of the booter (0 suppressed).

Outputs	1st	2nd	3rd	4th	5th
B_1	1.	1.	1.	1.	1.
B_2	1.	1.	1.	1.	1.
B_3	1.	1.	1.	1.	1.
B_4	1.	1.	1.	1.	1.
B_5	1.	1.	1.	1.	1.
B_6	1.	1.	1.	1.	1.
B_7	1.	1.	1.	1.	1.
B_8	0.999858	1.	0.999884	0.99988	1.
B_9	1.	1.	0.999968	1.	0.926248
B_{10}	0.415646	0.10617	0.369702	0.810966	0.0591573
B_{11}	0.349149	0.269581	0.546788	0.0783579	0.478834
B_{12}	0.656057	0.904608	0.719275	0.709268	0.886417
B_{13}	0.0636272	0.292971	0.439085	0.926816	0.354477
B_{14}	0.994904	0.388312	0.688698	0.0523952	0.610518
B_{15}	0.943661	0.457131	0.173981	0.34268	0.659302
B_{16}	0.806287	0.313299	0.211509	0.495947	0.762681
B_{17}	0.892633	0.514589	0.552164	0.0554408	0.3439
B_{18}	0.44802	0.344326	0.578483	0.963813	0.665435
B_{19}	0.441611	0.355715	0.0319679	0.216351	0.828746
B_{20}	0.0219037	0.775335	0.445655	0.653318	0.330011
B_{21}	0.0359443	0.86928	0.791367	0.238231	0.751933
B_{22}	0.434032	0.119962	0.19941	0.013384	0.626764
B_{23}	0.469654	0.113235	0.539935	0.482852	0.0602773
B_{24}	0.739223	0.197051	0.917797	0.643172	0.8482

We experimented a simple differential attack on IV to the booter, and the result was satisfactory. Actually, we may use any block cipher as the booter, as far as they have enough strength, so we have plenty of choice.

REFERENCES

[1] eSTREAM Optimized Code Howto, `http://www.ecrypt.eu.org/stream/perf/`.

[2] Haramoto, H., Panneton, F., Nishimura, T., and Matsumoto, M. Hearty Twister: a new random number generator, a talk in Fifth IMACS seminar on Monte-Carlo Method MCM2005, 2005 May at Florida State University.

[3] Matsumoto, M. and Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 8 (1998) 3–30.

[4] Matsumoto, M., Nishimura, T., Saito, M. and Hagita, M. Cryptographic Mersenne Twister and Fubuki stream/block cipher, `http://eprint.iacr.org/2005/165`.
This is an extended version of "Mersenne Twister and Fubuki stream/block cipher" submitted for eSTREAM proposal `http://www.ecrypt.eu.org/stream/`.

[5] Matsumoto, M. Saito, M., Nishimura, T. and Hagita, M. Cryptanalysis of CryptMT: Effect of Huge Prime Period and Multiplicative Filter, to appear in SASC2006 Conference Volume `http://www.ecrypt.eu.org/stream/`.

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: m-mat@math.sci.hiroshima-u.ac.jp

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: saito@math.sci.hiroshima-u.ac.jp

Department of Mathematics, Yamagata University, Yamagata Japan
E-mail address: nisimura@sci.kj.yamagata-u.ac.jp

Department of Information Science, Ochanomizu University, Tokyo Japan
E-mail address: hagita@is.ocha.ac.jp

T-function based streamcipher TSC-4

Dukjae Moon, Daesung Kwon, Daewan Han, Jooyoung Lee, Gwon Ho Ryu,
Dong Wook Lee, Yongjin Yeom, and Seongtaek Chee

National Security Research Institute
161 Gajeong-dong, Yuseong-gu
Daejeon, 305-350, Korea
{djmoon,ds_kwon,dwh,jlee05,jude,dwlee,yjyeom,chee}@etri.re.kr

Abstract. In this article, we present a synchronous stream-cipher named TSC-4, together with security analysis and implementation results. TSC-4 is designed to be well suited for constrained hardware with an intended security level of 80 bits. With 4×4 s-boxes at its core, the design leaves open the possibility for implementations of very low power consumption. As an improvement of TSC-3, TSC-4 shows better resiliency against distinguishing attacks.

Keywords: TSC-4, T-function, single cycle, streamcipher, s-box, nonlinear filter

1 Introduction

Few years ago, Klimov and Shamir started developing the theory of T-functions[1–3]. A T-function is a function acting on a collection of memory words, with a weak one-wayness property. It started out as a tool for block ciphers, but is now more of a building block for a stream cipher.

An important class of T-functions consists of those with *single cycle* property. Any T-function with single cycle property is equivalent to a LFSR of maximum length, and has potential to construct a very fast stream cipher. Unfortunately, only a small family of single cycle T-functions are known for now.

In 2004, we presented a new class of single cycle T-function[4, 5]. Although previous T-functions targeted software implementations, our T-function was designed to be light and was well suited for constrained hardware. Also, we proposed the stream cipher based on this T-function, TSC-1, TSC-2[5] and TSC-3[6]. We used the T-function to resist against the powerful attacks which are applied to the stream ciphers based on LFSR, such as algebraic attacks [10–12] and correlation attacks[8, 9] and to be possible to work out the period. However, Künzli *et al.* and Muller *et al.* described distinguishing and key recovery attacks against TSC family[13, 14]. This attack was used that our T-function did not offer a sufficient level of diffusion. In order to prevent distinguishing attacks, we modified the cipher by carefully choosing an s-box and a nonlinear function in it.

In this article, we present a synchronous stream-cipher named TSC-4 (T-function based Stream-Cipher ver 4), together with security analysis and implementation results. The main environment of the cipher is targeted to constrained

hardware with an intended security level of 80 bits. With 4×4 s-boxes at its core, the design leaves open the possibility for implementations of very low power consumption.

2 Cipher specification

In this section, we describe specifications of TSC-4, including the internal state, the cipher body and state initialization. As seen in Fig. 1, TSC-4 is a filter generator based on T-functions, whose internal state consists of two 128-bit states of T-functions. After each update, an 8-bit output keystream is produced from the states through a nonlinear filter.

2.1 Internal state of T-function

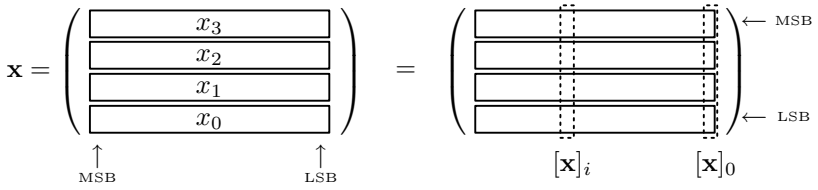
We denote a 128-bit state by

$$\mathbf{x} = (x_k)_{k=0}^{3},$$

where each word $x_k, k = 0, \ldots, 3$ has 32 bits in length. Let $[x]_i, i = 0, \ldots, n-1$ denote the i-th bit of an n-bit word x. Then the word(vector) x will interchangeably represent an integer, if necessary, by the following equation:

$$x = \sum_{i=0}^{n-1} [x]_i 2^i. \tag{1}$$

With the above notations, we can represent each internal state in a matrix form as follows:



Here $[\mathbf{x}]_i$ denotes the i-th column of state $\mathbf{x}$.

2.2 Main body

TSC-4 takes an 80-bit length secret key K and an 80-bit length public initialization vector IV. The structure of TSC-4 is illustrated in Fig. 1.

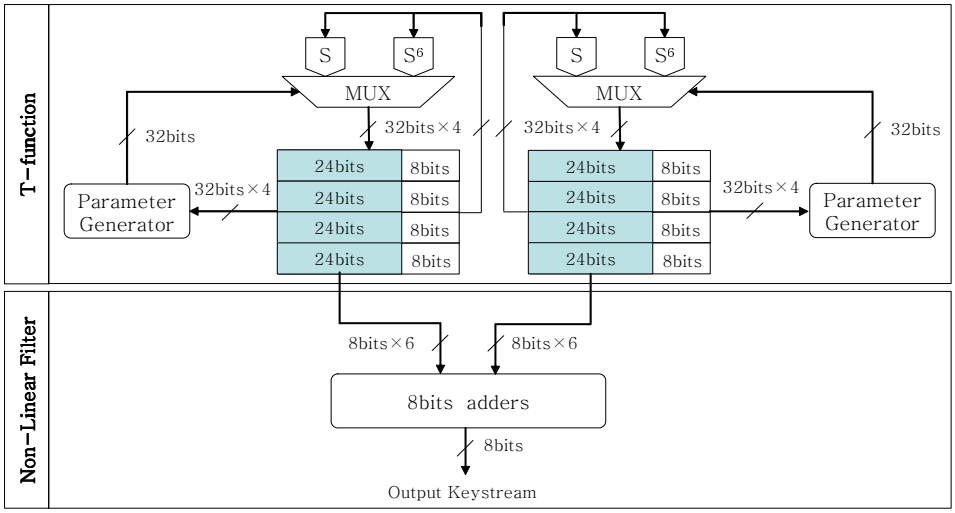


Fig. 1. The structure of TSC-4

Parameters: Two parameters $p_1(\mathbf{x})$ and $p_2(\mathbf{y})$ are defined with a number of temporary variables as follows:

$$\begin{aligned} \pi(\mathbf{x}) &= x_0 \wedge x_1 \wedge x_2 \wedge x_3, \\ o_1(\mathbf{x}) &= \pi(\mathbf{x}) \oplus (\pi(\mathbf{x}) + \texttt{0x51291089}), \\ e(\mathbf{x}) &= (x_0 + x_1 + x_2 + x_3)_{\ll 1}, \\ p_1(\mathbf{x}) &= o_1(\mathbf{x}) \oplus e(\mathbf{x}), \\ \\ \pi(\mathbf{y}) &= y_0 \wedge y_1 \wedge y_2 \wedge y_3, \\ o_2(\mathbf{y}) &= \pi(\mathbf{y}) \oplus (\pi(\mathbf{y}) + \texttt{0x12910895}), \\ e(\mathbf{y}) &= (y_0 + y_1 + y_2 + y_3)_{\ll 1}, \\ p_2(\mathbf{y}) &= o_2(\mathbf{y}) \oplus e(\mathbf{y}), \end{aligned} \tag{2}$$

where $\wedge$, $\oplus$ and $\ll$ denote bitwise `AND`, bitwise `XOR` operation, and left shift of 32-bit words, respectively. The additions are done modulo 2^{32} using the equation (1). Note that $o_i, i = 1, 2$ are *odd parameters* and e is an *even parameter* [5].

S-box application: We fix a 4×4 s-box S, defined in C-language style as follows:

$$S\texttt{[16] = \{9,2,11,15,3,0,14,4,10,13,12,5,6,8,7,1\};} \tag{3}$$

Now T-functions $\mathbf{T}_i, i = 1, 2$ on input states $\mathbf{x}$, $\mathbf{y}$ are defined as follows:

$$[\mathbf{T}_1(\mathbf{x})]_i = \begin{cases} S\ ([\mathbf{x}]_i) & \text{if } [\mathbf{p}_1(\mathbf{x})]_i = 1, \\ S^6([\mathbf{x}]_i) & \text{if } [\mathbf{p}_1(\mathbf{x})]_i = 0, \end{cases} \tag{4}$$

$$[\mathbf{T}_2(\mathbf{y})]_i = \begin{cases} S\ ([\mathbf{y}]_i) & \text{if } [\mathbf{p}_2(\mathbf{y})]_i = 1, \\ S^6([\mathbf{y}]_i) & \text{if } [\mathbf{p}_2(\mathbf{y})]_i = 0, \end{cases} \tag{5}$$

where the columns $[\mathbf{x}]_i$, $[\mathbf{T}_1(\mathbf{x})]_i$, $[\mathbf{y}]_i$ and $[\mathbf{T}_2(\mathbf{y})]_i$ are regarded as 4-bit integers by the equation (1).

Nonlinear filter: The filter produces the actual output keystream from the current internal states. We compute six 8-bit temporary variables $(a_0, \cdots, a_5)$ as follows:

$$\begin{aligned} a_0 &= ((x_3)_{\gg 24} \ \wedge\ \texttt{0xff}) + ((y_1)_{\gg 8} \ \wedge\ \texttt{0xff}), \\ a_1 &= ((x_0)_{\gg 24} \ \wedge\ \texttt{0xff}) + ((y_2)_{\gg 8} \ \wedge\ \texttt{0xff}), \\ a_2 &= ((x_2)_{\gg 16} \ \wedge\ \texttt{0xff}) + ((y_3)_{\gg 16} \ \wedge\ \texttt{0xff}), \\ a_3 &= ((x_1)_{\gg 16} \ \wedge\ \texttt{0xff}) + ((y_0)_{\gg 16} \ \wedge\ \texttt{0xff}), \\ a_4 &= ((x_3)_{\gg 8} \ \wedge\ \texttt{0xff}) + ((y_2)_{\gg 24} \ \wedge\ \texttt{0xff}), \\ a_5 &= ((x_0)_{\gg 8} \ \wedge\ \texttt{0xff}) + ((y_1)_{\gg 24} \ \wedge\ \texttt{0xff}), \end{aligned} \tag{6}$$

where the additions are done modulo 2^8. Now the 8-bit keystream z is defined to be

$$z = a_0 \ \oplus\ (a_1)_{\ggg 5} \ \oplus\ (a_2)_{\ggg 2} \ \oplus\ (a_3)_{\ggg 5} \ \oplus\ (a_4)_{\ggg 6} \ \oplus\ (a_5)_{\ggg 2}, \tag{7}$$

where $\ggg$ denote rotation to the right.

2.3 State initialization

We now describe how the state is initialized from a given key and an IV. The internal state consists of 8 words as seen in Fig. 2.

$$\mathbf{x} = \begin{pmatrix} \boxed{x_3} \\ \boxed{x_2} \\ \boxed{x_1} \\ \boxed{x_0} \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} \boxed{y_3} \\ \boxed{y_2} \\ \boxed{y_1} \\ \boxed{y_0} \end{pmatrix}$$

Fig. 2. Internal state of TSC-4

Key/IV Loading: Let $K = (k_{79}, k_{78}, \cdots, k_1, k_0)$ and $IV = (iv_{79}, iv_{78}, \cdots, iv_1, iv_0)$ be an 80-bit key and an 80-bit IV, respectively. Then the internal state is initialized as follows:

1. $x_0 = (k_{31}, k_{30}, \cdots, k_1, k_0)$
2. $x_1 = (k_{63}, k_{62}, \cdots, k_{33}, k_{32})$
3. $x_2 = (iv_{31}, iv_{30}, \cdots, iv_1, iv_0)$
4. $x_3 = (iv_{63}, iv_{62}, \cdots, iv_{33}, iv_{32})$
5. $y_0 = (iv_{15}, \cdots, iv_0, iv_{79}, \cdots, iv_{64})$
6. $y_1 = (iv_{47}, iv_{46}, \cdots, iv_{17}, iv_{16})$
7. $y_2 = (k_{15}, \cdots, k_0, k_{79}, \cdots, k_{64})$
8. $y_3 = (k_{47}, k_{46}, \cdots, k_{17}, k_{16})$

Warm-up: Once the internal state is initialized, the K and IV are mixed by the following process.

1. Run cipher body once to produce a single 8-bit output.
2. Rotate x_1 and y_0 to the left by 8 bits.
3. XOR the output to the least significant 8 bits of x_1 and y_0.

The key and IV setup is completed by repeating the above three steps by eight times.

3 Security

TSC-4 is intended for 80-bit security. For the moment, the best attack on TSC-4 we know of is the brute force attack of complexity 2^{80}.

3.1 Statistical tests

We have done tests similar to the ones presented in [7] and have verified that this proposal gives good statistical results.

3.2 Period

The period of TSC-4 is 2^{128}. To see this, we already know that the period of each T-function is 2^{128}, as guaranteed by the single cycle property [5]. So, first note that the period of TSC-4 has to be a divisor of 2^{128}. Now, initialize two register contents with the all zero state and consider what each content of the registers would be after 2^{124} iterated applications of the T-function. Since the period of each T-function restricted to the lower 31 columns is 2^{124}, all columns except the most significant column should be zero. Now we can show that there exists a nonzero bit in the output 8-bit keystream, since the most significant columns determine the i-th output bit for i=1, 2, 5, 7. Furthermore, when observed every 2^{124} iterations apart, due to description (4) and (5) and the definition of an odd

parameter, the change of the most significant columns follow some fixed odd power of the S-box, which is of cycle length 16. Explicit calculation of the 16 keystream output words for each odd power of the s-box confirms that, in all odd power cases, one has to go through all 16 points before reaching the starting point. Hence the period of the cipher is $16 \cdot 2^{124} = 2^{128}$.

3.3 Correlation attack

Difficulty of correlation attacks can also be obtained from the rotations in the filter. In the last step of a correlation attack, one needs to guess a part of the state and compare calculated outputs with the actual keystream, checking for the occurrence of expected correlation.

In our situation, any correlation found to exist with a single output bit will involve multiple input bits. Hence correlation attacks do not seem to be applicable.

3.4 Algebraic attack

In many cases, algebraic attacks are possible on stream ciphers built on LFSRs. Once a single equation connecting the internal state to the output keystream is worked out, the cipher logic can be run forward to produce more such equations. During this process, the linear property of LFSRs keep the degree of new equations equal to the first equation. And this is the main reason for the success of algebraic attacks on streamciphers.

In the case of TSC-4, the source of randomness, i.e., the T-function, is already nonlinear. During the action of T-functions $\mathbf{T}_1$ and $\mathbf{T}_2$ on internal states $\mathbf{x}$ and $\mathbf{y}$, the degree of new equation increase in the degree of a previous equation. Hence algebraic attacks do not seem to be applicable.

3.5 Guess-then-determine attack

One property of T-functions, that could be bad from the viewpoint of security, is that it can be restricted to any number of its lower columns. In other words a part of internal state of T-function can be guessed and run forward indefinitely, opening up the possibility of a guess-then-determine attack.

The rotations used in the filter eliminates this weakness. They have been chosen so that any single output bit receives direct effect of more twelve bits that are spread widely apart within two states. So it is not possible to calculate any output bit with the information of any small number of internal states.

Even if all modular additions in the filter were replaced with XORs, in order to calculate any one of the 8 output bits continuously, one would need to guess 96 bits (8×12 bits), so no meaningful attack can be achieved through this approach.

3.6 Distinguishing attack

Bit-flip probability: We have chosen the s-box (3) to satisfy the following conditions.

1. At the application of S, each of the four bits has bit-flip probability of $\frac{1}{2}$.
2. The same is true for S^6.

More precisely, the first condition states that

$$\#\{\, 0 \le t < 16 \mid \text{the } k\text{-th bit of } t \oplus S(t) \text{ is } 1\} = 8,$$

for each $k = 0, 1, 2, 3$. Due to this property, regardless of the behavior of the odd parameters $\mathbf{p}_1(\mathbf{x})$ and $\mathbf{p}_2(\mathbf{y})$, every bit in the state is guaranteed to have bit-flip probability $\frac{1}{2}$ at the action of $\mathbf{T}$.

Bit-flip bias of multiple applications of T-function: There are strong distinguishing attacks[13, 14] applicable to previous versions[5, 6] of this cipher. The main observation used in the attack is that even though the bit-flip probability of T-function is close to $\frac{1}{2}$, this is not true for its multiple applications. This property is still present in the current design. However, TSC-4 is designed to be resistant to the distinguishing attacks by taking the following cases into account:

Case 1 The strongest bit-flip bias between the same bit position for multiple applications. The algorithms TSC-1 and TSC-2[5] are analyzed using this property[13, 14]. In this case, we deal with the bias of $[z]_i^t \oplus [z]_i^{t+\delta}$, where δ is the number of iterations of T-function.

Case 2 The strongest bit-flip bias between the distinct bit position in the same column for multiple applications. The algorithm TSC-3[6] is analyzed using this property[14]. In this case, we deal with the bias of $[z]_i^t \oplus [z]_j^{t+\delta}, i \neq j$.

Case 3 The strongest bit-flip bias between the linear relations of the same bits for multiple applications. This property is considered in this paper. In this case, we deal with the bias of $[z]_i^t \oplus [z]_j^t \oplus [z]_j^{t+\delta} \oplus [z]_j^{t+\delta}, i \neq j$.

