Securing AES Implementation Against Fault Attacks

Laurie Genelle, Christophe Giraud and Emmanuel Prouff
Oberthur Technologies
{l.genelle, c.giraud, e.prouff}@oberthur.com

Abstract—On smart card environment, speed and memory optimization of cryptographic algorithms are an ongoing pre-occupation. In addition, there is the necessity to protect the device against various attacks. In this paper we present a fault attack detection scheme for the AES using digest values. They are deduced from the mathematical description of each AES individual transformation. The security of our countermeasure is provided in a realistic Fault Model. We moreover show that it can be combined with data masking to obtain an implementation thwarting both FA and DPA. Eventually, implementations of our method are presented, showing that it can be an interesting alternative to the traditional doubling countermeasure method.

Keywords: Fault Attacks, Countermeasure, AES.

I. INTRODUCTION

Fault Attack (FA) is a powerful technique which enables to break unprotected cryptographic schemes very efficiently. The idea is to inject one or several fault(s) during the execution(s) of an implementation and to use the faulty output(s) to obtain information on the secret key stored in the secure component. A particular class of FA called Differential Fault Analysis (DFA) and using statistical treatments focuses on block ciphers such as DES or AES [1], [2]. The most straightforward solution to counteract DFA consists in repeating every sensitive operation twice, then checking some consistency between the results. However this countermeasure, usually called doubling method, does not give full satisfaction. First, it doubles the cost of each sensitive calculation. For some implementation, it may increase considerably the overall cost of the processing, especially if the latter one implements Side Channel Analysis (SCA) countermeasures (such as those presented in [3], [4]). Secondly, the doubling method does not detect attacks injecting the same error twice (which occurs with a non-negligible probability when errors are very localized). In view of the two drawbacks mentioned above, researchers have looked for alternative solutions to thwart FA, paying particular attention to AES. The studies published in the literature [5]–[7] try to solve the FA-protection problem by designing appropriate countermeasures for each of the four main operations (SubBytes, ShiftRows, MixColumns and AddRoundKey) performed in AES. At FDTC 2005, Malkin et al. [6] list the whole combination of security methods based on space/time redundancies. Notably in [5] the authors propose two methods to protect hardware implementations against FA. The first one is based on a hybrid partitioning and the second one uses systematic nonlinear robust codes. In their model, the authors assume that errors are uniformly distributed throughout the circuit. It seems to be unrealistic for software implementation, since as explained in [8], errors affect the circuit in places (faults affecting only one bit are actually in this context much harder to induce than faults affecting bytes). Other countermeasures listed by Malkin et al. use parity bits. As explained by the authors, models using a single parity bit are unrealistic, and for the ones using multiple parity bits, faults of even order may still be not detected with non-negligible probability. A work by Mozzaffari-Kermani et al. [7] proposes a structure-independent fault detection model which can be applied to any hardware implementation of the SubBytes transformation. The authors exhibit a relationship between the input and the output of the S-Box in matrix formulation. Although the suggested test detects every kind of error, its implementation is very costly. The authors therefore propose to transform the test into a single-bit parity test. Even if this method is less costly than the previous one, it is not efficient since the obtained fault coverage is around 50%.

In this paper, we present a redundant fault detection scheme based on digest values computations. This approach runs on detecting faults for every AES transformation by associating it with digest values. We study the security of our scheme in a realistic model and we compute fault detection rates. Furthermore, we compare the efficiency of our countermeasures with the doubling method.

In the next section, we recall the description of the AES algorithm and we introduce the notations used throughout this paper. We also fix the Fault Attack Model. Then, we describe in Section III our countermeasure and we propose several appropriate digest values for each AES transformation. In Section IV, we analyze both the security and the efficiency of our proposal when combined with DPA countermeasures. In Section V, we implement our method on a smart card and we compare its efficiency with the doubling method.

II. PRELIMINARIES

This paper deals with the FA protection of AES software implementations. We briefly introduce hereafter the AES algorithm. Then, we specify the kind of faults we want to detect and we discuss the pertinence of our model.
A. AES Algorithm

1) General Description: The Advanced Encryption Standard (AES) [9] is a 128-bit block cipher which can involve three different master key lengths (128, 192 or 256 bits) depending on the required security level. It is performed by iterating the same round transformation several times, the number of iterations being defined according to the master key length. Each round is parameterized with a unique 128-bit round-key derived from the master key by a transformation called KeyScheduling.

To transform the plaintext into a ciphertext, the AES round transformations operate on a 4 × 4 array of bytes, termed the State.

2) Notations: In the rest of this paper, we denote by \( x \) a 16-tuple \((x_0, x_1, \ldots, x_{15})\) inGF(2)\(^8\)\(^{16}\). When involved in linear algebra operations, it is represented as a 4 × 4 matrix:

\[
x = \begin{bmatrix}
x_0 & x_4 & x_8 & x_{12} \\
x_1 & x_5 & x_9 & x_{13} \\
x_2 & x_6 & x_{10} & x_{14} \\
x_3 & x_7 & x_{11} & x_{15}
\end{bmatrix}.
\]

After denoting by \( x_{C_i} \) (resp. \( x_{R_i} \)) the \( i \)th column (resp. the \( i \)th row) of \( x \), we also use the following representation of \( x \):

\[
x = \begin{bmatrix}
x_{C0} & x_{C1} & x_{C2} & x_{C3}
\end{bmatrix} = \begin{bmatrix}
x_{R0} \\
x_{R1} \\
x_{R2} \\
x_{R3}
\end{bmatrix}.
\]

Eventually, we denote by \( s \) (resp. by \( s' \)) the input state (resp. the output state) of every individual transformation composing an AES round. We give their definition in the following subsection.

3) A Round: The round function is composed of four transformations: AddRoundKey, SubBytes, ShiftRows and MixColumns\(^1\).

The AddRoundKey transformation performs a bitwise addition between the State and the round-key.

The SubBytes transformation is a byte substitution that operates independently on each byte \( s_i \) of the State using a so-called S-Box function. The S-Box is a combination of two transformations: a Multiplicative Inverse Transformation (MIT) and an Affine Transformation (AT).

1. MIT is defined over GF(2)\(^8\) by:

\[
\text{MIT} : s_i \mapsto