Table 1. Bit-flip bias of $[x_k]_i^t = [x_k]_i^{t+\delta}$ $(1 \le \delta \le 15)$

δ	1	2	3	4	5	6	7	8
$\lvert log_2\varepsilon \rvert$	∞	∞	**5**	6	7	6	∞	8.42
δ	9	10	11	12	13	14	15	$\cdots$
$\lvert log_2\varepsilon \rvert$	9.42	7.42	13	9.91	6.25	7.94	10.71	$\cdots$

First of all, we could obtain the property that a bit-flip bias between the same bit positions for δ $(1 \le \delta \le 1000)$ iterations of T-function is less than 2^{-5} through the experiments (Fig. 3). The pattern of the plot in Fig. 3 suggests that the property holds for $\delta > 1000$ iterations. Table 1 shows the exact bit-flip bias

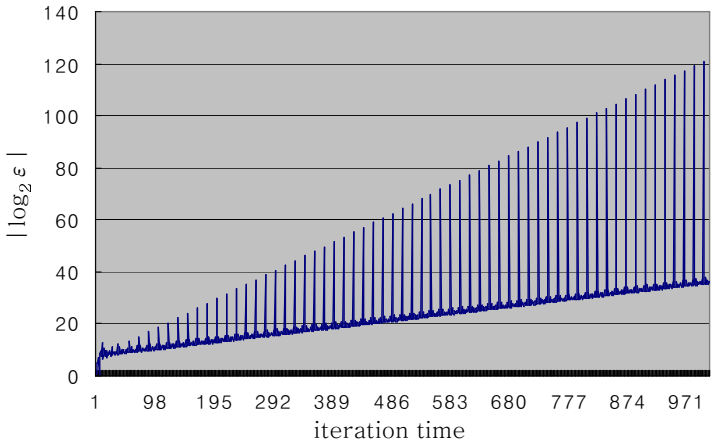


Fig. 3. Bit-flip bias of $[x_k]_i^t = [x_k]_i^{t+\delta}$ $(1 \leq \delta \leq 1000)$

"ε"[1] between the same bit positions after δ $(1 \leq \delta \leq 15)$ times iteration, where $\mathbf{x}^{t+\delta}$ denote $\mathbf{T}^{\delta}(\mathbf{x}^t)$. By using the nonlinear filter, we can obtain a linear relation of the output filter like this $(i = 0, \cdots, 7)$:

$$\begin{aligned}[z]_i^t \oplus [z]_i^{t+\delta} = &([a_0]_i^t \oplus [a_0]_i^{t+\delta}) \oplus ([a_1]_{i+5(\bmod 8)}^t \oplus [a_1]_{i+5(\bmod 8)}^{t+\delta}) \\ &\oplus ([a_2]_{i+2(\bmod 8)}^t \oplus [a_2]_{i+2(\bmod 8)}^{t+\delta}) \oplus ([a_3]_{i+5(\bmod 8)}^t \oplus [a_3]_{i+5(\bmod 8)}^{t+\delta}) \\ &\oplus ([a_4]_{i+6(\bmod 8)}^t \oplus [a_4]_{i+6(\bmod 8)}^{t+\delta}) \oplus ([a_5]_{i+2(\bmod 8)}^t \oplus [a_5]_{i+2(\bmod 8)}^{t+\delta}).\end{aligned}$$

In this relation, each $[a_k]_i^t \oplus [a_k]_i^{t+\delta}$ $(k = 0, \cdots, 5)$ is approximated as a linear relation like this:

$$\begin{aligned}
[a_0]_i^t \oplus [a_0]_i^{t+\delta} &= [x_3]_{i+24}^t \oplus [x_3]_{i+24}^{t+\delta} \oplus [y_1]_{i+8}^t \oplus [y_1]_{i+8}^{t+\delta} \oplus R_0(i), \\
[a_1]_{i+5(\bmod 8)}^t \oplus [a_1]_{i+5(\bmod 8)}^{t+\delta} &= [x_0]_{i+24}^t \oplus [x_0]_{i+24}^{t+\delta} \oplus [y_2]_{i+8}^t \oplus [y_2]_{i+8}^{t+\delta} \oplus R_1(i), \\
[a_2]_{i+2(\bmod 8)}^t \oplus [a_2]_{i+2(\bmod 8)}^{t+\delta} &= [x_2]_{i+16}^t \oplus [x_2]_{i+16}^{t+\delta} \oplus [y_3]_{i+16}^t \oplus [y_3]_{i+16}^{t+\delta} \oplus R_2(i), \\
[a_3]_{i+5(\bmod 8)}^t \oplus [a_3]_{i+5(\bmod 8)}^{t+\delta} &= [x_1]_{i+16}^t \oplus [x_1]_{i+16}^{t+\delta} \oplus [y_0]_{i+16}^t \oplus [y_0]_{i+16}^{t+\delta} \oplus R_3(i), \\
[a_4]_{i+6(\bmod 8)}^t \oplus [a_4]_{i+6(\bmod 8)}^{t+\delta} &= [x_3]_{i+8}^t \oplus [x_3]_{i+8}^{t+\delta} \oplus [y_2]_{i+24}^t \oplus [y_2]_{i+24}^{t+\delta} \oplus R_4(i), \\
[a_5]_{i+2(\bmod 8)}^t \oplus [a_5]_{i+2(\bmod 8)}^{t+\delta} &= [x_0]_{i+8}^t \oplus [x_0]_{i+8}^{t+\delta} \oplus [y_1]_{i+24}^t \oplus [y_1]_{i+24}^{t+\delta} \oplus R_5(i),
\end{aligned}$$

where $R_k(i)$ $(k = 0, \cdots, 5)$ represents the carry bit. By using the above linear approximation, we have a plausible argument that show the bit-flip bias of filter output to be much less than $2^{-49} (= 2^{-1} \times (2^{-4})^{12})$. The bit-flip bias is approximated using the *Piling-up Lemma* in case of $\delta = 3$. In order to detect this bias, data size of more than 2^{98} is needed.

[1] If $\varepsilon = 0$ then we represent $|log_2 \varepsilon|$ as "∞"

Table 2. Bit-flip bias of $[x_k]_i^t = [x_{k'}]_i^{t+\delta}$ ($|\ log_2\varepsilon|$)

case $\delta = 1$

input \ output	$[x_0]_i^{t+1}$	$[x_1]_i^{t+1}$	$[x_2]_i^{t+1}$	$[x_3]_i^{t+1}$
$[x_0]_i^t$	∞	4	4	∞
$[x_1]_i^t$	3	∞	∞	4
$[x_2]_i^t$	∞	3	∞	4
$[x_3]_i^t$	4	∞	3	∞

case $\delta = 2$

input \ output	$[x_0]_i^{t+2}$	$[x_2]_i^{t+2}$	$[x_2]_i^{t+2}$	$[x_3]_i^{t+2}$
$[x_0]_i^t$	∞	∞	∞	∞
$[x_1]_i^t$	∞	∞	∞	∞
$[x_2]_i^t$	∞	∞	∞	∞
$[x_3]_i^t$	∞	∞	∞	∞

case $\delta = 3$

input \ output	$[x_0]_i^{t+3}$	$[x_1]_i^{t+3}$	$[x_2]_i^{t+3}$	$[x_3]_i^{t+3}$
$[x_0]_i^t$	5	∞	5	5
$[x_1]_i^t$	∞	5	6	5
$[x_2]_i^t$	5	6	5	∞
$[x_3]_i^t$	5	5	∞	5

case $\delta = 4$

input \ output	$[x_0]_i^{t+4}$	$[x_1]_i^{t+4}$	$[x_2]_i^{t+4}$	$[x_3]_i^{t+4}$
$[x_0]_i^t$	6	5	4	∞
$[x_1]_i^t$	6	6	∞	4
$[x_2]_i^t$	∞	5	6	5
$[x_3]_i^t$	4	∞	6	6

case $\delta = 5$

input \ output	$[x_0]_i^{t+5}$	$[x_1]_i^{t+5}$	$[x_2]_i^{t+5}$	$[x_3]_i^{t+5}$
$[x_0]_i^t$	7	4.6	5.4	8
$[x_1]_i^t$	5.6	7	6.8	5.4
$[x_2]_i^t$	8	6.5	7	4.6
$[x_3]_i^t$	5.4	8	6	7

The second, we observe a certain pair of distinct bit positions in the same column yields a bit-flip bias worse than any bias between the same bit positions, as seen in Table 2. These pairs with this property are like this:

The pair (x_0, x_1)**:** The bit-flip bias of $[x_0]_i^t = [x_1]_i^{t+1}$ is 2^{-4} and the bit-flip bias of $[x_1]_i^t = [x_0]_i^{t+1}$ is 2^{-3}.
The pair (x_2, x_3)**:** The bit-flip bias of $[x_2]_i^t = [x_3]_i^{t+1}$ is 2^{-4} and the bit-flip bias of $[x_3]_i^t = [x_2]_i^{t+1}$ is 2^{-3}.
The other pair: At least one case of the bit-flip bias is "0". For example, the bit-flip bias of $[x_0]_i^t = [x_3]_i^{t+1}$ is "0", the bit-flip bias of $[x_3]_i^t = [x_0]_i^{t+1}$ is 2^{-4}.

By using the property, we remove the nonlinear filter from relation of the pair (x_0, x_1), (x_2, x_3). The nonlinear filter of TSC-4 is carefully chosen such that its linear approximation contains the minimum number of pairs whose bit-flip bias is less than 2^{-5}.

Finally, we check the bit-flip bias between the linear relations of the same bits for multiple applications. Those linear relations are as follows:

1. $[x_0]_i^t \oplus [x_1]_i^t = [x_0]_i^{t+\delta} \oplus [x_1]_i^{t+\delta}$, $[x_2]_i^t \oplus [x_3]_i^t = [x_2]_i^{t+\delta} \oplus [x_3]_i^{t+\delta}$.
2. $[x_0]_i^t \oplus [x_2]_i^t = [x_0]_i^{t+\delta} \oplus [x_2]_i^{t+\delta}$, $[x_1]_i^t \oplus [x_3]_i^t = [x_1]_i^{t+\delta} \oplus [x_3]_i^{t+\delta}$.
3. $[x_0]_i^t \oplus [x_3]_i^t = [x_0]_i^{t+\delta} \oplus [x_3]_i^{t+\delta}$, $[x_1]_i^t \oplus [x_2]_i^t = [x_1]_i^{t+\delta} \oplus [x_2]_i^{t+\delta}$.

Since the first relation is removed in the nonlinear filter, we consider other two relations. Table 3 shows the bit-flip biases for each case.

Table 3. Bit-flip bias of $[x_k]_i^t \oplus [x_{k'}]_i^t = [x_k]_i^{t+\delta} \oplus [x_{k'}]_i^{t+\delta}$ ($|\, log_2 \varepsilon|$)

δ	$(k,k') = (0,2)$	$(k,k') = (1,3)$	$(k,k') = (0,3)$	$(k,k') = (1,2)$
1	**2.4150**	**2.4150**	2.4150	∞
2	∞	∞	∞	3.0000
3	∞	∞	∞	3.4150
4	∞	∞	∞	2.6781
5	3.7521	3.7521	3.7521	5.6781
6	3.4150	3.4150	**3.4150**	**2.1926**
7	4.9556	4.9556	4.9556	2.6163
8	4.3561	4.3561	4.3561	4.3561
9	8.5406	8.5406	8.5406	2.9860
10	9.4150	9.4150	9.4150	3.0170
11	4.8707	4.8707	4.8707	3.8401
12	7.2996	7.2996	7.2996	3.1703
13	3.7527	3.7527	3.7527	2.3618
14	5.3276	5.3276	5.3276	4.9125
15	5.7574	5.7574	5.7574	3.3714
16	8.3927	8.3927	8.3927	3.0438
⋮	⋮	⋮	⋮	⋮

Combining the two relation $(k, k') = (0, 2)$ and $(k, k') = (1, 3)$ in $\delta = 1$, we get the maximum bit-flip bias of this relation as $2^{-3.83}(= 2^{-1} \times (2^{-1.415})^2)$. Similarly, In case of $(k, k') = (0, 3)$ and $(k, k') = (1, 2)$, the maximum bit-flip bias is $2^{-4.6076}(= 2^{-1} \times 2^{-2.415} \times 2^{-1.1926})$ in $\delta = 6$. So, we use the relation of the pair $(x_0, x_3),\ (x_1, x_2)$ in the nonlinear filter.

Therefore, we can assume that the distinguishing attack is not applicable to the algorithm TSC-4.

3.7 Time-memory trade-off

We analyze the security of TSC-4 against time-memory-data(TMD) tradeoffs presented in [18, 19]. Then, it guarantees the security against two well-known TMD tradeoffs [15–17].

Simple case[18]: Since TSC-4 takes 80-bit key with 80-bit IV, Search space of an attacker is the entropy space of size $N = 2^k (k = 160)$. The cost of TMD attacks is $O(2^{k/2})$. So, TMD attacks are expected to have complexity not lower than $O(2^{80})$.

Sampling case[19]: Since TSC-4 takes 256-bit internal state and we can find the set of all 256-bit keystream segments which starts with 8 zeros, search space of an attacker is the entropy space of size $N = 2^k (k = 248)$. The cost of TMD attacks is $O(2^{k/2})$. So, TMD attacks are expected to have complexity not lower than $O(2^{124})$.

3.8 State initialization

We consider security issues related to key setup in this section. Our state retains 160-bit entropy after state initialization.

Entropy loss: Let us consider the question of whether our state initialization process allows every possible 160-bit state to occur with equal possibility. This question is closely related to whether each step of the rekeying process is invertible. Checking all the steps of Key/IV Loading and warm-up presented in Section 2.3, we can see that all step is invertible. So, the states produced through our state initialization process has exactly 160-bit entropy. Therefore no equivalent keys are present.

Statistical property: For a good state initialization process, we would expect one bit difference in key or IV to result in about half the state bits changing. We did some basic experiments to verify this on our warm-up process.

4 Implementation

4.1 Hardware Implementation

TSC-4 consists of two T-functions and a nonlinear filter. In hardware implementation, critical path is an even parameter of a T-function, and 4×4 s-boxes are components which requires large area. In updating internal states, s-box is applied to all 64 columns.

In normal hardware design, one implement 64 s-boxes to maximize the throughput. On the other hand, we can reduce the area by implementing one s-box for each T-function, or by implementing one T-function instead of two.

Let Type A, Type B, Type C denote normal implementation, implementation with one s-box for each T-function, implementation with one T-function and one s-box respectively.

In Table 4 we summarize hardware figures when the implementation was simulated on ASIC using Samsung 0.13μm library.

Table 4. Hardware related figures for TSC-4

Type	State Initialization	Gate Count	Max. Clock /Throughput	Throughput/Power drain (100KHz clock)
A	X	10510	100MHz/800Mbps	800kbps/11.86μW
A	O	11878	100MHz/800Mbps	800kbps/12.78μW
B	X	3100	250MHz/62.5Mbps	25kbps/4.65μW
B	O	4027	198MHz/49.5Mbps	25kbps/5.52μW
C	X	3026	230MHz/28.75Mbps	12.5kbps/4.51μW
C	O	3958	198MHz/24.75Mbps	12.5kbps/5.50μW

4.2 Software Implementation

Our C-language implementation (not optimized) of TSC-4 shows the following performance.

machine	Pentium-IV 2.4GHz, 1GB RAM
OS	Windows XP (SP1)
compiler	Microsoft Visual C++ 6.0
encryption	150 cycles/byte

5 Conclusion

A synchronous streamcipher TSC-4 of 80-bit intended security level was presented with some security analysis and hardware related figures. As a result,

we failed to find an attack which is better than exhaust key search. The cipher is suitable for constrained hardware environments, allowing for a wide range of implementation choices.

References

1. A. Klimov and A. Shamir, A new class of invertible mappings. *CHES 2002*, LNCS 2523, Springer-Verlag, pp.470–483, 2003.
2. A. Klimov and A. Shamir, Cryptographic application of T-functions. *SAC 2003*, LNCS 3006, Springer-Verlag, pp.248–261, 2004.
3. A. Klimov and A. Shamir, New cryptographic primitives based on multiword T-functions. *FSE 2004*, LNCS 3017, Springer-Verlag, pp.1–15, 2004.
4. J. Hong, D. H. Lee, Y. Yeom, and D. Han, A new class of single cycle T-functions and a stream cipher proposal. *SASC*(State of the Art of Stream Ciphers, Brugge, Belgium, Oct. 2004) workshop record.
5. J. Hong, D. H. Lee, Y. Yeom, and D. Han, New class of single cycle T-functions. *FSE 2005*, LNCS 3557, pp.68–82, Springer-Verlag, 2005.
6. J. Hong, D. H. Lee, Y. Yeom, D. Han, S. Chee, T-function based streamcipher TSC-3. *SKEW*(Symmetric Key Encryption Workshop), Available from `http://www.cosic.esat.kuleuven.ac.be/ecrypt/stream/`, 2005.
7. NIST. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST Special Publication 800-22.
8. F. Jonsson and T. Johansson, *A Fast Correlation Attack on LILI-128*, Information Processing Letters Vol 81, No. 3, 2001, pp.127-132.
9. W. Meier and O. Staffelbach, *Fast correlation attacks on certain stream ciphers*, J. Cryptology Vol 1, 1989, 159-176.
10. F. Armknecht and M. Krause, Algebraic attacks on combiners with memory, *Crypto 2003*, LNCS 2729, Springer-Verlag, pp.162–175, 2003.
11. N. Courtois and W. Meier, Algebraic attacks on stream ciphers with linear feedback, *Eurocrypt 2003*, LNCS 2656, Springer-Verlag, pp.345–359, 2003.
12. N. Courtois, Fast algebraic attack on stream ciphers with linear feedback, *Crypto 2003*, LNCS 2729, Springer-Verlag, pp. 176–194, 2003.
13. S. Künzli, P. Junod, and W. Meier, Distinguishing attacks on T-functions. *Mycrypt 2005,* LNCS 3715, pp. 2–15, Springer-Verlag, 2005.
14. F. Muller and T. Peyrin, Linear Cryptanalysis of TSC Stream Ciphers - Applications to the ECRYPT proposal TSC-3. *SKEW* Available from `http://www.cosic.esat.kuleuven.ac.be/ecrypt/stream/`, 2005.
15. S. H. Babbage, Improved exhaustive search attacks on stream ciphers. *European Convention on Security and Detection,* IEE Conference publication No. 408, pp. 161–166, IEE, 1995.
16. A. Biryukov and A. Shamir, Cryptanalytic time/memory/data tradeoffs for stream ciphers. *Asiacrypt 2000,* LNCS 1976, pp. 1–13, Springer-Verlag, 2000.
17. J. Dj. Golić, Cryptanalysis of alleged A5 stream cipher. *Eurocrypt'97,* LNCS 1233, pp. 239–255, Springer-Verlag, 1997.
18. J. Hong and P. Sarkar, New Applications of Time Memory Data Tradeoffs. *Asiacrypt 2005,* LNCS 3788, pp. 353–372, Springer-Verlag, 2005.
19. J. Hong and W. Kim, TMD-Tradeoff and State Entropy Loss Considerations of Streamcipher MICKEY. *Indocrypt 2005,* LNCS 3797, pp. 169–182, Springer-Verlag, 2005.

Update on F-FCSR Stream Cipher

F. Arnault*, T.P. Berger* and C. Lauradoux†

Abstract

The F-FCSR family of algorithms have been presented about one year ago with [2] and [1]. While some flaws where found in the initial propositions (on the IV-setup procedure, and a TMD tradeoff attack), there are yet no known weaknesses of the core of these algorithms.

We sum up here some of the properties of the automaton that are better understood now, and that have been presented in [2], [3], [4], and [6] and we propose two revised algorithms correcting all known weaknesses.

1 Recalls on F-FCSR

1.1 FCSR automaton

Detailed descriptions can be found in [3, 1, 2].

A Feedback with Carry Shift Register (FCSR) is an automaton which computes the binary expansion of a 2-adic number p/q, where p and q are some integers, with q is odd. We will assume that $q < 0 < p < |q|$. The size n of the FCSR is such that $n+1$ is the bitlength of $|q|$.

In our applications, p depends on the secret key (and the IV), and q is a public parameter. The choice of q induces many properties of the keystream. The most important one is that it completely determines the length of the period of the keystream. The conditions for an optimal choice are:

Conditions 1

- *q is a (negative) prime of bitsize $n+1$.*
- *The order of 2 modulo q is $|q|-1$.*
- *$T = (|q|-1)/2$ is also prime.*
- *Set $d = (1+|q|)/2$. The Hamming weight $W(d)$ of the binary expansion of d is not too small. Typically, $W(d) > n/2$.*

1.1.1 Software description of the transition function

The FCSR automaton contains two registers (sets of cells): the main register M and the carries register C.

The main register M contains n cells. We denote m_i $(0 \leq i \leq n-1)$ the binary digits contained in these cells and we call the integer $m = \sum_{i=0}^{n-1} m_i 2^i$ the content (or state) of M.

*XLIM, Université de Limoges, 123 avenue A. Thomas, 87060 Limoges CEDEX, France
Email : arnault@unilim.fr thierry.berger@unilim.fr

†INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
Email : cedric.lauradoux@inria.fr

Let d be the positive integer $d = (1-q)/2$ and $d = \sum_{i=0}^{n-1} d_i 2^i$ its binary expansion. The carries register contains ℓ cells where $\ell + 1$ is the number of nonzero d_i digits. More precisely, the carries register contains one cell for each nonzero d_i with $0 \leq i \leq n-2$. We denote c_i the binary digit contained in this cell. We also put $c_i = 0$ when $d_i = 0$ or when $i = n-1$. We call the integer $c = \sum_{i=0}^{n-2} c_i 2^i$ the content (or state) of C. The Hamming weight of the binary expansion of c is at most ℓ.

The transition function can be described by

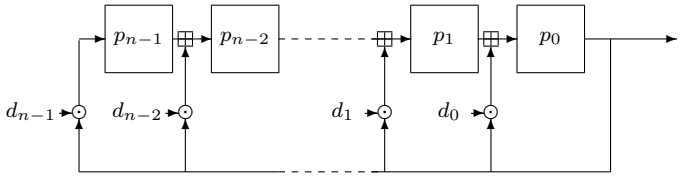
$$\begin{aligned} m(t+1) &:= (m(t) \div 2) \oplus c(t) \oplus m_0(t)d \\ c(t+1) &:= (m(t) \div 2) \otimes c(t) \oplus c(t) \otimes m_0(t)d \oplus m_0(t)d \otimes (m(t) \div 2) \end{aligned}$$

where $\oplus$ denotes bitwise XOR, $\otimes$ denotes bitwise AND, and $\div 2$ is a just a shift to the right.

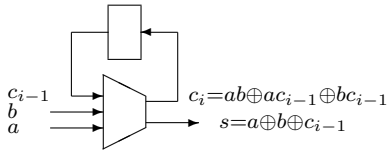
Note that $m_0(t)$ is the least significant bit of $m(t)$. The integers $m(t)$, $c(t)$ and d are integers of bitsize n (or less).

1.1.2 Hardware description of the transition function

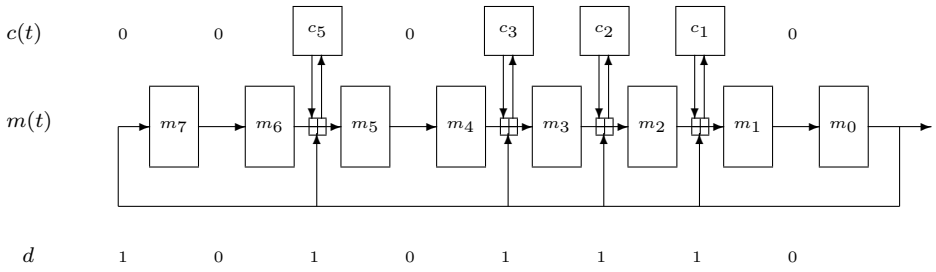
With the same notations, the hardware description of the FCSR generator is



where the symbol $\boxplus$ denotes the addition with carry, i.e., it corresponds to the following scheme:



As an example, if $q = -347$, so $d = 174 = 0$xAE, $n = 8$ and $\ell = 4$, we obtain the following diagram:



1.2 Filtering

We extract each pseudorandom bit from the state of the main register of the FCSR automaton using a filter. This filter describes which cells are selected to produce the pseudorandom bit. In order to obtain a multi-bit output, eight or sixteen one bit subfilters are used to extract an output 8 or 16 bits word after each transition of the automaton.

1.2.1 Principle of one bit filtering

The filter F is a bitstring $(f_0, \ldots, f_{n-1})$ of length n (or equivalently the integer $\sum_{i=0}^{n-1} f_i 2^i$). The output bit is obtained by computing the weight parity of the bitwise AND of the state M of the main register and of the filter F:

$$\text{Output bit} := \bigoplus_{i=0}^{n-1} f_i m_i.$$

Or, equivalently: $S = M \otimes F$ Output bit := parity(S)

1.2.2 Word filtering

In a similar way, we propose a method to extract an s bits word from the state of the FCSR. The value of s will be 8 for F-FCSR-H, and 16 for F-FCSR-16.

The filter F is also a bitstring $(f_0, \ldots, f_{n-1})$ of length n (which is a multiple of s). It splits into s subfilters $F_0, \ldots, F_{s-1}$ each defined by

$$F_j = \sum_{i=0}^{n/s-1} f_{si+j} 2^i.$$

Each subfilter F_j selects some cells m_i in the main register among the ones satisfying $i \equiv j$ modulo s. The parity of the binary word obtained gives one pseudorandom bit :

$$\text{bit } j \text{ of output word} := \bigoplus_{i=0}^{n/s-1} f_{si+j} m_{si+j}.$$

As there are s subfilters, we get s bits at each transition of the automaton.

This procedure can be described equivalently as follows. The filter F and the state of M are combined with the AND function. The result is split into n/s words. The pseudorandom word is obtained by XORing these n/s words:

$S := M \otimes F$
Define S_i by $S = \sum_{i=0}^{n/s-1} S_i \cdot 2^{si}$, with $0 \leq S_i \leq 2^s - 1$
Output word := $\bigoplus_{i=0}^{n/s-1} S_i$.

Note that it is faster to extract a whole word than a single bit.

2 Known issues on F-FCSR

2.1 Structure of the cycles of an FCSR automaton

Consider the transition function of an FCSR automaton. It is easy to see that it has two fixed points, namely the state with all cells containing a 0 bit, and the state with all cells containing

a 1 bit. The values of (m, c) for these states are $(0, 0)$ and $(2^n - 1, d - 2^n)$ respectively, and they correspond to the developpement of the 2-adic fractions $0/q = 0$ and $|q|/q = -1$. All other states are noninvariant by the transition function.

Since we assume that the order of 2 modulo q is $|q| - 1$, we can prove that the graph of the transition function consists of exactly three connected components: the two single point components corresponding to the two fixed points and another component containing all the $2^{n+l} - 2$ remaining points. Moreover, this component consists in a cycle of size $|q| - 1$ and pathes converging to it. More details on the transition function of FCSR automatons can be found in [3].

Definition 1 *Two states (m_1, c_1) and (m_2, c_2) are said equivalent if they satisfy $m_1 + 2c_1 = m_2 + 2c_2$.*

The following fundamental property can be shown:

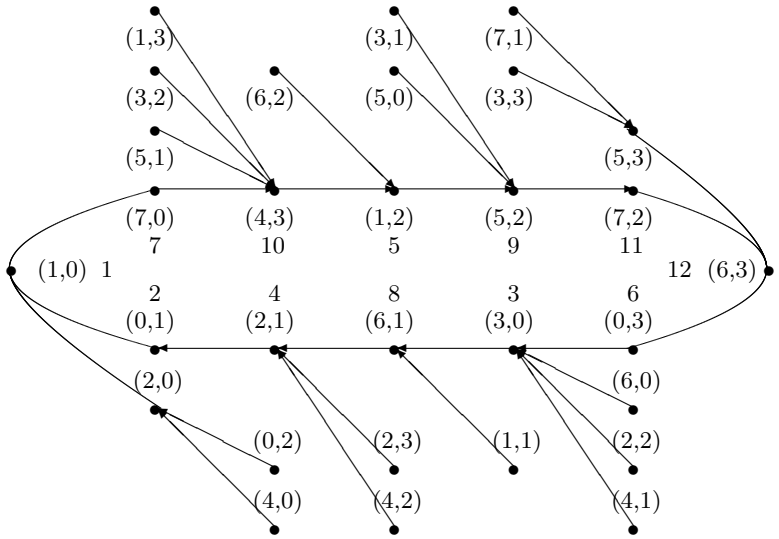
Proposition 1 *Two noninvariant states are equivalent if and only if they eventually converge to the same state of the main cycle in the same number of steps.*

As $|q| - 1 \simeq 2^n$, the expected number of states which eventually converge to a given state of the cycle is approximatively 2^l.

It can be shown also that the relative number of leaves for the transition function is $1-(3/4)^l$, which (for $l \geq 2$) is much larger than for a random function, where it is e^{-1}.

From the existence of a large cycle and of a large number of leaves, we can expect that the length of the pathes converging to the cycle are very short. Experimentally, this is indeed the case. Convergence occurs generally in less than $(n+l)/2$ iterations, while this should be $2^{(n+l)/2}$ if the transition function was a random one.

The following figure shows the main composant of the graph associated to $q = -13$. The couples of numbers correspond to a state (m, c) and the single numbers to the value $p = m + 2c$.



2.2 Consequences for Time/Memory/Data tradeoff attacks

First, the states that are not on the main cycle have only a very small impact on the cost of a Time/Memory/Data tradeoff attacks. So the number of states that should be considered when evaluating security of FCSRs with static filter should be $|q| - 1$ instead of 2^{n+l}.

Hence, in the first version of F-FCSR-8 submitted to Ecrypt, the size $k = 128$ of the key was equal to the length of the main register. But we also used a dynamic filter to increase the number of total states of our automaton.

However, using the fact that we used 8 subfilters of length 16 to output 8 bits at each transition, and using the fact that the number of possible such subfilters was too small, E. Jaulmes and F. Muller showed in [5] that the presence of a dynamic filter does not provide enough security. Their attack has a time cost in 2^{80} and uses data of size about 2^{67}.

The solution to prevent TMD-attacks is to increase the size of the prime q up to $n = 2 \times 128 = 256$. Note that in this case, it is possible to output two bytes instead of a single one at each iteration. Hence the number of operations per output byte is not increased and in fact the speed of the generator will be slightly better. Moreover, dynamic filter is no longer needed, and this greatly simplifies hardware and also software implementations.

There exist recent developpements on TMDtreadoffs cryptanalysis of stream cipher generators [8]. We want to notice that it possible to increase the size of IV of F-FCSR stream cipher until the size k of the key without any information. Moreover, in the procedure of change of IV, the key is concatened to the IV, which ensure a total entropy of our system equals to the size of the key plus the size of the IV.

2.3 Algebraic cryptanalysis

For the F-FCSR generator, the transition function of the automaton T_q is quadratic, and the filter F_l is linear.

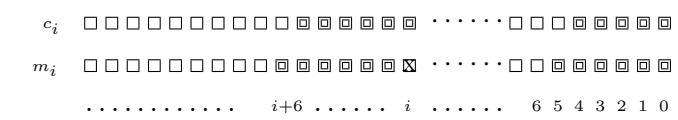
We denote by x the initial state of the generator: it is a binary vector of size equal to the number of the unknown values of the registers. The algebraic attack consists in the determination of x from the equations $F(T^i(x)) = s_i$, where the s_i are the successive observed bits output by the generator.

This leads to a system of equations $F_l(T_q^i(x)) = s_i$. The degree of the i-th equation is the degree of T_q^i. The first equation is linear, the second quadratic. An increase of the degree is expected at each iteration. However this increase depends on many factors as the choice of the filter or the values of I_d. It seems not possible to find a formula available in the general case.

However, In [4], M. Minier and T. Berger studied these equations in more details and designed an attack on an earlier version of F-FCSR proposed at FSE 05 [2]. In that situation, there was only 6 iterations after each change of IV.

The main result is the fact that, even if the degree of equations increases at each iteration, the number of monomials remains smaller than expected as long as the number of iterations is less than the size n of the register. The following Proposition describes this property:

Proposition 2 *The value of the content of the i-th register at the t-iteration $m_i(t)$ depends only on the initial values $(m_0(0), \cdots, m_{t-1}(0), c_0(0), \cdots, c_{t-2}(0)$ et $(m_{i+1}(0), \cdots, m_{i+t}(0), c_i(0)$ $, \cdots, c_{i+t-1}(0))$.*



An example for $t = 6$

In the attack described in [4], the IV values are known, that is the initial values $c_i(0)$ are given. The following table gives the number of distinct monomials obtained in the algebraic equations for a register of size 128.

nb of iterations	0	1	2	3	4	5	6
nb of monomials	128	129	256	758	2490	8830	32836
Algebraic degree	1	1	2	3	4	6	8
Binomial bound	129	129	8257	349633	11017633	$\approx 2^{32}$	$\approx 2^{40}$

There are some remarks about these results:

- From Proposition 2, the number of monomials is linear in the length n of the generator.
- From a computationnal point of view, the first difficult problem is not to solve the equations, but to compute them: we were not able to compute the equations corresponding to the 8-th iteration on a register of size 128. At the present moment, it seems not computationnaly feasible to complete the 12-th iteration.
- The F-FCSR-8 and F-FCSR-H stream ciphers proposed to Ecrypt are resistant to this kind of attacks.

2.4 Diffusion of differences

Another possible weakness of the first designs of F-FCSR stream ciphers resides in the slowness of diffusion of differences. A difference introduced in some cell of the FCSR automaton remains localized when clocking the automaton, as long as this difference does not reach the feedback end of the register. In fact, except when this end is reached, the difference only affects the next right cell after one transition and, with probability 1/2 only, the corresponding carry cell is also changed. This change in this carry cell, when it occurs, will cause subsequent differences at subsequent transitions. However, this change in the carry cell has low probability ($1/2^n$ after n transitions) not to disappear.

We illustrate this fact in the following example, where we choosed $q = -347$. The length n of the main register is then 8. We have chosen randomly a value m_1 for the main register m, strictly less than 2^7. For the initial values m_1 and $m_2 = m_1 + 2^7$, we computed the differences obtained in the main register after i iterations of the transition function, for $i = 0$ up to 9. The following table gives the typical results obtained this way, with two different values for m_1.

Position of carries								
1	0	1	0	1	1	1	0	
Diffusion of difference								
1	0	0	0	0	0	0	0	
0	1	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	
0	0	1	1	0	0	0	0	
0	0	1	1	1	0	0	0	
0	0	0	1	1	1	0	0	
0	0	0	0	1	1	1	0	
0	0	0	0	0	1	1	1	
1	0	1	0	1	0	1	1	
1	1	1	1	1	0	0	1	

Position of carries								
1	0	1	0	1	1	1	0	
Diffusion of difference								
1	0	0	0	0	0	0	0	
0	1	0	0	0	0	0	0	
0	0	1	0	0	0	0	0	
0	0	1	1	0	0	0	0	
0	0	0	1	1	0	0	0	
0	0	0	0	0	1	0	0	
0	0	0	0	0	1	1	0	
0	0	0	0	0	1	0	1	
1	0	1	0	1	1	1	0	
0	1	0	1	0	1	0	1	

This fact was noticed by E. Jaulmes and F. Muller (cf. [6, 5]) and used to design attacks on the change of IV procedure. They obtained a key recovery attack on F-FCSR-8 and a distinguishing attack on F-FCSR-H.

There are two independant ways in order to stop these attacks:

1. A better insertion of IV and key in the change of IV procedure. In our first version, we used a simple concatenation of the values.
2. A larger number of iterations of the transition function before outputing data, greater than the length n of the register, to ensure a full diffusion of the differences

As an example, in our first version of F-FCSR H , it is sufficient to increase the number of iterations from 160 to 162 in order to stop the distinguishing attack.

2.5 Other attacks

At the present moment, we do not known any other attack against this design, in particular any correlation attack.

3 New Design

3.1 F-FCSR-H: Profile 2, output 1 byte per round

This proposal uses keys of length 80 and IV of bitsize v with $32 \leq v \leq 80$. An IV of value 0 can be used as a default if no value is provided. The core of this new version of the F-FCSR-H algorithm is identical to the one proposed in [1]. Only the key+IV Setup procedure has been updated in view of the attacks presented in [6].

The FCSR length (size of the main register) is $n = 160$. The carries register contains $\ell = 82$ cells. The retroaction prime is

$$q = -199352459131827501532804161134421503646014008796 3$$

so addition boxes and carries cells are present at the positions matching the ones (except of the leading one) in the following 160 bits string (which has Hamming weight 83)

$$d = (1 + |q|)/2 = (\text{AE985DFF 26619FC5 8623DC8A AF46D590 3DD4254E})_{16}.$$

Filtering

To extract one pseudorandom byte, we use the static filter

$$F = d = (\text{AE985DFF 26619FC5 8623DC8A AF46D590 3DD4254E})_{16}$$

The filter F splits in 8 subfilters (subfilter j is obtained by selecting the bit j in each byte of F)

$$\begin{aligned} F_0 &= (0011\ 0111\ 0100\ 1010\ 1010)_2, & F_4 &= (0111\ 0010\ 0010\ 0011\ 1100)_2, \\ F_1 &= (1001\ 1010\ 1101\ 1100\ 0001)_2, & F_5 &= (1001\ 1100\ 0100\ 1000\ 1010)_2, \\ F_2 &= (1011\ 1011\ 1010\ 1110\ 1111)_2, & F_6 &= (0011\ 0101\ 0010\ 0110\ 0101)_2, \\ F_3 &= (1111\ 0010\ 0011\ 1000\ 1001)_2, & F_7 &= (1101\ 0011\ 1011\ 1011\ 0100)_2. \end{aligned}$$

Recall that the bit b_i (with $0 \leq i \leq 7$) of each extracted byte is expressed by

$$b_i = \bigoplus_{j=0}^{19} f_i^{(j)} m_{8j+i} \qquad \text{where } F_i = \sum_{j=0}^{19} f_i^{(j)} 2^j$$

and where the m_k are the bits contained in the main register.

a+b. Key+IV setup (Inputs a key K of length $k = 80$ and an IV of length $v \leq 80$)

1. The main register M is initialized with the key and the IV:

$$M := K + 2^{80} \cdot IV = (0^{80-v} \| \text{IV} \| K).$$

2. The carries register is initialized to 0 :

$$C := 0 = (0^{82}).$$

3. A loop is iterated 20 times. Each iteration of this loop consists in clocking the FCSR and then extracting a pseudorandom byte S_i ($0 \leq i \leq 19$) using the filter.

4. The main register M is reinitialized with these bytes:

$$M := \sum_{i=0}^{19} S_i = (S_{19} \| \cdots \| S_1 \| S_0).$$

5. The FCSR is clocked 162 times (output is discarded in this step).

c. Extraction of pseudorandom data After setup phase, the pseudorandom stream is produced by repeating the following process as many times as needed

- Clock the FCSR
- Extract one pseudorandom byte using filter F as described above.

3.2 Upgrade from F-FCSR-8 to F-FCSR-16

In the F-FCSR-8 algorithm presented in [1], the pseudorandom stream was extracted using a dynamic filter. The purpose of this filter was to enlarge the number of states of the FCSR-automaton, in order to prevent Time-Memory-Data tradeoff attacks. However, the paper [6] shows that such a dynamic filter does not provide the expected security. In the light of this result, the new algorithm F-FCSR-16 uses a static filter and the required number of states of the automation is obtained by enlarging the size of the registers. Note that the larger size of the register allows to extract more pseudorandom bits at each transition of the automaton. So the new algorithm is as fast as the previous one.

3.2.1 F-FCSR-16: Profile 1, output 2 bytes per round

This proposal uses keys of length $k = 128$ and an IV of length $v = 128$ or 64 (any length $v \leq 128$ can be used). An IV of value 0 can be used as a default if no value is provided by the application.

According to Conditions 1 we choose for q the following number

$$-q = 183971440845619471129869161809344131658298317655923135753017128462155618715019$$

as the public parameter of the automaton. The corresponding bitstring $d = (|q| + 1)/2$ which describes the positions of the carries cells is

$$d = (\text{CB5E129F AD4F7E66 780CAA2E C8C9CEDB 2102F996 BAF08F39 EFB55A6E 390002C6})_{16}.$$

Its Hamming weight is 131 and there are $\ell = 130$ cells (the Hamming weight of $d^* = d - 2^{255}$) in the carries register and $n = 256$ cells in the main register.

To extract two pseudorandom bytes, we use the static filter

$$F = d$$

The filter F splits in 16 subfilters (subfilter j is obtained by selecting the bit j in each 16-bit word of F)

$$\begin{array}{ll} F_0 = (0110\ 0011\ 0001\ 1000)_2, & F_8 = (1010\ 0000\ 1101\ 1010)_2, \\ F_1 = (1111\ 0101\ 1100\ 0101)_2, & F_9 = (1101\ 0101\ 0011\ 1101)_2, \\ F_2 = (1111\ 1100\ 0100\ 1101)_2, & F_{10} = (0011\ 0001\ 0001\ 1000)_2, \\ F_3 = (1110\ 1111\ 0001\ 0100)_2, & F_{11} = (1011\ 1111\ 0111\ 1110)_2, \\ F_4 = (1100\ 0001\ 0111\ 1000)_2, & F_{12} = (0101\ 1000\ 0110\ 0110)_2, \\ F_5 = (0001\ 0100\ 0011\ 1100)_2, & F_{13} = (0011\ 1100\ 1110\ 1010)_2, \\ F_6 = (1011\ 0011\ 0010\ 0101)_2, & F_{14} = (1001\ 1011\ 0100\ 1100)_2, \\ F_7 = (0100\ 0011\ 0110\ 1001)_2. & F_{15} = (1010\ 0111\ 0111\ 1000)_2. \end{array}$$

Recall that the bit b_i (with $0 \le i \le 15$) of each extracted word is expressed by

$$b_i = \bigoplus_{j=0}^{15} f_i^{(j)} m_{16j+i} \qquad \text{where } F_i = \textstyle\sum_{j=0}^{15} f_i^{(j)} 2^j$$

and where the m_k are the bits contained in the main register.

a+b. Change of IV (Input: an IV of bitsize $v \le 128$)

$M := K + 2^{128} \cdot \text{IV} = (0^{128-v} \| IV \| K)$
$C := 0 = (0^{130})$ (Clear the carries)
For i from 0 to 15 Repeat
    Clock the FCSR automaton
    Extract a pseudorandom word S_i using the filter F
End For
$M := \sum_{i=0}^{15} S_i \cdot 256^i = (S_{15} \| \cdots \| S_0)$
$C := 0 = (0^{130})$ (Clear the carries)
Clock the FCSR automaton 258 times (discard output in this step)

c. Extraction of the pseudorandom stream We use the word filtering method described above, with $s = 16$, while pseudorandom data is needed. At each clock of the FCSR automaton, the content of the main register M is ANDed with the filter F:
$S = M \otimes F$
S is split in 16 words each of bitlength 16 $S = \sum_{i=0}^{15} S_i 2^{16i}$
The pseudorandom byte is the XOR of these bytes: Output word $:= \bigoplus_{i=0}^{15} S_i$

3.2.2 F-FCSR-16, Profile 2

The F-FCSR-16 algorithm can also satisfy profile 2. As in this case the key-length is 80, the first line of the Change of IV procedure now reads

$$M := K + 2^{128} \cdot \mathrm{IV} = (0^{128-v} \| IV \| 0^{48} \| K)$$

Comparing to F-FCSR-H, the pseudorandom word extracted at each transition of the automaton is twice larger, while the size of the registers is only 8/5 larger. In applications with profile 2 where extremely high speed of pseudorandom data generation is needed, the F-FCSR-16 algorithm should be also considered.

4 Performances

The software performance of F-FSCR-16 stream cipher depends on the processor register width. For instance, we observe a speedup by four with 128-bits ALTIVEC implementation over 32-bits implementation. This observation was already performed in [9] with F-FCSR-8. The main mechanisn of F-FCSR-H remains unchanged and results on its implementation can be found in [9].

CISC target	parameters		performance		
	Frequency	L2 Cache Size	Speed	Code	Initialization
Pentium 3	800 Mhz	256KB	83 cycles/B	8 KB	39140 cycles/IV
Pentium 4	2.3 Ghz	512KB	85 cycles/B	8 KB	54491 cycles/IV
Pentium 4	2.6 Ghz	512KB	95 cycles/B	8 KB	38351 cycles/IV
Pentium 4	3.2 Ghz	1MB	82 cycles/B	6 KB	43354 cycles/IV

RISC target	parameters		performance		
	Frequency	L2 Cache Size	Speed	Code	Initialization
PPC 7457	1.2 Ghz	512 KB	90 cycles/B	18 KB	44860 cycles/IV
PPC 7457 (ALTIVEC)	1.2 Ghz	512 KB	22 cycles/B	14 KB	11828 cycles/IV

Figure 1: F-FCSR-16 32-bit evaluation and ALTIVEC implementation

References

[1] F. Arnault and T.P. Berger. Design of new pseudorandom generators based on a filtered FCSR automaton. In *SASC*, State of the Art of Stream Ciphers Workshop, pages 109–120, Bruges, Belgium, October 2004.

[2] F. Arnault and T.P. Berger. F-FCSR: design of a new class of stream ciphers. In H. Handschuh H. Gilbert, editor, *Fast Softward Encryption 2005*, number 3557 in Lecture Notes in Computer Science, pages 83–87. Springer, 2005.

[3] F. Arnault and T.P. Berger. Design and properties of a new pseudorandom generator based on a filtered FCSR automaton. *IEEE, Transactions on Computers. IEEE, Transactions on Computers*, 54(11):1374–1383, November 2005.

[4] T.P. Berger and M. Minier. Two algebraic attacks against the F-FCSRs using the IV mode. in S. Maitra, C.E. Veni Madhavan, R. Venkatesan editors, *Progress in Cryptology - INDOCRYPT 2005* number 3797 in Lecture Notes in Computer Science, pages 143–154. Springer, 2005.

[5] E. Jaulmes and F. Muller. Cryptanalysis of Ecrypt candidates F-FCSR-8 and F-FCSR-H. ECRYPT Stream Cipher Project Report 2005/046, 2005. `http://www.ecrypt.eu.org/stream`.

[6] E. Jaulmes and F. Muller. Cryptanalysis of the F-FSCR stream cipher family. In *proceedings of 12th annual workshop on Selected Areas in Cryptography*, LNCS, Springer-Verlag, 2005.

[7] F. Arnault, T.P. Berger and C. Lauradoux. Preventing weaknesses on F-FCSR in IV mode and tradeoff attack on F-FCSR-8. ECRYPT Stream Cipher Project Report 2005/075, 2005. `http://www.ecrypt.eu.org/stream`.

[8] J. Hong and P. Sarkar. New Applications of Time Memory Data Tradeoffs. in B. Roy editor, *Advances in Cryptology - ASIACRYPT 2005* number 3788 in Lecture Notes in Computer Science, pages 353–372. Springer, 2005.

[9] F. Arnault, T.P. Berger and C. Lauradoux. F-FCSR. ECRYPT Stream Cipher Project Report 2005/008, 2005. `http://www.ecrypt.eu.org/stream`.

Security and Implementation Properties of ABC v.2

Vladimir Anashin[1], Andrey Bogdanov[2], and Ilya Kizhvatov[1]

[1] Russian State University for the Humanities,
Institute for Information Sciences and Security Technologies,
Faculty of Information Security,
Kirovogradskaya Str. 25/2, 117534 Moscow, Russia
{anashin,kizhvatov}@rsuh.ru
[2] escrypt GmbH – Embedded Security
Lise-Meitner-Allee 4, D-44801 Bochum, Germany
abogdanov@escrypt.com

Abstract. ABC is a synchronous stream cipher submitted to eSTREAM. Here we describe ABC v.2 – a tweaked version of ABC. The tweaks made ABC v.2 resistant to certain attacks, including the ones presented by Berbain and Gilbert and by Khazaei. We give a design rationale and a brief security analysis of ABC v.2. Also it is shown that the distinguishing attacks against ABC v.2 like the one suggested by Khazaei and Kiaei are totally impractical. ABC v.2 is extremely fast in software often heading the eSTREAM benchmark list. Further we define informal requirements for an industrial software stream cipher and show that ABC v.2 meets them. Moreover, we demonstrate that ABC v.2 is also suitable for embedded security applications demanding high performance.

Keywords: cryptography, stream cipher, ABC, eSTREAM, ECRYPT, distinguishing attack, stream cipher performance

1 Introduction

ABC is a synchronous stream cipher optimized for software applications which was submitted to eSTREAM [7]. ABC v.2 [8] with a 128-bit key and 32-bit internal variables, offers 128-bit security and is extremely fast in software often heading the eSTREAM performance benchmark list and ranking first in packet encryption [2].

This paper first outlines the tweaks to the original ABC that lead to ABC v.2. Then the attacks and the way the tweaks make ABC v.2 resistant to these attacks are described. Another possible tweak is discussed. We also show that 'Theorem 1' from the paper [12] by S. Khazaei describing an attack on ABC is wrong.

It is shown that the paper [13] by S. Khazaei and M. Kiaei does not present any distinguishing attack both on ABC v.1 and ABC v.2. The results of experiments are presented, indicating that the distinguisher for ABC v.2 has a complexity greater than that of a brute force attack.

Apart from its security properties, ABC v.2 meets a set of requirements which distinguish a stream cipher well suited for the real-world applications according to a number of features. We call these *industrial software implementation requirements* which are the following:

- High generic performance for all software platforms including embedded ones (at least twice as fast as AES on the same platform),
- Low memory consumption,
- Low costs of IV and key setup procedures.

Since these properties are mutually contradictory (e.g. more precomputations allow as a rule a faster implementation which leads, however, to a higher memory consumption), the latter two of them can be substituted for *flexibility* which means that a good industrial cipher should be capable of an efficient throughput/memory trade-off. ABC v.2 meets these requirements which is shown in the paper.

Actually ABC is a family of stream ciphers. This implies not only the flexibility of ABC implementation, but also the natural flexibility of the ABC design, which enabled us in [6] to suggest the tweaks raising its keystream period from $2^{32} \cdot (2^{63} - 1)$ 32-bit words to $2^{32} \cdot (2^{127} - 1)$ 32-bit words while keeping all the other properties of ABC stated in [7], including guaranteed uniform distribution and high linear complexity of the keystream.

Moreover, the ABC stream cipher is highly scalable which gives a possibility of natural extension of the cipher to a larger computational base (e.g. 64-bit version of ABC) and to exchange its separate components with very low overhead. This was done in ABC v.2 and can be further extended to create a version of ABC providing 256-bit security with a negligible performance overhead.

The paper is organized as follows. In Section 2 ABC v.2 is introduced and its differences from ABC v.1 are discussed. Section 3 describes a class of distinguishing and correlation attacks which could be applicable to ABC v.1 and ABC v.2. In Section 4 a number of ways avoiding this attack possibilities are suggested and the remedy selection for ABC v.2 is motivated. Section 5 provides experimental evidence demonstrating that ABC v.2 is robust to the distinguishing attack. In Section 6 we consider the industrial software implementation requirements, show that ABC v.2 meets them, discuss in what way ABC v.2 is superior to the other eSTREAM ciphers and demonstrate that ABC v.2 clearly outperforms AES on embedded platforms. We conclude in Section 7.

2 Moving from ABC v.1 to ABC v.2

Here the tweaks in the ABC keystream generator making ABC v.2 out of ABC v.1 are briefly outlined. The adjusted setup procedures described in [6,8] are not discussed here, we just note that some inaccuracy concerning the initialization routine mentioned in [9] was corrected. The following notation is used in the description of the cipher.

$x, y \in \mathbb{Z}/2^{32}\mathbb{Z}$ denote the state of the function B and the output of the keystream generator respectively;

z is a 128-bit integer value for ABC v.2 and a 64-bit integer value for ABC v.1 denoting the state of the transform A; it can also be represented as $z = (\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4$ for ABC v.2 and $z = (\bar{z}_1, \bar{z}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^2$ for ABC v.1, $\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0 \in \mathbb{Z}/2^{32}\mathbb{Z}$;

$d_0, d_1, d_2, e, e_0, e_1, \ldots, e_{31} \in \mathbb{Z}/2^{32}\mathbb{Z}$ denote the coefficients of the transforms B and C respectively;

$w \in \mathbb{Z}/2^5\mathbb{Z}$ denotes the length in bits of the optimization window used in computation of the transform C;

$_i(\cdot)$ is the i-th bit selection operator returning the value of the i-th bit of an integer, e.g. $_0(x)$ is the least significant bit of x;

is the bitwise modulo 2 addition ('XOR') operation;

, , $\ggg$ denote correspondingly left (zero-fill) bit shift, right (zero-fill) bit shift and right rotation of binary expansion of a 32-bit integer.

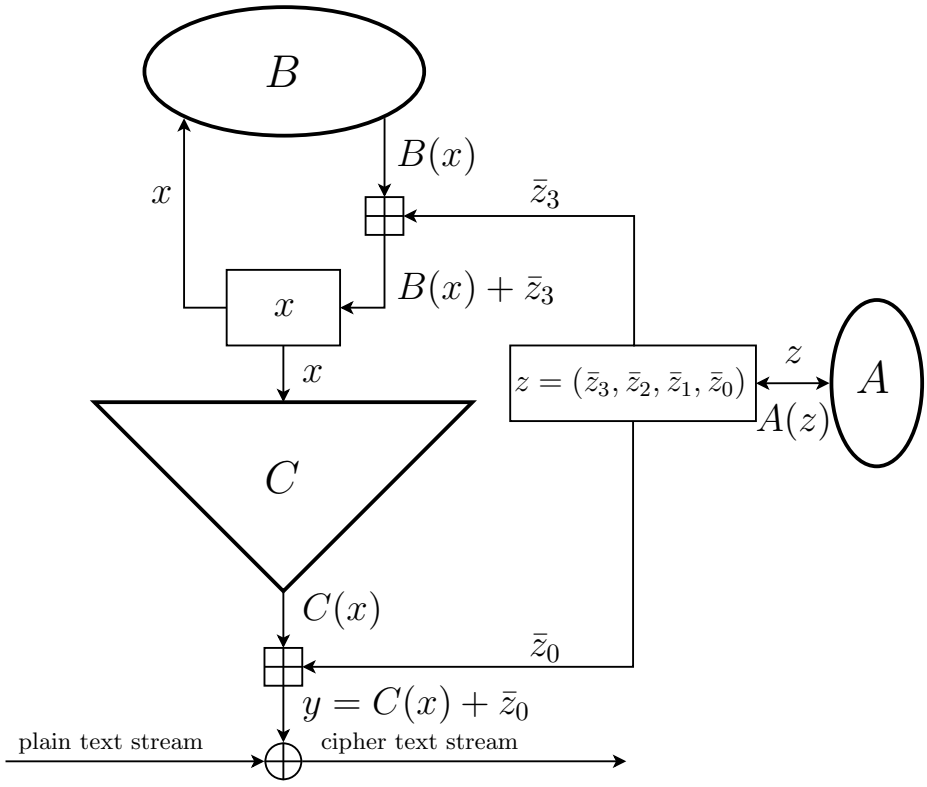


Fig. 1. ABC v.2 keystream generator

The keystream generator of ABC v.2 is illustrated in Fig. 1. In both versions of ABC A is a linear transformation of the vector space $\mathbb{V}_n = \mathrm{GF}(2)^n$ with a cycle of length $2^{n-1} - 1$ (where $n = 128$ for ABC v.2, and $n = 64$ for ABC v.1), B is a single cycle T-function on 32-bit words, and $C\colon \mathbb{Z}/2^{32}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}$ is a filter function: C takes x as argument and produces y in the following way:

$$\begin{aligned} \zeta &= S(x), \\ y &= \zeta \ggg 16, \end{aligned} \tag{1}$$

where $\zeta \in \mathbb{Z}/2^{32}\mathbb{Z}$ and $S : \mathbb{Z}/2^{32}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}$ is a mapping defined by

$$S(x) = e + \sum_{i=0}^{31} e_{i\ \ i}(x) \bmod 2^{32}, \tag{2}$$

$e_{31} \equiv 2^{16} \pmod{2^{17}}$. Coefficients $e, e_0, \ldots, e_{31} \in \mathbb{Z}/2^{32}\mathbb{Z}$ are obtained from the key during the initialization procedure.

The single cycle function B used in the ABC v.2 cipher can be specified through the following equation:

$$B(x) = ((x \quad d_0) + d_1) \quad d_2 \bmod 2^{32}, \tag{3}$$

where $d_0 \equiv 0 \pmod 4$, $d_1 \equiv 1 \pmod 4$, $d_2 \equiv 0 \pmod 4$. In the non-modified ABC v.1 the function B was of the form

$$B(x) = d_0 + 5(x \qquad d_1) \bmod 2^{32}, \tag{4}$$

with $d_0 \equiv 1 \pmod 2$, $d_1 \equiv 0 \pmod 4$.

Under the restrictions mentioned above the following properties of the keystream produced by the ABC v.2 keystream generator are proved:

- The length P of the shortest period of the keystream sequence of 32-bit words is $P = 2^{32} \cdot (2^{127} - 1)$.
- The distribution of the keystream sequence of 32-bit words is uniform in the following sense: For each 32-bit word a the number (a) of occurrences of a at the period of the keystream satisfies the following inequality:

$$\left| \frac{(a)}{P} - \frac{1}{2^{32}} \right| < \frac{1}{\sqrt{P}}.$$

- The linear complexity λ of the keystream bit sequence satisfies the inequality $2^{31} \cdot (2^{127} - 1) + 1 \geq \lambda \geq 2^{31} + 1$.

Proofs are based on the results presented in [4] and can be found in the updated ABC specification [8].

3 Attack Possibilities

In this section we describe some attacks that lead to recovering the internal state of the (non-modified) ABC v.1, and which are more efficient than a brute force attack. The corresponding remedies are discussed in Section 4.

Suppose that one has a statistical test $\mathcal{T}$ (which is further called a *distinguisher*) that could tell the keystream sequence $Y = \{y_j \in \mathbb{Z}/2^{32}\mathbb{Z}\}_{j=0}^{\infty}$ from the intermediate sequence $C(X) = \{C(x_j) \in \mathbb{Z}/2^{32}\mathbb{Z}\}_{j=0}^{\infty}$, which is the output of the function C. Then trying different initial states $\hat{z}$ of the LFSR A and testing the sequences $\overline{C(X)}(z) = \{y_j - \bar{z}_{0,j}(\hat{z}) \bmod 2^{32}\}$ with $\mathcal{T}$, where $\bar{z}_{0,j}(\hat{z})$ is the the 32 low order bits of the output of the LFSR A at the j-th step, one finds $\bar{z}$.

In other words, if the guess for LFSR state is correct, subtracting the LFSR sequence from the keystream sequence results in bare C output. If the guess for LFSR state is incorrect, the subtracting leads to some other sequence $\overline{C}$. Now, if we distinguish C from $\overline{C}$, we determine the correct guess. Actually, the awaited statistical properties of $\overline{C}$ are as good as those of the keystream sequence Y. So from the point of view of simplest and effective distinguishers $\overline{C}$ and Y are the same. That is why C can be distinguished from $\overline{C}$ by such a distinguisher that can tell C from Y.

Under the assumption that $\mathcal{T}$ makes no errors in distinguishing, the computational cost of finding the true initial state of the LFSR is $(2^n - 1)T$ computations of AB, where T is the computational cost of testing one sequence with the test $\mathcal{T}$, and n is the length of the LFSR registry (i.e., $n = 63$ in non-modified ABC, and $n = 127$ in the modified one). After finding the true initial state $\hat{z}$ of the LFSR, one tests coefficients of the function B and then, solving the corresponding congruences modulo 2^{32} with respect to the unknown values of $e, e_0, \ldots, e_{31}$, totally recovers the internal state of the ABC.

Attacks of this kind were mounted by Berbain and Gilbert in [9], and by Shahram Khazaei in [12]. They were successfully thwarted (actually prior to their publishing) by the ABC v.2 update, containing the remedies described in the next section.

4 Remedies

We need only those remedies that do not worsen the important properties of ABC (long period, uniform distribution and high linear complexity of the keystream) and/or significantly reduce its performance. There are several such remedies; two of them are described below.

4.1 Remedy 1: Special Coefficients

Since the coefficients $e, e_0, \ldots, e_{31}$ of the function S of (2) are produced in a pseudorandom way during the initialization stage, the probability the mapping C of (1) is bijective is too small; see Corollary 1 below for the exact value of that probability (the estimate of [9] is just an empirical conjecture and the one of [12]

is based on the erroneous 'Theorem 1' of [12]). Hence, with high probability the distribution of the sequence $C(X) = \{C(x_j) \in \mathbb{Z}/2^{32}\mathbb{Z}\}_{j=0}^{\infty}$, is not uniform since the distribution of the sequence $X = \{x_j \in \mathbb{Z}/2^{32}\mathbb{Z}\}_{j=0}^{\infty}$, which is the output if the function B, is uniform. This follows from the results stated [4] and can be found in [8]. At the same time, the distribution of the keystream sequence $Y = \{y_j \in \mathbb{Z}/2^{32}\mathbb{Z}\}_{j=0}^{\infty}$ is uniform (see Section 2). Hence, the distribution of the sequence $\overline{C(X)}(\hat{z})$ is not uniform in case of the right guess of the initial state $\hat{z}$ of the LFSR A, since the distribution of the output sequence of the LFSR A is uniform.

Thus, a distinguisher $\mathcal{T}$ just tests the uniformity of distribution of the sequence $\overline{C(X)}(z)$ for various z; in case the distribution is not uniform, the corresponding $z = \hat{z}$ is accepted as a true one. Distinguishers of [9] and of [12] are exactly of this sort.

To make sequences $\overline{C(X)}(\hat{z})$ indistinguishable one from another with respect to the test $\mathcal{T}$ for all the choices of $\hat{z}$ it suffices to choose coefficients of S in some special way to ensure that S is bijective.

Thus one needs criteria the coefficients should satisfy to make S bijective. In [12, Theorem 1] the following 'criterion' is stated: The function

$$\begin{array}{l} S(x) = e + \sum_{i=0}^{k-1} e_i \ {}_i(x) \pmod{2^k}, \\ x, e, e_i \in \mathbb{Z}/2^k\mathbb{Z}, i = 0, \ldots, k-1, \end{array} \tag{5}$$

induces a permutation of the residue ring $\mathbb{Z}/2^k\mathbb{Z}$ iff for each non-empty subset $M \subset \{0, 1, \ldots, k-1\}$

$$\sum_{i \in M} e_i \not\equiv 0 \pmod{2^k}.$$

However, it could be immediately shown that the above 'criterion' (as well as the whole 'Theorem' 1 of [12]) *are merely wrong*: Take $k = 3$, put $e_0 = 1$, $e_1 = 2$, $e_2 = 3$ and verify that the mapping $x \mapsto {}_0(x) + 2 \cdot {}_1(x) + 3 \cdot {}_2(x)$ is *not* a permutation of the residue ring modulo 8.

The right criterion reads the following.

Theorem 1. *The function* (5) *induces a permutation on the ring* $\mathbb{Z}/2^k\mathbb{Z}$ *if and only if*

$$e_{j_0} \equiv 1 \pmod 2, \quad e_{j_1} \equiv 2 \pmod 4, \ldots, e_{j_{k-1}} \equiv 2^{k-1} \pmod{2^k},$$

for some permutation $(j_0, j_1, \ldots, j_{k-1})$ *of* $(0, 1, \ldots, k-1)$.

Corollary 1. *There are exactly* $k! \cdot 2^{\frac{k(k+1)}{2}}$ *permutations among all* $2^{k(k+1)}$ *pairwise distinct transformations of the form* (5) *of the residue ring* $\mathbb{Z}/2^k\mathbb{Z}$. *Hence, the probability that* S *is a permutation is* $k! \cdot 2^{-\frac{k(k+1)}{2}}$.

In other words, S of (2) is a permutation iff $e_0, \ldots, e_{31}$ could be reordered so that $e_i = 2^i \cdot e_i'$, where e_i' are odd, $i = 0, 1, \ldots, 31$. Note that our condition $e_{31} \equiv 2^{16} \pmod{2^{17}}$ is in a certain sense a 'remnant' of our Theorem 1.

Theorem 1 follows immediately from a (more than 10 year old) result of one of us, see [3, Proposition 4.8]. Also, it could be easily deduced from the older result of DeBruijn, see [15, Section 4.1, Exercise 30]. Of course, it is not difficult to prove this theorem directly.

Thus, just to avoid the kind of attack described in [9] and [12] it is sufficient *only to make minor modifications to the initialization procedure* so that one of $e_0, \ldots, e_{31}$ always has 1 in the least significant bit position, another has 01 in its two rightmost bit positions, a further one has 001 in the three rightmost bit positions, etc. *The modification does not change the ABC keystream generation routine at all*, leaving both the performance and other properties (period length, uniform distribution, linear complexity) unchanged.

So the assumption of [12] by S. Khazaei that 'The designers of ABC have not neither evaluated C function theoretically nor using statistical simulations and just have designed C function to provide a provably minimum period for its output sequences' is just not true. We certainly could make S (whence, C) balanced (that is, bijective) at the very first stage of the ABC design procedure: We had mathematical tools to construct balanced mappings. These tools have been developed long before (see e.g. the bibliography in [7] and [8]) and are more effective than the ones of paper [14]. However, the *arbitrary* choice of coefficients in accordance with our Theorem 1 *might lead to some attacks* unless some special countermeasures are undertaken.

4.2 Remedy 2: Long LFSR

This solution is based on the usage of LFSR with period $2^{127} - 1$ instead of the LFSR with period $2^{63} - 1$ in the keystream generator, see Fig. 1. In spite of the fact that it implies modification of the keystream routine (we had also to modify the B function to compensate some speed reduction), the solution makes the ABC resistant to *all possible* attacks of the described kind *independently* of concrete distinguishers $\mathcal{T}$ they are based on: The computational cost is then $(2^{127} - 1) \cdot T \approx 2^{127} \cdot T \geq 2^{128}$, since we could hardly imagine a distinguisher with computational cost $T = 1$, under every reasonable definition of what the computational cost is. Thus, every attack of the described type becomes less effective than a brute force attack. As a bonus we obtain certain increase of security of the function B, since some extra bits of security are added (cf. (3) and (4)).

4.3 Scalability of ABC Design

The architecture of ABC stream cipher is highly scalable. This provides one with a possibility of natural extension of the cipher.

First, the extension can aim at a larger computational base, e.g. 64-bit version of ABC for 64-bit platforms, such as Intel Itanium or PowerPC G5. Second, the separate components of ABC, namely, A, B and C transforms, can be exchanged with a very low overhead.

Moreover, a natural extension of the digit capacity of the components of ABC can lead to a more secure and efficient cipher. This was done in ABC v.2 and can be further extended to create a version of ABC providing 256-bit security with a negligible performance overhead. Such a version of ABC with a 256-bit word-oriented LFSR, 256-bit key and 256-bit IV encrypts at 4.21 processor cycles per byte (the measurements were performed on a 1.73 GHz Intel Pentium M processor using the eSTREAM testing framework), which is only 4 percent more than for ABC v.2 with a 128-bit LFSR.

5 The Impracticability of Some Distinguishing Attacks

In their paper [13] Shahram Khazaei and Mohammad Kiaei claimed that there is a distinguisher on both versions of ABC with the complexity of about 2^{32}. The claim was supported by the empirical results of computer experiments with a set of reduced versions of ABC. However, the authors of [13] have made multiple errors, see [5] for details.

The idea of the possible attack is to reduce the influence of the LFSR A on the keystream and then detect a bias originating from the non-balanced filter function C. One can imagine that the LFSR influence can be reduced by applying an annihilator of the LFSR sequence to the keystream sequence. The word-oriented recurrence relation of A for ABC v.2 [8] induces the word-oriented annihilator

$$\bar{z}_{0,i} \quad \bar{z}_{0,i-2} \quad (\bar{z}_{0,i-3} \quad 31) \quad (\bar{z}_{0,i-4} \quad 1) = 0, \tag{6}$$

where $\bar{z}_{0,i-4}, \ldots, \bar{z}_{0,i}$ are the successive states of the word $\bar{z}_0$ of the LFSR A.

The application of (6) to the keystream sequence $\{y_n\} = \{c_n\} + \{z_{0,n}\}$ formed by the output of function C $\{c_n\}$ and the output of LFSR $\{z_{0,n}\}$ results in the sequence $\{u_n\}$ where

$$u_i = y_i \quad y_{i-2} \quad (y_{i-3} \quad 31) \quad (y_{i-4} \quad 1), \quad u_i \in \mathbb{Z}/2^{32}\mathbb{Z}. \tag{7}$$

According to the idea of approximating the arithmetic addition modulo 2^{32} with the bitwise exclusive OR exploited in [13] the keystream word can be seen as $y_i = c_i \quad z_{0,i} \quad w_i$. The term $w_i \in \mathbb{Z}/2^{32}\mathbb{Z}$ is the noise induced by carries of the arithmetic addition. Hence from (7)we have

$$u_i = c_i \quad c_{i-2} \quad (c_{i-3} \quad 31) \quad (c_{i-4} \quad 1) \quad W_i, \tag{8}$$

where $W_i = w_i \quad w_{i-2} \quad (w_{i-3} \quad 31) \quad (w_{i-4} \quad 1)$. The idea of [13] now suggests to detect the bias of u_n as both w_i and c_i are biased.

For the observation of the assumed bias in the sequence $\{u_n\}_{n=0}^{N-1}$ of length N the word frequency statistic

$$\bar{\chi}^2 = \sum_{a=0}^{2^{32}-1} \frac{(\bar{} [a] - \lambda)}{\lambda} \tag{9}$$

is used, where $\bar{}[a]$ is the number of occurrences of the 32-bit word a in the sequence $\{u_n\}$ and λ is the awaited number of occurrences of each word for a random sequence ($\lambda = 1$ for $N = 2^{32}$). The values of $\bar{\chi}^2$ are supposed to be biased for the keystream of ABC when compared to the random sequence. The error probability of distinguishing can be estimated empirically by comparing the two sets of $\bar{\chi}^2$ values, one for the ABC keystream and another for a good (pseudo)random sequence.

Our computer experiments with the reduced versions of ABC v.2 [1] modeled the same distinguishing algorithm and employed good truly random sequences. The latter were obtained from a physical source of randomness. The experiments showed that distinguishing is completely impossible with time and data complexities of about 2^m for ABC with m-bit words.

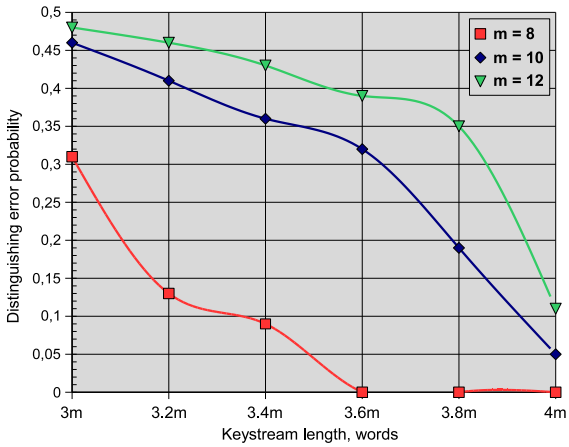


Fig. 2. Error probability for m-bit ABC distinguishers

To ensure that distinguishing attacks of this kind on ABC v.2 are impractical, extensive simulations on a high-performance computing cluster were performed. The results presented in Fig. 2 indicate that distinguishing of m-bit ABC for $m \geqslant 12$ in the way suggested in [13] with a negligible error probability cannot be carried out with time and data complexity less or equal to 2^{4m} and with memory complexity less or equal to 2^m (note that $m = 32$ for the full-size ABC v.2). Note that 2^{4m} the size of the corresponding key space. Therefore, our experiments suggest that efficiently distinguishing the full-scaled ABC v.2 from the random sequence requires more time resources than the 2^{128} brute force key search. This gives us grounds to conjecture that the application of distinguisher from [13] against ABC v.2 is nonsensical and totally impractical.

6 ABC v.2 and Stream Cipher Implementation Issues

In this section it is shown that ABC v.2 meets all the industrial implementation requirements mentioned above which make it perfectly suitable for various real-world applications including some embedded security systems.

6.1 ABC v.2 and Generic Performance

In many industrial applications it is difficult to optimize cryptographical algorithms for all concrete computer architectures. This is due to the following problems:

- High costs of assembly language implementations,
- Code portability requirement.

In practice even rather large firms cannot afford to pay for the optimized implementation of every cipher from the cryptographical library (the number of symmetrical cryptographical algorithms that are to be implemented within one library is often over 10) for every computer platform (the number of the platforms on which some consumers want to run their cryptographical libraries is often considerable and can exceed $10-20$). Such an implementation would require over $100 = 10 \cdot 10$ optimized realizations of individual cryptographical algorithms for specific computer platforms. This indicates that even inline assembly language sections can be not allowed. Another reason is that the industry wants to have the algorithms only once implemented and does not want to spend money every 6 months or 1 year (as new platforms demand immediate action rather frequently) for the same library again.

Thus, we consider the generic performance of a cipher an extremely important property for industrial applications. That is, a good stream cipher should be not only secure and at the least twice faster than AES. It should also provide the possibility of an easy and very efficient implementation in ANSI C using generic compilers (maybe with specific options).

We treat the generic implementation performance property as one of the most important stream cipher properties. For this reason we are not going to provide assembly language implementations of the ABC v.2 for the eSTREAM benchmarks since ABC v.2 holds its leading performance positions, even when implemented in ANSI C.

Here we present the results of the performance evaluation of ABC v.2. All the throughput values are gained for a generic implementation. Throughput values and costs of the setup routines for the reference implementation can be found in Table 1. The table contains results for different optimization window sizes obtained on a 3.2 GHz Intel Pentium 4 Northwood processor under the same measurement conditions as described in [7].

According to the performance benchmark tables published at the eSTREAM web site [2] ABC v.2 performance is very high. ABC v.2 is the fastest candidate at plain encryption on AMD64, PowerPC G4, and UltraSPARC-III processors,

Table 1. ABC v.2 performance for Intel Pentium 4

w	Speed, Gbps	Cycles per byte	Lookup tables, bytes	Key setup, cycles	IV setup, cycles
2	2.19	11.68	256	2056	372
4	3.36	7.65	512	4792	259
8	6.91	3.70	4096	90519	207

occupying the third place for HP9000 and second place (after Py or TRIVIUM) in the list for the rest of reported CPUs. Due to the rapid IV setup ABC v.2 is uneclipsed among Profile II (software ciphers) candidates at the encryption of short packets on all of the reported CPUs.

6.2 ABC v.2 Key and IV Setup Flexibility

ABC possesses a bundle of properties that make it fit well in various real-life applications and satisfy the industrial demands. One of these properties is the fast IV setup, which leads to the very effective packet encryption with a per-packet nonce. The ABC v.2 performance at packet encryption for various optimization window sizes (measured on an Intel Pentium M processor) is showed in Fig. 3.

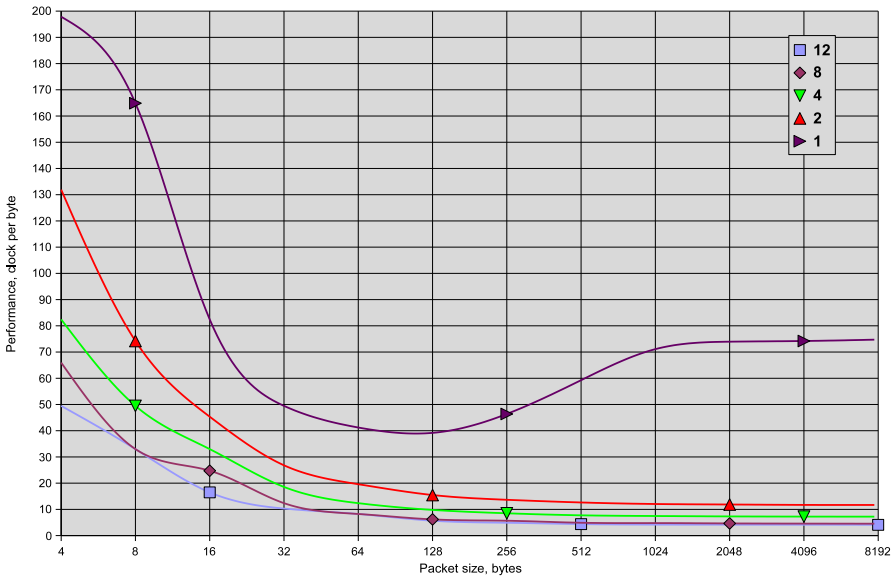


Fig. 3. ABC v.2 performance at packet encryption with IV setup

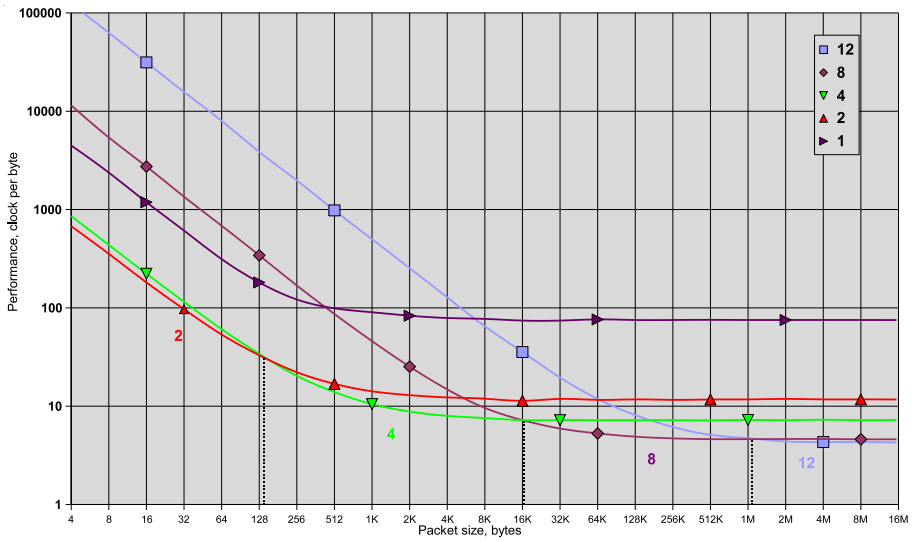


Fig. 4. ABC v.2 performance at packet encryption with key and IV setup

Another property is the natural flexibility of the ABC design, which allows one to choose the optimization parameters suitable for a given application. In practice a high demand for frequent key reinitializations means that short data segments are processed. The relatively costly key setup procedure of ABC v.2 for long optimization windows (e.g $w = 8$) is compensated by extremely high keystream generation performance of ABC v.2 with this parameter. If a less expensive key setup procedure is required, then a lower value of parameter w can be selected which results in a higher overall performance of the ABC v.2 key setup, IV setup and keystream generation routine. That is, the effect of a time-consuming key setup is easily leveled by choosing the appropriate optimization window size depending on the size of data blocks being processed. Note that the cipher remains in all the cases the same, the implementation parameters changing only. Figure 4 presents in what way the choice can be done, showing the performance of ABC v.2 at packet encryption with per-packet key and IV setup for various optimization window sizes. The relative cost of key and IV setup procedures at packet encryption is shown in Fig. 5. The measurements were performed on an Intel Pentium M processor.

6.3 ABC v.2 for Embedded Security Systems

The variable length of ABC v.2 optimization tables enables implementations with rather low memory consumption starting from 256 byte. ABC is the only cipher in the eSTREAM project providing a working 8-bit implementation [1].

The performance of ABC v.2 for a standard i8051 controller (Philips 80/87C51 microcontroller belonging to the MCS-51 family) can be found in Table 2

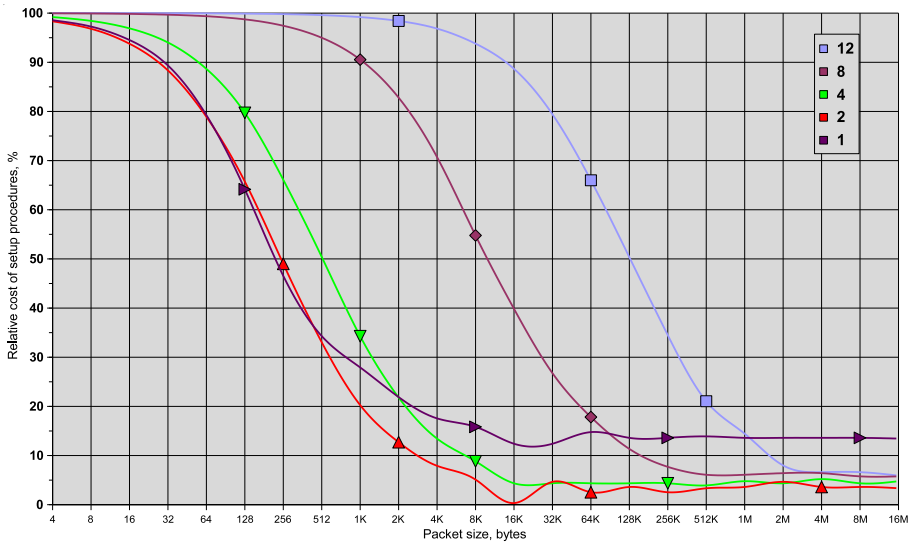


Fig. 5. Relative cost of ABC v.2 setup procedures at packet encryption

Table 2. Comparison of 8-bit ABC v.2, AES and RC6 implementations

Implementation	Code Size byte	Const byte	Ram Size IDATA+XRAM,byte	Clock cycles per byte	Key setup, clock cycles
ABC, w=2 [8]	1649	256	79+452	253	52562
ABC, w=4 [8]	1493	512	79+708	174	59854
AES [10]	760	256	65	198	
RC6 [10]	596	0	221	900	43200

which compares the implementation[3] of ABC v.2 with that for AES [10] and RC6 [11]. The performance of ABC v.2 was measured by encrypting 16-byte blocks fitting in the IRAM (internal RAM) of the microcontroller.

Contemporary smart cards possess as a rule several KByte RAM (1-4 KByte XRAM) which makes the implementation of ABC v.2 with $w = 4$ rather practicable. So, being primarily a 32-bit oriented cipher, ABC v.2 also performs well on constraint platforms. This indicates that the scope of application of ABC v.2 is not restricted by 32-bit platforms. Thus, the architecture of ABC v.2 is universal with respect to software implementations on numerous platforms.

A further RISC processor which is often applied in embedded systems and was not included in the eSTREAM benchmarks (as of January 2006) is the ARM microprocessor. The performance figures of ABC v.2 for ARM7 in comparison with those for AES and RC6 [11] can be found in Table 3.

[3] The implementation and also performance figures for ARM [8] are kindly provided by S. Kumar, COSY RUB, Germany.

Table 3. Comparison of ABC v.2, AES and RC6 on ARM

Implementation	Cycles per byte
ABC, w=2 [8]	97
ABC, w=4 [8]	55
ABC, w=8 [8]	35
AES [11]	91
RC6 [11]	49

7 Conclusion

In this paper we have presented ABC v.2 – a tweaked modification of the original ABC stream cipher. The tweaks increase the period of the ABC stream cipher in a simple way and also enlarge it's secret state. They totally eliminate the attacks described in [9] and [12]. ABC v.2 performance evaluation showed that the tweaks do not lead to significant overhead and that ABC v.2 is extremely fast in software often heading the eSTREAM benchmark list [2].

Also another way of thwarting these attacks was studied. It was explained why we preferred the way of [6]. It was also noted that the results stated in [13] are erroneous. Our computer experiments indicate that distinguishing ABC v.2 from the random sequence as suggested seems unreasonable as compared to the exhaustive key search.

The natural scalability of the ABC design was emphasized. It was shown that a version of ABC with a 256-bit key and a 256-bit IV offering 256-bit security can be made out of ABC v.2 at a very low performance cost.

Apart from its leading positions concerning software performance ABC v.2 meets a number of industrial software implementation properties such as generic performance property, flexible storage consumption and flexible cost of IV/key setup procedures. This makes ABC v.2 applicable not only on standard 32-bit platforms, but in some embedded security systems with high performance requirements as well. This was exhibited by the eSTREAM benchmarks: among eSTREAM candidates of software profile ABC v.2 is uneclipsed at such a real-life task as the encryption of short packets.

References

1. The ABC stream cipher page. http://crypto.rsuh.ru. 9, 12
2. eSTREAM optimized code HOWTO. http://www.ecrypt.eu.org/stream/perf. 1, 10, 14
3. Vladimir Anashin. Uniformly distributed sequences over p-adic integers. In I. Shparlinsky A. J. van der Poorten and H. G. Zimmer, editors, *Number theoretic and algebraic methods in computer science. Proceedings of the Int'l Conference (Moscow, June–July, 1993)*, pages 1–18. World Scientific, 1995. 7
4. Vladimir Anashin. Pseudorandom number generation by p-adic ergodic transformations, 2004. Available from http://arXiv.org/abs/cs.CR/0401030. 4, 6
5. Vladimir Anashin, Andrey Bogdanov, and Ilya Kizhvatov. ABC is safe and sound. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/079, 2005. http://www.ecrypt.eu.org/stream. 8
6. Vladimir Anashin, Andrey Bogdanov, and Ilya Kizhvatov. Increasing the ABC stream cipher period. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/050, 2005. http://www.ecrypt.eu.org/stream. 2, 14
7. Vladimir Anashin, Andrey Bogdanov, Ilya Kizhvatov, and Sandeep Kumar. ABC: A new fast flexible stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream. 1, 2, 7, 10
8. Vladimir Anashin, Andrey Bogdanov, Ilya Kizhvatov, and Sandeep Kumar. ABC: A new fast flexible stream cipher. Version 2, 2005. http://crypto.rsuh.ru/papers/abc-spec-v2.pdf. 1, 2, 4, 6, 7, 8, 13, 14
9. Côme Berbain and Henry Gilbert. Cryptanalysis of ABC. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/048, 2005. http://www.ecrypt.eu.org/stream. 2, 5, 6, 7, 14
10. Joan Daemen and Vincent Rijmen. The Rijndael block cipher. NIST, 1999. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf. 13
11. G. Hachez, F. Koeune, and J. Quisquater. cAESar results: Implementation of four AES candidates on two smart cards. In *Second Advanced Encryption Standard Candidate Conference*, pages 95–108, 1999. 13, 14
12. Shahram Khazaei. Divide and conquer attack on ABC stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/052, 2005. http://www.ecrypt.eu.org/stream. 1, 5, 6, 7, 14
13. Shahram Khazaei and Mohammad Kiaei. Distinguishing attack on the ABC v.1 and v.2. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/061, 2005. http://www.ecrypt.eu.org/stream. 1, 8, 9, 14
14. Alexander Klimov and Adi Shamir. A new class of invertible mappings. In B.S.Kaliski Jr.et al., editor, *Cryptographic Hardware and Embedded Systems 2002*, volume 2523 of *Lect. Notes in Comp. Sci*, pages 470–483. Springer-Verlag, 2003. 7
15. Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition, 1998. 7

DECIM^{v2} *

C. Berbain[1], O. Billet[1], A. Canteaut[2], N. Courtois[3], B. Debraize[3,4], H. Gilbert[1], L. Goubin[4], A. Gouget[5], L. Granboulan[6], C. Lauradoux[2], M. Minier[2], T. Pornin[7] and H. Sibert[5]

Abstract

DECIM is a hardware oriented stream cipher with 80-bit key and 64-bit IV which was submitted to the ECRYPT stream cipher project. The design of DECIM is based on both a nonlinear filter LFSR and an irregular decimation mechanism called the ABSG. As a consequence, DECIM is of low hardware complexity. Recently, Hongjun Wu and Bart Preneel pointed out two flaws in the stream cipher DECIM. The first flaw concerns the initialization stage and the second one, which is the more serious flaw, concerns the filter used in the keystream generation algorithm; the ABSG mechanism is not affected by these two flaws. In this paper, we propose a new version of DECIM, called DECIM^{v2}, which does not only appear to be more secure, but also has a lower hardware complexity than DECIM.

1 Introduction

DECIM [3] is a hardware oriented stream cipher submitted to the ECRYPT Stream Cipher Project [1]; we now call it DECIM^{v1}. It has been developed around the ABSG mechanism which provides a method for irregular decimation of pseudorandom sequences. The general running of DECIM^{v1} (and also DECIM^{v2}) consists in generating a binary sequence $\mathbf{y}$ in a regular way from a Linear Feedback Shift Register (LFSR) which is filtered by a Boolean function. The sequence $\mathbf{y}$ is next filtered by the ABSG mechanism.

Recently, Hongjun Wu and Bart Preneel [6] found two flaws in the stream cipher DECIM^{v1}. The first flaw concerns the initialization stage, i.e. the computation of the initial inner state for starting the keystream generation. In a nutshell, the initialization mechanism of DECIM^{v1} works as follows.

[1]France Télécom Recherche et Développement, 38/40 rue du Général Leclerc, F-92794 Issy les Moulineaux cedex 9, {come.berbain,olivier.billet,henri.gilbert}@francetelecom.com

[2]INRIA-Rocquencourt, projet CODES, domaine de Voluceau, B.P. 105, F-78153 Le Chesnay cedex, {anne.canteaut,marine.minier,cedric.lauradoux}@inria.fr

[3]Axalto Smart Cards, 36-38, rue de la Princesse - B.P. 45, F-78431 Louveciennes cedex, {ncourtois,bdebraize}@axalto.com

[4]Laboratoire PRiSM, Université de Versailles, 45 avenue des Etats-Unis, F-78035 Versailles cedex, louis.goubin@prism.uvsq.fr

[5]France Télécom Recherche et Développement, 42 rue des Coutures, BP 6243, F-14066 Caen cedex, {aline.gouget,herve.sibert}@francetelecom.com

[6]Département d'Informatique, École Normale Supérieure, 45 rue d'Ulm, F-75230 Paris cedex 05, louis.granboulan@ens.fr

[7]Cryptolog International, 16-18 rue Vulpian, F-75013 Paris, thomas.pornin@cryptolog.com

*Work partially supported by the French Ministry of Research RNRT Project "X-CRYPT" and by the European Commission via ECRYPT network of excellence IST-2002-507932.

1. Filling of the LFSR from a 80-bit secret key and a 64-bit public IV.
2. 192 updates of the LFSR. One update consists of the three following steps:
   (a) Computation of the feedback value (in a nonlinear way);
   (b) Application of one among two permutations over 7 elements of the current LFSR state; the choice of the permutation is controlled by the output of the ABSG;
   (c) Shifting by one position of the LFSR.

The aim of the permutations is to provide high nonlinearity during the initialization stage. However, the side effect of the permutations is that a large number of elements of the LFSR (after the initial filling) may never be updated with a high probability during the initialization process. This flaw allowed Hongjun Wu and Bart Preneel to mount an efficient key recovery attack on DECIM^{v1}. For DECIM^{v2}, we propose a simpler and more secure initialization procedure than the one of DECIM^{v1} (in particular, the permutations involved in the initialization procedure of DECIM^{v1}, which imply a significant increase of the hardware cost, are removed in DECIM^{v2}).

The main flaw pointed out by Hongjun Wu and Bart Preneel [6] is in the keystream generation algorithm which is described in Figure 1. More precisely, the flaw is in the generation

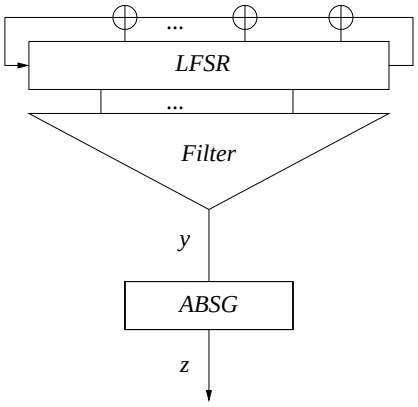


Figure 1: DECIM keystream generation

of the sequence $\mathbf{y}$ which is the output of the filter (the sequence $\mathbf{y}$ is next decimated by the ABSG mechanism). In a few words, this flaw is due to the fact that the sequence $\mathbf{y}$ is directly the output of a symmetric Boolean function which is not *correlation-immune of order* 1. There exists a correlation between the outputs of the function associated to two input vectors which have one element in common. By using this weakness, Hongjun Wu and Bart Preneel show a correlation between some bits of the keystream sequence and then they show that the keystream of DECIM^{v1} is heavily biased. For DECIM^{v2}, we propose a simpler and more secure filter than the one of DECIM^{v1} by choosing a filter which is correlation immune of order 1.

The outline of the paper is as follows. In Section 2, we give an overview of DECIM^{v2} and we describe the slight modifications between DECIM^{v1} and DECIM^{v2}. In Section 3, we provide a full description of DECIM^{v2}. In Section 4, we explain the design modifications. In Section 5,

we discuss the hardware implementation of DECIM^{v2}. In Section 6, we discuss the security properties of DECIM^{v2}. Finally, we conclude in Section 7.

2 Overview of Decim^{v2}

In accordance with the specification given by the Ecrypt stream cipher project, DECIM^{v2} takes as an input a 80-bit length secret key and a 64-bit length public initialization vector.

2.1 Keystream generation

The size of the inner state of DECIM^{v2} is unchanged, i.e. 192 bits. The keystream generation mechanism is described in Figure 2. The bits of the internal state of the LFSR are numbered from 0 to 191, and they are denoted by $(x_0, \ldots, x_{191})$. The sequence of the linear feedback values of the LFSR is denoted by $\mathbf{s} = (s_t)_{t \geq 0}$.

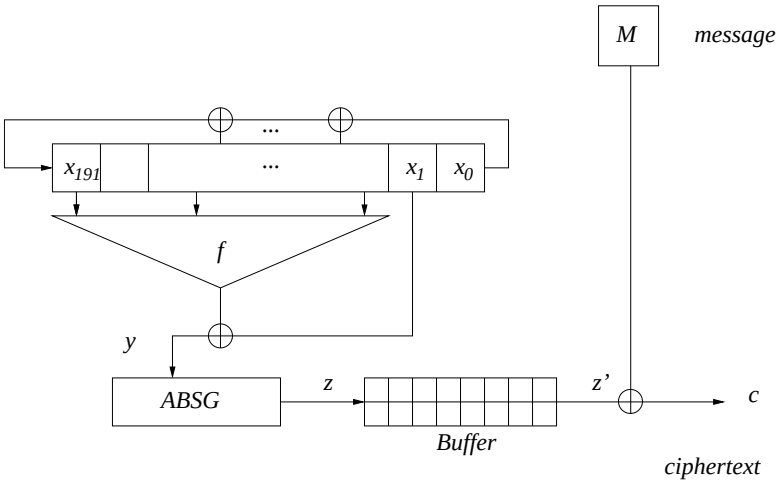


Figure 2: DECIM^{v2} keystream generation

The Boolean function f is a 13-variable quadratic symmetric function which is balanced. Let $x_{i_1}, \ldots, x_{i_{14}}$ denote the 14 initial internal state bits of the LFSR that are the inputs of the filter. The sequence $\mathbf{y}$ outputs by the filter is defined by:

$$y_t = f(s_{i_1+t}, \ldots, s_{i_{13}+t}) \oplus s_{i_{14}+t}$$

The ABSG takes as an input the sequence $\mathbf{y} = (y_t)_{t \geq 0}$. The sequence output by the ABSG is denoted by $\mathbf{z} = (z_t)_{t \geq 0}$. The buffer mechanism guarantees a constant throughput for the keystream; we choose a 32 bit-length buffer and the buffer outputs 1 bit for every 4 shifts by one position of the LFSR (see [3] for details).

Remark 1 *For the keystream generation, the gap between* DECIM^{v1} *and* DECIM^{v2} *is the choice of the filter. In* DECIM^{v1}*, the filter is a vectorial function defined by:*

$$F : \mathbb{F}_2^{14} \longrightarrow \mathbb{F}_2^2; \quad x_{i_1}, \ldots, x_{i_{14}} \mapsto (f(x_{i_1}, \ldots, x_{i_7}), f(x_{i_8}, \ldots, x_{i_{14}}))$$

where f is a 7-variable symmetric Boolean function which is balanced but which is not correlation immune of order 1.

2.2 Key/IV setup

The initial filling of the LFSR from the key and the initialization vector is modified in Decim^{v2} compared to Decim^{v1} (see Section 3). The Key/IV setup mechanism consists in clocking $4 \times 192 = 768$ times the LFSR using the nonlinear feedback which is described in Figure 3.

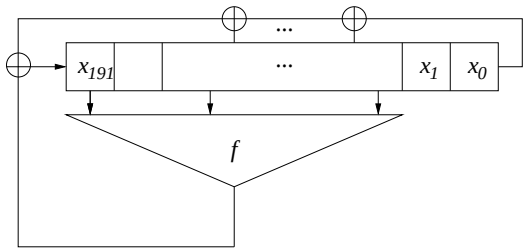


Figure 3: Key/IV setup mechanism

Remark 2 *For the initialization stage, the main differences between* Decim^{v1} *and* Decim^{v2} *are the filling of the LFSR which is changed, the deletion of the permutations and the choice of the filter. As a consequence, the number of clocks in the initialization stage increases from* 192 *up to* 768*.*

3 Specification

In this section, we describe each component of Decim^{v2} and we describe the changes between Decim^{v1} and Decim^{v2}; we refer to [3] when no modification has been done.

3.1 The filtered LFSR

This section describes the filtered LFSR that generates the sequence $\mathbf{y}$ (the sequence $\mathbf{y}$ is the input of the ABSG mechanism).

The LFSR (unchanged). The underlying LFSR is a maximum-length LFSR of length 192 over $\mathbf{F}_2$. It is defined by the following primitive feedback polynomial:

$$P(X) = X^{192} + X^{189} + X^{188} + X^{169} + X^{156} + X^{155} + X^{132} + X^{131} + X^{94} + X^{77} + X^{46} + X^{17} + X^{16} + X^5 + 1\ .$$

The filter (changed). The filter function is the 14-variable Boolean function defined by:

$$F : \mathbb{F}_2^{14} \longrightarrow \mathbb{F}_2; \quad a_1, \ldots, a_{14} \mapsto f(a_1, \ldots, a_{13}) \oplus a_{14}$$

where f is the symmetric quadratic Boolean function defined by:

$$f(a_1, \ldots, a_{13}) = \bigoplus_{1 \le i < j \le 13} a_i a_j \bigoplus_{1 \le i \le 13} a_i$$

The tap positions of the filter are:

$$191 - 186 - 178 - 172 - 162 - 144 - 111 - 104 - 65 - 54 - 45 - 28 - 13 - 1$$

and the input of the ABSG at the stage t is:

$$y_t = f(s_{t+191}, s_{t+186}, s_{t+178}, s_{t+172}, s_{t+162}, s_{t+144}, s_{t+111}, s_{t+104}, s_{t+65}, s_{t+54}, s_{t+45}, s_{t+28}, s_{t+13}) \oplus s_{t+1}$$

3.2 Decimation (unchanged)

This part describes how the keystream sequence $\mathbf{z}$ is obtained from the sequence $\mathbf{y}$. The ABSG algorithm is given in Figure 4.

Input: $(y_0, y_1, \dots)$
Set: $i \leftarrow 0$; $j \leftarrow 0$;
Repeat the following steps:
1. $e \leftarrow y_i$, $z_j \leftarrow y_{i+1}$;
2. $i \leftarrow i + 1$;
3. while $(y_i = \overline{e})$ $i \leftarrow i + 1$;
4. $i \leftarrow i + 1$;
5. output z_j
6. $j \leftarrow j + 1$

Figure 4: ABSG Algorithm

3.3 Buffer mechanism (unchanged)

The rate of the ABSG mechanism is irregular and therefore we use a buffer in order to guarantee a constant throughput. We choose a buffer of length 32 and for every 4 bits that are input into the ABSG, the buffer is supposed to output one bit exactly. With these parameters, the probability that the buffer is empty while it has to output one bit is less than 2^{-89}.

If the ABSG outputs one bit when the buffer is full, then the newly computed bit is not added into the queue, i.e. it is dropped. Assuming that the initial inner state is computed (it is denoted by $z_0, \dots, z_{191}$), the ABSG mechanism starts at the beginning loop and the buffer is empty. The keystream generation process starts when the buffer is full.

3.4 Key/IV Setup

This subsection describes the computation of the initial inner state for starting the keystream generation. Notice that the ABSG mechanism is not used anymore during the initialization stage.

3.4.1 Initial filling of the LFSR (changed)

The secret key K is a 80-bit key denoted by $K = K_0, \dots, K_{79}$ and the initialization vector IV is a 64-bit IV denoted by $IV_0, \dots, IV_{63}$.

The initial filling of the LFSR is done as follows.

$$x_i = \begin{cases} K_i & 0 \leq i \leq 79 \\ K_{i-80} \oplus IV_{i-80} & 80 \leq i \leq 143 \\ K_{i-80} \oplus IV_{i-144} \oplus IV_{i-128} \oplus IV_{i-112} \oplus IV_{i-96} & 144 \leq i \leq 159 \\ IV_{i-160} \oplus IV_{i-128} \oplus 1 & 160 \leq i \leq 191 \end{cases}$$

The number of possible initial values of the LFSR state is $2^{80+64} = 2^{144}$.

3.4.2 Update of the LFSR state

The LFSR is clocked $4 \times 192 = 768$ times using a nonlinear feedback relation. Let y_t denote the output of f at time t and let lv_t denote the linear feedback value at time $t > 0$. Then, the value of x_{191} at time t is computed using the equation:

$$x_{191} = lv_t \oplus y_t \ .$$

Notice that there is no bit of the LFSR state output during this step.

4 Design rationale

The rationale behind the design of Decim^{v2} relies on the fact that the main ideas behind Decim^{v1}, namely, to filter and then decimate the output of an LFSR using the ABSG mechanism was in no way questioned. Thus, the core of Decim^{v2} is a single Boolean function-based filtering, followed by an ABSG-based decimation.

4.1 The filter

In Decim^{v2} (and also in Decim^{v1}) a Boolean function is used to filter the LFSR whereas the Shrinking Generator or the Self-Shrinking Generator are both directly applied on LFSRs. The linear complexity of the sequence outputs by an LFSR with a primitive feedback polynomial is the length of the LFSR. The interest of the filter is to significantly increase the linear complexity of the sequence which is the input sequence of the ABSG mechanism. That comes to significantly increase the minimal length of the equivalent LFSR which generates the same sequence as those outputs by the filtered LFSR.

The choice of the filter is very important since the filter must not introduce some weaknesses in the stream cipher (as it is the case for Decim^{v1}). An important property for the filter is that the output of the filter must be uniformly distributed. In Decim^{v1}, the 7-variable Boolean function f used in the filter is balanced, i.e., the value of f is uniformly distributed in $\{0, 1\}$ when the evaluation of f is done uniformly over $\{0, 1\}^7$.

Decim^{v1} is a hardware-oriented stream cipher and the filter must have a low-cost hardware implementation. In Decim^{v1}, the filter is a symmetric Boolean function f (i.e. the value of f only depends on the Hamming weight of the input) in order to reduce the hardware cost and the function f is balanced.

The attack given by Hongjun Wu and Bart Preneel [6] has shown that it is important to choose a Boolean function f which is *correlation-immune of order* 1, i.e. a function such that there is no correlation between the outputs of the function associated to two input vectors

which have one element in common. Since the Boolean function f must also be balanced, that means that f must be 1-*resilient*. In DECIM^{v1}, the Boolean function is balanced but it is not 1-resilient.

The filter of DECIM^{v2} is constructed from a balanced 13-variable symmetric function (which is not correlation immune of order 1) and the whole filter F is a 1-resilient Boolean function.

4.2 Tap positions : filter and feedback polynomial

Assuming knowledge of the keystream $\mathbf{z}$, an attacker will have to guess some bits of the sequence $\mathbf{y}$ in order to attack the function f. The knowledge of the bits of $\mathbf{y}$ directly yields equations in the bits of the initial state of the LFSR. Thus, the number of monomials in the bits of the initial state of the LFSR that are involved in these equations has to be maximized. Moreover, this number has to grow quickly during the first clocks of the LFSR. This implies the following two conditions:

1. each difference between two positions of bits that are input to f should appear only once;
2. some inputs of f should be taken at positions near the one of the feedback bit (which means that some inputs should be leftmost on Figure 2).

Finally, the tap positions of the inputs of the Boolean function f and the inputs of the feedback relation should be independent.

4.3 Key/IV Setup

The components of the keystream generation are re-used for the key/IV setup; we do not introduce new components.

By using a 80-bit key and a 64-bit IV, the number of possible initial states is at most 2^{144} which is the case in DECIM^{v2} whereas the number of possible initial states is 2^{136} in DECIM^{v1}.

The first attack given in [6] exploits the effects of the permutations π_1 and π_2 used in the initialization process. Indeed, some bits of the LFSR are improperly updated. Then, the attack consists in tracing some bits during the initialization process. In DECIM^{v2}, the permutations are removed and the number of clocks of the register is increased in order to ensure that the nonlinearity of the initialization stage is sufficient.

5 Hardware implementation

The number of gates involved in an hardware implementation can be estimated as follows, based on the estimation for elementary components given in [2], i.e., 12 gates for a flip-flop, 2.5 gates for an XOR, 1.5 gates for an AND and 5 gates for a MUX.

Here, we have the following values for each component in the circuit:

- LFSR: 2339 gates corresponding to 192 flip-flops and 14 XORs (instead of 3334 gates for DECIM^{v1}).
- Filtering function: 86.5 gates corresponding to 6 Full Adders and 7 XORs (instead of 74 gates for DECIM^{v1}; details on the hardware implementation of quadratic symmetric functions are given in [3]).

- 1-input ABSG, as described in Figure 5: 67 gates corresponding to 2 MUX, 3 XORs, 1 AND, and 4 flip-flops.

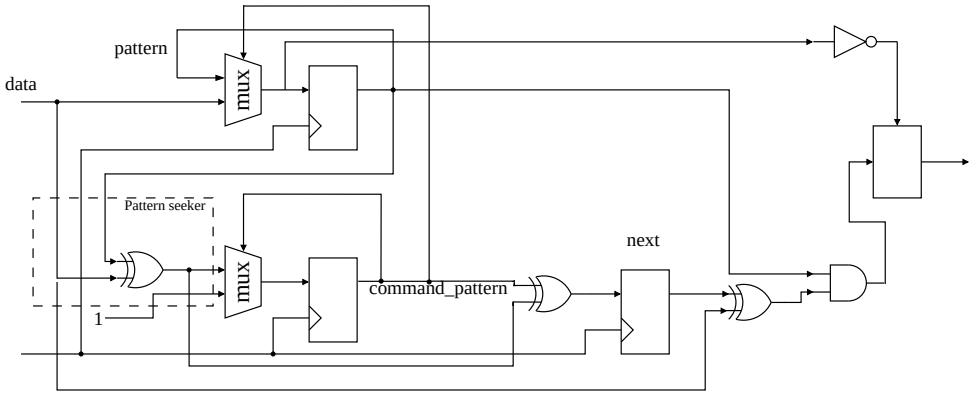


Figure 5: Hardware implementation of the ABSG

Remark 3 *For the proposed hardware implementation, the main differences between* DECIM^{v1} *and* DECIM^{v2} *is that the LFSR has now to be clocked 4 times instead of 2 before outputting a bit, i.e.* DECIM^{v2} *is twice lower than* DECIM^{v1}*.*

Moreover, the throughput of the generator can be doubled at a low implementation cost by using a simple speed-up mechanism. This can be done with a circuit which computes two feedback bits for the LFSR, simultaneously, as described in [3, Section 6.1]. This LFSR with doubled clock rate can be implemented within 192 flip-flops and 28 XORs. One additional copy of the filtering function is also required, and a 2-input ABSG mechanism must be used (see [3] for further details).

6 Security properties

The discussion given in [3] on guess-and-determine attacks, distinguishing attacks and also side channel attacks holds for DECIM^{v2}. Clock-controlled linear feedback shift registers, i.e. LFSRs that are irregularly clocked according to a decimation sequence which defines the number of symbols to be deleted before the next output symbol is produced, are immune to fast correlation attacks [5]. In [4], Golic developed a theory of fast correlation attacks on irregularly clocked LFSRs based on a linear statistical weakness. This attack may be realistic in special cases but DECIM^{v2} may be immune to such type of attack. Indeed, in order to increase the linear complexity of the sequence (i.e. the minimal length of the equivalent LFSR that generates the same sequence) that is shrunked by the ABSG mechanism, we use an LFSR which is filtered by a Boolean function. Like this, the expected linear complexity of the sequence outputs by the Boolean function is 17472, i.e. the expected minimal length of the LFSR that generates the same sequence as those generated by the filtered LFSR of DECIM is 17472.

7 Conclusion

We have proposed a new stream cipher DECIM^{v2}. The design is based on the eStream proposal DECIM^{v1} and addresses all weaknesses found in the original construction. A complete description of DECIM^{v2} was given and the differences from DECIM^{v1} were discussed.

The stream cipher DECIM^{v2} is especially suitable for hardware applications with restricted resources such as limited storage or gate count. For applications requiring higher throughputs, speed-up mechanisms can be used to accelerate DECIM^{v2} at the expense of a higher hardware complexity.

Acknowledgements. The authors wish to thank Frédéric Muller and Matt Robshaw for helpful comments.

References

[1] eStream, Stream cipher project of the European Network of Excellence in Cryptology ECRYPT. `http://www.ecrypt.eu.org/stream/`.

[2] L. Batina, J. Lano, S.B. Örs, B. Preneel, and I. Verbauwhede. Energy, perfomance, area versus security trade-offs for stream ciphers. In *The State of the Art of Stream Ciphers: Workshop Record*, pages 302–310, Brugge, Belgium, October 2004.

[3] C. Berbain, O. Billet, A. Canteaut, N. Courtois, B. Debraize, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert. Decim – A new Stream Cipher for Hardware applications. In *ECRYPT Stream Cipher Project Report 2005/004*. Available at `http://www.ecrypt.eu.org/stream/`.

[4] J. Golić. Towards fast correlation attacks on irregularly clocked shift registers. In *Proceedings of Eurocrypt'95*, Lecture Notes in Computer Science, 1995.

[5] Willi Meier and Othmar Staffelbach. Fast correlation attacks on certain stream ciphers. *J. Cryptol.*, 1(3):159–176, 1989.

[6] Hongjun Wu and Bart Preneel. Cryptanalysis of Stream Cipher Decim. Available at `http://www.ecrypt.eu.org/stream/`.

Status of Achterbahn and Tweaks

Berndt M. Gammel, Rainer Göttfert and Oliver Kniffler

Infineon Technologies AG
81726 Munich
Germany

berndt.gammel@infineon.com
rainer.goettfert@infineon.com
oliver.kniffler@infineon.com

Abstract

We report on the results of computations concerning the linear complexities of the NLFSRs deployed in Achterbahn's keystream generator. We outline a probabilistic algorithm for estimating the linear complexities of binary sequences of period $2^N - 1$. We define Achterbahn-Version 2 whose keystream generator consists of ten shift registers. We introduce the new combining function. We discuss recent cryptanalysis results against Achterbahn-Version 1. The last part of the paper is concerned with hardware optimization of the feedback functions of the deployed nonlinear primitive shift registers.

Keywords: Stream cipher, NLFSR, linear complexity, probabilistic algorithm, keystream generator.

1 Introduction

Achterbahn is a binary additive stream cipher. The keystream generator (KSG) of Achterbahn-Version 1 consists of eight nonlinear primitive binary feedback shift registers of lengths N between 22 and 31. The KSG of Achterbahn-Version 2 consists of ten primitive shift registers of lengths between 19 and 32. We call an N-stage feedback shift register *primitive* if it produces a sequence of least period $2^N - 1$ for every nonzero initial state $\mathbf{s}_0 \in \mathbb{F}_2^N = \{0,1\}^N$. Both versions of Achterbahn were designed for 80-bit secret key size and support initial values up to 80 bits.

The sequences produced by the eight, respectively ten, nonlinear feedback shift registers (NLFSRs) are combined by a Boolean combining function $R : \mathbb{F}_2^8 \to \mathbb{F}_2$, respectively $S : \mathbb{F}_2^{10} \to \mathbb{F}_2$, to produce the keystream $\zeta = (z_n)_{n=0}^{\infty}$. In reduced Achterbahn the sequences to be combined are the standard output sequences of the NLFSRs (corresponding to given initial states of the shift registers). The standard output sequence of a feedback shift register is obtained by emitting the content of the right-most cell D_0 of the shift register at each clock pulse (assuming that the shifts are performed from left to right).

In full Achterbahn each NLFSR is endowed with a configurable linear feedforward output function controlled by the secret key and the initial value. The produced output sequence $\tau = (t_n)_{n=0}^{\infty}$ is a linear combination of the standard output sequence $\sigma = (s_n)_{n=0}^{\infty}$ and some shifted versions thereof. For instance, let us assume that $t_n = s_n + s_{n+1} + s_{n+4}$ for $n \geq 0$. We then write $\tau = f(T)\sigma$, where $f \in \mathbb{F}_2[x]$ is called the *filter polynomial* and T denotes the shift operator on the $\mathbb{F}_2$-vector space $\mathbb{F}_2^{\infty}$ under termwise operations on sequences. That is, $T\sigma = (s_{n+1})_{n=0}^{\infty}$ for all binary sequences $\sigma = (s_n)_{n=0}^{\infty}$. In the above example, $f(x) = 1 + x + x^4$.

Notice that if all applied filter polynomials are equal to the constant polynomial $f(x) = 1$, the keystream produced by full Achterbahn—under this specific configuration of the output functions—is identical to the keystream produced by reduced Achterbahn. In other words, the KSG of reduced Achterbahn is contained in the KSG of full Achterbahn as a special case. An implementation of full Achterbahn can, therefore, also be operated in the *reduced Achterbahn mode*. A millionaire possessing full Achterbahn can exchange secret information with a pauper who can only afford low cost reduced Achterbahn.

2 Linear complexity of the keystream

Any two nonzero standard output sequences of a primitive feedback shift register have the same minimal polynomial and, therefore, the same linear complexity, which we call *the linear complexity* of the shift register.

Throughout this report, we use the following abbreviations. The lengths of the shift registers are denoted by $N_1, N_2, \ldots$. The linear complexities of the shift registers are designated by $L_1, L_2, \ldots$. The least periods of the nonzero output sequences of the shift registers are denoted by $P_1, P_2, \ldots$. Thus, $P_i = 2^{N_i} - 1$ for all i. A nonzero standard output sequence of the ith shift register is denoted by σ_i. The filter polynomials defining the linear feedforward output functions are denoted by $f_1, f_2, \ldots$. The Boolean combining functions of Achterbahn-Version 1 and Version 2 are designated by $R(x_1, \ldots, x_8)$ and $S(x_1, \ldots, x_{10})$, respectively. The keystream is denoted by $\zeta = (z_n)_{n=0}^{\infty}$. Thus, for instance, in the case of reduced Achterbahn-Version 1, we have $\zeta = R(\sigma_1, \ldots, \sigma_8)$, and in the case of full Achterbahn-Version 2, $\zeta = S(f_1(T)\sigma_1, \ldots, f_{10}(T)\sigma_{10})$.

Suppose we are given $t \geq 1$ primitive binary NLFSRs of lengths $N_1, \ldots, N_t$ and linear complexities $L_1, \ldots, L_t$. Let $\sigma_1, \ldots, \sigma_t$ be standard output sequences of the t shift registers corresponding to any nonzero initial states. Let $F(x_1, \ldots, x_t)$ be an arbitrary Boolean function of t variables. Let $\zeta = R(\sigma_1, \ldots, \sigma_t)$, that is $\zeta = (z_n)_{n=0}^{\infty}$ with $z_n = F(\sigma_1(n), \ldots, \sigma_t(n))$ for $n = 0, 1, \ldots$.

If the lengths $N_1, \ldots, N_t$ of the t shift registers are pairwise relatively prime, then the linear complexity $L(\zeta)$ of ζ can be expressed as

$$L(\zeta) = F(L_1, \ldots, L_t) \tag{1}$$

with the understanding that F is now regarded as a function over the integers. Formula (1) is well known for primitive LFSRs under less restrictive assumptions on the lengths of the shift registers [10]. For primitive NLFSRs of pairwise relatively prime lengths, the formula is implicitly contained in [10, Corollary 6], [9, Theorem 5], and [2, Theorem 3].

If the lengths of the primitive NLFSRs are not pairwise relatively prime, then equation (1) does not hold. In this case, $F(L_1, \ldots, L_t)$ provides only an upper bound for $L(\zeta)$. However, in many cases, it is still possible to derive a reasonable lower bound for the linear complexity of ζ.

Lemma 1. *Let $\sigma_1, \ldots, \sigma_t$ be nonzero output sequences of primitive binary NLFSRs of lengths $N_1, \ldots, N_t$, respectively, and with linear complexities $L_1, \ldots, L_t$, respectively. Let $F(x_1, \ldots, x_t)$ be a Boolean function of algebraic degree $d \geq 1$. A lower bound for the linear complexity of the sequence $\zeta = F(\sigma_1, \ldots, \sigma_t)$ can be given if the following two conditions are fulfilled:*

1. *The algebraic normal form (ANF) of $F(x_1, \ldots, x_t)$ contains a monomial $x_{i_1} x_{i_2} \cdots x_{i_d}$ of degree d for which the corresponding shift register lengths $N_{i_1}, \ldots, N_{i_d}$ are pairwise relatively prime.*
2. *For all other monomials of degree d, which have the form $x_{i_1} \cdots x_{i_{j-1}} x_k x_{i_{j+1}} \cdots x_{i_d}$, we have $\gcd(N_{i_j}, N_k) = 1$.*

If both assumptions are true, then

$$L_{i_1} L_{i_2} \cdots L_{i_d} \leq L(\zeta). \tag{2}$$

Proof. We only give a sketch of the proof. See [2] for more details. We first recall some facts of [11, Chap. 4]. Let $f, g, \ldots, h$ be binary polynomials of positive degree and with nonzero constant terms. Then $f \vee g \vee \cdots \vee h \in \mathbb{F}_2[x]$ is defined to be the polynomial whose roots are the distinct products $\alpha\beta\cdots\gamma$, where α is a root of f, β a root of g, and γ a root of h. The polynomial $f \vee g \vee \cdots \vee h$ is irreducible if and only if the polynomials $f, g, \ldots, h$ are all irreducible and of pairwise relatively prime degrees. In this case, $\deg(f \vee g \vee \cdots \vee h) = \deg(f)\deg(g)\cdots\deg(h)$.

Let the canonical factorization of the minimal polynomial of σ_k over $\mathbb{F}_2$ be given by

$$m_{\sigma_k} = \prod_{j_k=1}^{c_k} h_{j_k} \quad \text{for } k = 1, \ldots, t.$$

The polynomials h_{j_k} are distinct binary irreducible polynomials with $\deg(h_{j_k}) > 1$ and $\deg(h_{j_k})$ divides N_k.

Consider d sequences of $\{\sigma_1, \ldots, \sigma_t\}$. For simplicity of notation, say, $\sigma_1, \ldots, \sigma_d$. We associate to the sequences $\sigma_1, \ldots, \sigma_d$ the polynomial

$$f_{12\ldots d} = \prod_{j_1=1}^{c_1} \cdots \prod_{j_d=1}^{c_d} (h_{j_1} \vee \cdots \vee h_{j_d}). \tag{3}$$

If $N_1, \ldots, N_d$ are pairwise relatively prime, then $f_{12\ldots d}$ is the minimal polynomial of the product sequence $\sigma_1 \ldots \sigma_d$. In fact, (3) represents the canonical factorization of the minimal polynomial. Using $\deg(h_{j_1} \vee \cdots \vee h_{j_d}) = \deg(h_{j_1}) \cdots \deg(h_{j_d})$, we obtain for the linear complexity of $\sigma_1 \cdots \sigma_d$:

$$\begin{aligned} L(\sigma_1 \cdots \sigma_d) = \deg(f_{12\ldots d}) &= \sum_{j_1=1}^{c_1} \cdots \sum_{j_d=1}^{c_d} \deg(h_{j_1} \vee \cdots \vee h_{j_d}) \\ &= \prod_{k=1}^{d} \left(\sum_{j_k=1}^{c_k} \deg(h_{j_k}) \right) = \prod_{k=1}^{d} L(\sigma_k) = \prod_{k=1}^{d} L_k. \end{aligned}$$

This explains why we need the first requirement in the theorem. The second requirement guarantees that no other products of sequences appearing in $\zeta = F(\sigma_1, \ldots, \sigma_t)$ will cancel out some irreducible factors of the the polynomial in (3) □

In order to assign a numerical value to to lower bound for $L(\zeta)$ derived in Lemma 1, we need to know either the exact numerical values or at least lower bounds for the linear complexities $L_1, \ldots, L_t$ of the deployed shift registers.

It should be mentioned that a general nontrivial lower bound for the linear complexity L of a nonzero output sequence of a primitive binary N-stage feedback shift register is not known. We have, of course, $N \leq L \leq 2^N - 2$. The trivial lower bound $L = N$ is attained if and only if the primitive shift register is linear. For nonlinear primitive shift registers experimental results show that mostly the upper bound $L = 2^N - 2$ is attained (in over 50% of our observations). We also observed that occasionally the linear complexity L drops below the value 2^{N-1}. This happened in 0.00003% of our observations comprising about 10^8 primitive NLFSRs. The situation is different compared to de Bruijn sequences [8], where the linear complexity of the sequence never drops below the value $2^{N-1} + N$.

Since no nontrivial lower bounds for binary primitive NLFSR-output sequences have been proved in the literature, we have to roll our sleeves up and determine lower bounds for the numbers L_i by way of computation. We did this in two ways, using the Berlekamp-Massey algorithm and using a new probabilistic algorithm.

The KSG of Achterbahn-Version 1 consists of eight NLFSRs of lengths $N = 22$, 23, 25, 26, 27, 28, 29, and 31. For the first three shift registers we found, applying the Berlekamp-Massey algorithm, $L_1 = 2^{22} - 13$, $L_2 = 2^{23} - 2$, and $L_3 = 2^{25} - 2$. For the remaining five shift registers we verified that $L_i \geq 2^{25.8}$ for $i = 5, \ldots, 8$, using again the Berlekamp-Massey algorithm. Using the probabilistic algorithm [5], we found that with probability $> 1 - 2^{-100}$ all eight NLFSRs have linear complexities $L \geq 2^{N-1}$, if N denotes the length of the shift register.

The KSG of Achterbahn-Version 2 consists of ten primitive NLFSRs of lengths $N =$ 19, 22, 23, 25, 26, 27, 28, 29, 31, and 32. With the Berlekamp-Massey algorithm we found $L_1 = 2^{19} - 2$, $L_2 = 2^{22} - 2$, $L_3 = 2^{23} - 2$, $L_4 = 2^{25} - 2$, and verified that $L_i \geq 2^{25.2}$ for $i = 5, \ldots, 10$. Using the probabilistic algorithm, we verified for all ten shift registers that $L \geq 2^{N-1}$ with probability of error $< 2^{-100}$.

We outline the basic ideas of the used probabilistic algorithm. Let us use a primitive NLFSR of length $N = 31$ as an example. Let $\sigma = (s_n)_{n=0}^{\infty}$ be any standard output sequence of the shift register corresponding to a nonzero initial state. We want to verify that the linear complexity of σ is greater than half the period of σ. The least period of σ is $P = 2^{31} - 1$. The polynomial $x^P - 1$ is a characteristic polynomial of σ. We have

$$x(x^P - 1) = x^{2^{31}} - x = x(x-1) \prod_{\substack{f \text{ irred.} \\ \deg(f)=31}} f(x),$$

where the product is extended over all binary irreducible polynomials of degree 31. It is easily seen that the minimal polynomial m_σ of σ does not contain the polynomials x or $x - 1$ as factors. Since the minimal polynomial of a periodic sequence divides any characteristic polynomial of the sequence, we conclude that m_σ is the product of distinct irreducible binary polynomials of degree 31. If m_σ contains more than one half of all irreducible polynomials of degree 31, then we know that the linear complexity of σ must be greater than half the period of σ.

Given a certain irreducible polynomial f of degree 31, we can check whether or not f is a factor of m_σ in the following way:

1. Compute the polynomial $g_f(x) = (x^P - 1)/f(x)$;
2. Check whether $g_f(T)\sigma \neq \mathbf{0}$.

Again, T denotes the shift operator, and $\mathbf{0}$ represents the zero sequence. The following lemma is crucial.

Lemma 2. *The polynomial f divides m_σ if and only if $g_f(T)\sigma \neq \mathbf{0}$. Furthermore, $g_f(T)\sigma \neq \mathbf{0}$ if and only if the first $N = \deg(f)$ terms of the sequence $\tau = g_f(T)\sigma$ are not all zero.*

Algorithm:

1. Choose at random a binary irreducible polynomial f of degree $N = 31$.
2. Check whether $g_f(T)\sigma \neq \mathbf{0}$.
3. Repeat the first two steps k times.

If in all k experiments $g_f(T)\sigma \neq \mathbf{0}$, then the statement $L(\zeta) \geq 2^{N-1}$ is true with probability $\geq 1 - 2^{-k}$.

The Boolean combining function $S(x_1, \ldots, x_{10})$ for Achterbahn-Version 2, defined in equation (9) below, has algebraic degree $d = 4$. The ANF of S contains the following 22 monomials of degree 4:

$$\begin{aligned} &x_1x_3x_6x_8,\ x_1x_3x_6x_9,\ x_1x_4x_6x_8,\ x_1x_4x_6x_9,\ x_1x_5x_6x_8,\ x_1x_5x_6x_9,\ x_2x_3x_6x_8,\\ &x_2x_3x_6x_9,\ x_2x_4x_6x_8,\ x_2x_4x_6x_9,\ x_2x_5x_6x_8,\ x_2x_5x_6x_9,\ x_4x_5x_8x_{10},\ x_4x_5x_9x_{10},\\ &x_4x_6x_7x_8,\ x_4x_6x_7x_9,\ x_4x_6x_8x_{10},\ x_4x_6x_9x_{10},\ x_5x_6x_7x_8,\ x_5x_6x_7x_9,\ x_5x_7x_8x_{10},\\ &x_5x_7x_9x_{10}. \end{aligned} \tag{4}$$

We use Lemma 1 to lower bound $L(\zeta)$. The monomial with highest indices satisfying condition 1 of Lemma 1 is

$$x_4x_6x_9x_{10}. \tag{5}$$

The lengths of the corresponding shift registers, $N_4 = 25$, $N_6 = 27$, $N_9 = 31$, $N_{10} = 32$, are pairwise relatively prime. There are exactly two monomials in (4) that overlap with the monomial in (5) in three positions, namely the monomials

$$x_4x_5x_9x_{10} \quad \text{and} \quad x_4x_6x_8x_{10}.$$

We have $\gcd(N_5, N_6) = \gcd(26, 27) = 1$ and $\gcd(N_8, N_9) = \gcd(29, 31) = 1$. Thus condition 2 in Lemma 1 is satisfied. Using $L_i \geq 2^{N_i - 1}$ for $i = 1, \ldots, 10$, we conclude that

$$L(\zeta) \geq L_4L_6L_9L_{10} > 2^{24} \cdot 2^{26} \cdot 2^{30} \cdot 2^{31} = 2^{111}.$$

Those of us who only trust results derived by the application of a deterministic algorithm, can use $L_i \geq 2^{25.2}$. It then follows that

$$L(\zeta) > 2^{100}.$$

Otherwise we can use the afore mentioned results derived by the described probabilistic algorithm.

Theorem 1. *The linear complexity of the keystream of Achterbahn-Version 2 satisfies* $L(\zeta) > 2^{100}$ *with certainty and* $L(\zeta) > 2^{111}$ *with probability* $> 1 - 2^{-100}$.

3 Definition of Achterbahn-Version 2

The Boolean combining function in the initial proposal of Achterbahn [3] is given by

$$R(x_1, \ldots, x_8) = x_1 + x_2 + x_3 + x_4 + x_5x_7 + x_6x_7 + x_6x_8 + x_5x_6x_7 + x_6x_7x_8. \quad (6)$$

Johansson, Meier and Muller [6] described two attacks against Achterbahn exploiting certain weaknesses of R. We responded in posting the following "improved combining functions" at the eSTREAM page [4]

$$R'(x_1, \ldots, x_8) = R(x_1, \ldots, x_8) + x_5x_6 + x_5x_8 + x_7x_8. \quad (7)$$

and

$$R'' = x_1 + x_2 + x_3 + \sum_{4 \le i < j \le 8} x_ix_j + \sum_{4 \le i < j < k \le 8} x_ix_jx_k + \sum_{4 \le i < j < k < l \le 8} x_ix_jx_kx_l. \quad (8)$$

Although the functions R' and R'' were meant as examples and never declared to be successor functions for R, in a recent report [7], Johansson, Meier and Muller demonstrated that Achterbahn with its initial combining function replaced by R' or R'' can also be broken.

Before we discuss the attacks found in [7] in detail, we make some general observations regarding desired properties of combining functions to be used in NLFSR-based combining generators, like the KSG of Achterbahn.

3.1 Some general remarks

A joint weakness of the three combining functions R, R' and R'' is that they all contain several variables linearly. This fact was exploited in the first attack in [6] and in the TMO-attack in [7] as well.

The following argument shows why variables should not appear linearly. Consider the function $R(x_1, \ldots, x_8)$ in (6) and the polynomial

$$g(x) = (x^{P_1} - 1)(x^{P_2} - 1)(x^{P_3} - 1)(x^{P_4} - 1),$$

where $P_i = 2^{N_i} - 1$ are the periods of the shift register output sequences $\sigma_1, \ldots, \sigma_4$. The polynomial $g(x)$ is a characteristic polynomial of $\sigma = \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4$, that is $g(T)\sigma = \mathbf{0}$. Therefore, if we apply the linear operator $g(T)$ to the keystream

$$\zeta = \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4 + \sigma_5\sigma_7 + \sigma_6\sigma_7 + \sigma_6\sigma_8 + \sigma_5\sigma_6\sigma_7 + \sigma_6\sigma_7\sigma_8,$$

we obtain

$$g(T)\zeta = g(T)(\sigma_5\sigma_7 + \sigma_6\sigma_7 + \sigma_6\sigma_8 + \sigma_5\sigma_6\sigma_7 + \sigma_6\sigma_7\sigma_8),$$

a sequence depending only on the states of the last four shift registers.

Even in the case when a variable does not appear linearly in the ANF of a Boolean function, but still with low degree, the influence of the corresponding shift register can

be undone by applying the linear operator $g(T)$ to the keystream, were g is sparse and has relatively small degree. For instance, if $F(x_1, x_2, x_3, x_4) = x_1x_2 + x_2x_3 + x_1x_3x_4$, the sequence $\tau = g(T)\zeta$ is independent of σ_2 (and thus, independent of the contents of the second shift register) for

$$g(x) = (x^{P_1P_2} - 1)(x^{P_2P_3} - 1).$$

Therefore, another requirement for the Boolean function should be that it contains each variable in a monomial of maximal degree.

Yet another important rule is that for each variable there exists a monomial in the ANF of the function which has maximum degree and has the property that the shift register lengths corresponding to the variables in that monomial are pairwise relatively prime. The last requirement implies that no polynomial of small degree (compared to the linear complexity of the keystream) exists—dense or sparse—that could cancel out the influence of one or several shift registers, when applied to the keystream in the above sense.

3.2 The combining function

While most attacks in [7] could easily be avoided by making sure that the used Boolean function has maximum nonlinearity (for the given order of resiliency) and contains all of its variables in a monomial of maximum degree, there is one attack described in [7] which is quite aggressive. In this attack one guesses the content of one shift register and uses a linear approximation as a mean to confirm or reject the guess. The authors use only linear approximations in [7]. However, if we also take into account quadratic and cubic approximations in combination with the described guessing trick, we see that Achterbahn-Version 1 can always be successfully attacked no matter what Boolean combining function has been chosen. The reason is that the small number of eight variables imposes a severe restriction to the order of correlation immunity and nonlinearity of the function.

In order to avert attacks based on quadratic approximations, we need a combining function of ten variables. As a consequence, the KSG of Achterbahn-Version 2 will consist of ten primitive NLFSRs.

The combining function for Achterbahn-Version 2 is given by

$$\begin{aligned} S(x_1, \ldots, x_{10}) &= x_1 + x_2 + x_3 + x_9 + G(x_4, x_5, x_6, x_7, x_{10}) \\ &\quad + (x_8 + x_9)(G(x_4, x_5, x_6, x_7, x_{10}) + H(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_{10})) \end{aligned} \tag{9}$$

with

$$\begin{aligned} G(x_4, x_5, x_6, x_7, x_{10}) &= x_4(x_5 \vee x_{10}) + x_5(x_6 \vee x_7) + x_6(x_4 \vee x_{10}) \\ &\quad + x_7(x_4 \vee x_6) + x_{10}(x_5 \vee x_7) \end{aligned}$$

and

$$\begin{aligned} H(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_{10}) &= x_2 + x_5 + x_7 + x_{10} + (x_3 + x_4)\bar{x}_6 \\ &\quad + (x_1 + x_2)(x_3\bar{x}_6 + x_6(x_4 + x_5)), \end{aligned}$$

where $a \vee b = a + b + ab$ and $\bar{a} = a + 1$ for $a, b \in \mathbb{F}_2$.

Function S has resiliency 5 and nonlinearity 448. The ANF of S contains 77 monomials, 22 thereof have degree 4. The function can be implemented in hardware with 63 GE. Each of the ten variables of S appears in a monomial of degree 4.

Since S has ten variables, we need another two NLFSRs. We choose shift registers of lengths 19 and 32.

Tweak: The KSG of Achterbahn-Version 2 consists of ten primitive binary NLFSRs of lengths 19, 22, 23, 25, 26, 27, 28, 29, 31, and 32. The maximum degrees of the corresponding filter polynomials describing the linear feedforward output functions of full Achterbahn are 3, 3, 3, 5, 6, 7, 8, 9, 10, 10.

Theorem 2. *The keystream ζ produced by the KSG of reduced Achterbahn-Version 2, as well as all 2^{64} translation distinct keystream sequences produced by full Achterbahn-Version 2, have least period*

$$\mathrm{Per}(\zeta) = \frac{1}{135} \prod_{i=1}^{10} \left(2^{N_i} - 1\right) > 2^{254}.$$

Consider the 22 monomials in (4). Each of the ten variables $x_1, \ldots, x_{10}$ appears in at least one monomial for which the corresponding shift register lengths are pairwise relatively prime. Due to this property and the verified fact that $L_i \geq 2^{N_i - 1}$ for $i = 1, \ldots, 10$, the following theorem can be proved.

Theorem 3. *Let ζ be a keystream produced by reduced or full Achterbahn-Version 2. For each polynomial $g \in \mathbb{F}_2[x]$ with $\deg(g) < 2^{80}$, the sequence $\tau = g(T)\zeta$ depends on all ten NLFSRs.*

3.3 Cryptanalysis of Achterbahn-Version 1

We now compare the complexities of all attacks described in [7] that were successfully applied against Achterbahn-Version 1 with combining functions R, R', or R'' with the complexity of the attack against Achterbahn-Version 2 with combining function S.

The attack described in [7, Sec. 4] makes use of the the fact that the function $R(x_1, \ldots, x_8)$ in (6) becomes linear for $x_5 = x_6 = 0$. The lengths of the corresponding shift registers are 27 and 28, which are the relevant parameters for the complexity of the attack. The complexity is $O(2^{27+28+1}) = O(2^{56})$ for reduced and $O(2^{73})$ for full Achterbahn-Version 1. The function $S(x_1, \ldots, S_{10})$ in (9) becomes only linear if we set at least five of the variables $x_4, x_5, x_6, x_7, x_8, x_9, x_{10}$ to constant values. Thus the length of the shift registers and the maximum degrees of the filter polynomials corresponding to the five variables that cause S to become linear are relevant for the complexity of this attack. We obtain the complexities $O(2^{139})$ and $O(2^{176})$ for reduced and full Achterbahn-Version 2, respectively.

The attack described in [7, Sec. 5] is a distinguishing attack, which exploits the fact that $R(x_1, \ldots, x_8)$ can be approximated by a linear function of eight variables containing five nonzero terms with probability 3/4. The attack requires the examination of 2^{64} keystream bits. The Boolean function $S(x_1, \ldots, x_{10})$ can at best be approximated by a linear function containing six nonzero terms and with probability 9/16. It follows that in order to detect the bias, $O(2^{384})$ keystream bits are necessary. As the keystream ζ of Achterbahn-Version 2 has least period $< 2^{255}$, the attack does not make sense.

The attack described in [7, Sec. 5.3] and [7, Sec. 7] is the most threatening attack in [7]. In Section 5.3, the function $R(x_1, \ldots, x_8)$ is attacked. Function R agrees with

$$L(x_1, \ldots, x_8) = x_1 + x_2 + x_3 + x_4 + x_6 \tag{10}$$

with probability $p = \frac{3}{4} = \frac{1}{2}(1+\frac{1}{2}) = \frac{1}{2}(1+\epsilon)$. The attacker guesses the first register. This step has complexity $O(2^{22})$. By guessing the first register, the approximation in (10) reduces from five to four nonzero terms. Consider the polynomial

$$g(x) = (x^{P_2} - 1)(x^{P_3} - 1)(x^{P_4} - 1)(x^{P_6} - 1).$$

The sequence $\tau = g(T)\zeta$ is the sum if 16 shifted versions of ζ. The bias for the sequence τ therefore is

$$\epsilon^{16} = \left(\frac{1}{2}\right)^{16} = 2^{-16}.$$

To take advantage of the bias one has to examine 2^{32} keystream bits. Altogether, the complexity of the attack is $2^{22} \cdot 2^{32} = 2^{54}$ for reduced and 2^{60} for full Achterbahn-Version 1.

The same method is used to attack R'' in [7, Sec. 7]. The time complexities of the attack against Achterbahn-Version 1 with R'' are $O(2^{70})$ for the reduced, and $O(2^{76})$ for the full version.

If we apply the attack to Achterbahn-Version 2, we observe that the best linear approximation to S has six nonzero terms and agrees with S with probability 9/16. This yields the complexity $O(2^{211})$, respectively $O(2^{214})$ if the attacker guesses the first register. A better strategy is to guess the contents of the first two registers. This attack has complexity $O(2^{137})$ for reduced and $O(2^{143})$ for full Achterbahn-Version 2. The best strategy consists in guessing the first three registers, which yields complexities $O(2^{112})$ and $O(2^{121})$.

The attack described in [7, Sec. 6.1] against R' takes advantage of the fact that R' contains the first four variables only linearly. The other four variables appear in the nonlinear part of R'. These four variables correspond to the last four shift registers which together can store 115 bits. A TMO-attack is described with time complexity $2^{57.5}$ requiring $2^{57.5}$ keystream bits.

The Boolean combining function S in Achterbahn-Version-2 does not depend linearly of any of its ten variables. Thus the nonlinear part of S coincides with the entire internal state of the KSG which has 262 bits. The complexity of the above attack is comparable with the complexity of a classical TM0-attack which here has time and data complexity 2^{131}.

The attack described in [7, Sec. 6.2] against full Achterbahn-Version 1 makes use of the fact that the function R' reduces to the affine function $L = x_1 + x_2 + x_3 + x_4 + x_7 + 1$ if the variables x_5 and x_6 are both set to 1. The attack requires some more keystream bits (approximately 2^{45}) than the attack described in [7, Sec. 4]. Otherwise the attacks are identical. The time complexity of the attack is $O(2^{73})$, since the lengths of the shift registers corresponding to variables x_5 and x_6 are 27 and 28. The maximum degrees of the corresponding filter polynomials are 8 and 9, respectively. This yields $27 + 28 + 8 + 9 + 1 = 73$, the exponent in the complexity estimation. The same attack applied to Achterbahn-Version 2 has time complexity $O(2^{176})$.

3.4 Quadratic approximations

Quadratic approximation attacks seem to be more threatening to our stream cipher than correlation attacks based on linear approximations. To estimate the threat, we have to consider all quadratic functions of ten variables which have a nonzero correlation coefficient with $S(x_1, \ldots, x_{10})$. The most threatening approximation is given by the quadratic function

$$Q(x_1, \ldots, x_{10}) = x_1 + x_2 + x_3x_4 + x_6x_{10}, \tag{11}$$

which agrees with S with probability

$$\frac{33}{64} = \frac{1}{2}\left(1 + \frac{1}{32}\right) = \frac{1}{2}(1 + \epsilon).$$

If we guess the first two registers of lengths $N_1 = 19$ and $N_2 = 22$, we have only two summands left in (11). The bias of the appropriately filtered keystream sequence is $\epsilon^4 = 2^{-20}$, so that 2^{40} keystream bits must be processed in order to confirm the guess. The overall complexity of the attack is $2^{19} \cdot 2^{22} \cdot 2^{40} = 2^{81}$, still above the complexity of exhaustive key search.

3.5 Cubic approximations

The most threatening cubic approximation is given by

$$C(x_1, \ldots, x_{10}) = x_4 + x_6x_9 + x_1x_2x_3, \tag{12}$$

which agrees with S with probability

$$\frac{63}{128} = \frac{1}{2}\left(1 - \frac{1}{64}\right) = \frac{1}{2}(1 + \epsilon).$$

We guess the the content of the fourth shift register, whose length is $N_4 = 25$. The terms of the sequence $\tau = g(T)\zeta$, where

$$g(x) = (x^{P_6P_9} - 1)(x^{P_1P_2P_3} - 1),$$

are biased with $\epsilon^4 = 2^{-24}$. Thus the time complexity to determine the contents of fourth shift register is $O(2^{73})$ and below the complexity of exhaustive key search. The degree of the polynomial g in (12) is greater than 2^{63}. The attacker needs more than 2^{63} keystream bits in order to run the attack. We counter such an attack by restricting the maximum frame length for our stream cipher to 2^{63} bits.

Tweak: The maximum length of a frame that can be used in the encryption process for Achterbahn-Version 2 is 2^{63} bits.

4 Hardware tweaks

In this section we show how the feedback logics of the driving NLFSRs can be improved with regard to their hardware efficiencies. The goals are:

— to reduce the gate count;

— to increase the frequency at which Achterbahn can be operated.

Both goals can be achieved without sacrificing security.

In the following, the design size is given in gate equivalents. One gate equivalent (GE) is the design size of a 2-input NAND gate. The reported figures have been derived from a synthesis of Achterbahn using high level description language VHDL and mapping the design on 130 nm CMOS standard cell library.

The design size of the KSG can be divided into the following four parts (compare 2):

1. The memory cells including one multiplexor per memory cell for the parallel key-loading.
2. The feedback logics of the ten NLFSRs.
3. The logic that implements the Boolean combining function.
4. The control logic.

How can we save hardware? We cannot shorten the lengths of the shift registers or use a sparser Boolean combining function without lowering the security level, nor can we reduce the control logic. However, there is room for savings in the circuits that implement the feedback functions of the shift registers.

4.1 Reducing the implementation costs of the feedback functions

In this section we describe a way how the implementation costs of the feedback functions can be reduced and at the same time the clock rates for the shift registers increased. The average design size of the feedback functions of the eight driving NLFSRs in the initial proposal of Achterbahn was 42.75 GE. This average value can be reduced to 24.7 GE per shift register in Achterbahn-Version 2.

The objective is to reduce the implementation costs of the feedback functions without thinning out their algebraic normal forms. This is important because a very sparse algebraic normal form would increase the required number of warm-up shifts in the last step of the key-loading algorithm and, thereby, extend resynchronization times. Considering that in many applications the resynchronization intervals are relatively short, this would not be acceptable. Besides, a very sparse feedback function provides less resistance against algebraic attacks [1] than a function of moderate sparsity does.

The objective is achieved by choosing primitive NLFSRs whose feedback functions can be implemented using less expensive gates. Also, 3-input gates are more efficient than 2-input gates. Table 1 lists the hardware costs for the implementation of various logical operations.

HW-Tweak: The initial feedback functions of the NLFSRs are replaced by more efficient feedback functions. The new feedback functions can be implemented at approximately half the hardware costs of the old ones and each function has logical depth three.

For the sake of illustration, let us consider the new NLFSR A. Its feedback function is given by

$$\begin{aligned} A(x_0, x_1, \ldots, x_{18}) = \mathrm{XOR}(&\mathrm{XOR}(x_0, x_3, \mathrm{MUX}(x_5, x_1; x_6)), \mathrm{XOR}(x_8, x_{12}, \mathrm{NAND}(x_4, x_7)), \\ &\mathrm{MUX}(\mathrm{NAND}(x_9, x_{11}), \mathrm{MUX}(x_6, x_{10}; x_4); \mathrm{MUX}(x_2, x_{10}; x_9))). \end{aligned}$$

Logical operation	Binary function	Hardware cost
NAND(a, b)	$ab + 1$	1.00 GE
NOR(a, b)	$1 + a + b + ab$	1.00 GE
AND(a, b)	ab	1.25 GE
OR(a, b)	$a + b + ab$	1.25 GE
XOR(a, b)	$a + b$	2.25 GE
NAND(a, b, c)	$abc + 1$	1.25 GE
NOR(a, b, c)	$1 + a + b + c + ab + ac + bc + abc$	1.50 GE
AND(a, b, c)	abc	1.50 GE
OR(a, b, c)	$a + b + c + ab + ac + bc + abc$	1.75 GE
XOR(a, b, c)	$a + b + c$	4.00 GE
MAJ(a, b, c)	$ab + ac + bc$	2.25 GE
MUX$(a, b; c)$	$a + ac + bc$	2.50 GE

Table 1: Hardware costs of logical operations

The algebraic normal form of the feedback function is

$$\begin{aligned} A(x_0, x_1, \ldots, x_{18}) = {} & x_0 + x_2 + x_3 + x_5 + x_8 + x_{12} + x_1x_6 + x_2x_6 + x_2x_9 \\ & + x_4x_7 + x_5x_6 + x_9x_{10} + x_9x_{11} + x_2x_4x_6 + x_2x_4x_{10} \\ & + x_2x_6x_9 + x_4x_9x_{10} + x_6x_9x_{10} + x_9x_{10}x_{11} \\ & + x_2x_4x_6x_9 + x_2x_4x_9x_{10} + x_4x_6x_9x_{10}. \end{aligned}$$

The implementation costs for the feedback function $A(x_0, x_1, \ldots, x_{18})$ are 24 GE. A switching circuit for shift register A is shown in Figure 1. Shift register A has linear complexity $2^{19} - 2$.

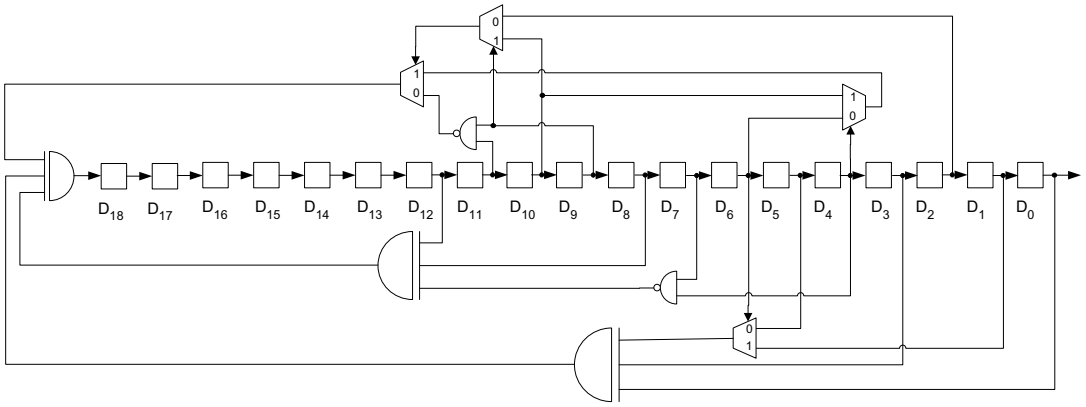


Figure 1: Switching circuit for the new NLFSR A

	Version 1 with DPA protection	**Version 2 with DPA protection**	**Version 2 without DPA protection**
Memory	1002 GE	1245 GE	1245 GE
DPA counter measure	528 GE	655 GE	—
Feedback functions	342 GE	247 GE	247 GE
Combining function	13 GE	63 GE	63 GE
Control logic	288 GE	298 GE	323 GE
Total	**2173 GE**	**2508 GE**	**1878 GE**

Table 2: Design sizes of reduced Achterbahn: Version 1 and Version 2

4.2 Design sizes of parallel implementations of Achterbahn-Version 2

Like the initial NLFSRs of Achterbahn, the new shift registers were chosen in order to facilitate parallel implementations of the KSG. While in a straightforward implementation of the KSG, one bit of keystream is produced per clock cycle, in the parallel implementations two, four, or eight keystream bits are generated per clock cycle. We list the design sizes of the parallel implementations of the KSG for reduced Achterbahn in Table 3. For the sake of comparison, we also list the design sizes of Achterbahn-Version 1. The table contains also the hardware efficiencies of the various implementations. This is the number of keystream bits produced per clock cycle divided by the design size in units of 1000 GE.

Besides the implementations in which countermeasures against the leakage of side channel information are taken (in Table 3 referred to as "Achterbahn with DPA protection"), we also include the design sizes of implementations in which no such counter measures are implemented (in the table referred to as "Achterbahn without DPA protection").

Recall the first part of Achterbahn's key-loading algorithm. In this part all memory cells of the KSG are loaded simultaneously with key bits. The first register, for instance, receives the 19 key bits $k_0, k_1, \ldots, k_{18}$, and the last register, of length 32, the key bits $k_0, k_1, \ldots, k_{31}$. In the next step, the remaining key bits and IV bits are fed serially into the shift registers via an XOR gate in the feedback loop of each shift register. In the third step, the content of one cell of each shift register is overwritten with the bit 1 so that no shift register can be in the all-zero state thereafter. In the last step of the key-loading algorithm, each shift register performs a certain number of warm-up shifts for diffusion purposes.

The intent of the parallel key-loading in step 1 is to avoid the leakage of side channel information in the initialization phase and during resynchronization. Unfortunately, one has to pay a relatively high price in hardware for this feature, to be precise: 655 GE for 262 multiplexors.

In some applications, protection against side channel attacks is not required. For such applications, we can implement the KSG using flip-flops (without reset-capability) which cost 4.75 GE rather than the more expensive scan flip-flops (7.25 GE). The task

of the first step of the key-loading algorithm is now accomplished by inserting the key bits serially into each shift register. Contrary to step 2, in this step no feedback values are added to the introduced key bits. The possibility to disable the feedback logic costs one extra multiplexor per shift register resulting in an increase of the control logic by 25 GE. Thus the total saving amounts to 630 GE. See Table 2.

	Achterbahn-Version 1 with DPA protection		**Achterbahn-Version 2 with DPA protection**		**Achterbahn-Version 2 without DPA protection**	
	Design size	Hardware efficiency	Design size	Hardware efficiency	Design size	Hardware efficiency
1-bit impl.	2173 GE	0.46	2508 GE	0.40	1878 GE	0.53
2-bit impl.	2412 GE	0.83	2820 GE	0.71	2188 GE	0.91
4-bit impl.	3113 GE	1.28	3852 GE	1.04	3274 GE	1.22
8-bit impl.	4778 GE	1.67	4888 GE	1.64	4386 GE	1.82

Table 3: Design size and hardware efficiency of parallel implementations of reduced Achterbahn

5 Conclusion

We reported on the results of our computations concerning the linear complexities of the initial and the new NLFSRs constituting the core of Achterbahn's KSG. We outlined a new probabilistic algorithm for estimating the linear complexities of primitive binary NLFSRs. We described tweaks on Achterbahn-Version 1 as specified in [3] that led to Achterbahn-Version 2. The reported cryptanalytic attacks of Johansson, Meier and Muller [7] were discussed and it was shown that the four attacks described in [7] are either not feasible against Achterbahn-Version 2 or have complexities above the complexity of exhaustive key search. We introduced new feedback functions of the shift registers that are more efficient in hardware. All feedback functions now have logical depth three. Properties of the Boolean combining function S for Achterbahn-Version 2 were discussed. The design sizes and hardware efficiencies for the parallel implementations of reduced Achterbahn were updated.

Acknowledgment: We wish to thank Thomas Johansson for sending us a copy of the preprint [7], which drew our attention to the potential threats arising from the combination of divide and conquer attacks with correlation attacks.

References

[1] N. Courtois and W. Meier: Algebraic attacks on stream ciphers with linear feedback, *Advances in Cryptology – EUROCRYPT 2003* (E. Biham, ed.), Lecture Notes in Computer Science, vol. 2656, pp. 345–359, Springer-Verlag, 2003.

[2] B. M. Gammel and R. Göttfert: Linear filtering of nonlinear shift register sequences, *Proc. of The International Workshop on Coding and Cryptography WCC '2005* (Bergen, Norway, 2005), P. Charpin and Ø. Ytrehus, eds., pp. 117-126.

[3] B. M. Gammel, R. Göttfert, and O. Kniffler: The Achterbahn stream cipher, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/002, 29 April 2005. http://www.ecrypt.eu.org/stream/papers.html

[4] B. M. Gammel, R. Göttfert, and O. Kniffler: Improved Boolean combining functions for Achterbahn, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/072, 14 October 2005. http://www.ecrypt.eu.org/stream/papers.html

[5] R. Göttfert: A probabilistic algorithm to determine the linear complexity of a periodic sequence of period $q^n - 1$, manuscript, Oct. 2005.

[6] T. Johansson, W. Meier, and F. Muller: Cryptanalysis of Achterbahn, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/064, 27 September 2005. http://www.ecrypt.eu.org/stream/papers.html

[7] T. Johansson, W. Meier, and F. Muller: Cryptanalysis of Achterbahn, Preprint, Jan. 2006.

[8] A. H. Chan, R. A. Games, and E. L. Key: On the complexities of de Bruijn sequences, *J. Combin. Theory Ser A* **33**, 233–246 (1982).

[9] J. Dj. Golić: On the linear complexity of functions of periodic GF(q) sequences, *IEEE Trans. Inform. Theory* **35**, 69–75 (1989).

[10] R. A. Rueppel and O. J. Staffelbach: Products of linear recurring sequences with maximum complexity, *IEEE Trans. Inform. Theory* **IT-33**, 124–131 (1987).

[11] E. S. Selmer: *Linear Recurrence Relations over Finite Fields*, Univ. of Bergen, 1966.

[12] T. Siegenthaler: Correlation-immunity of nonlinear combining functions for cryptographic applications, *IEEE Trans. Inform. Theory* **IT-30**, 776–780, 1984.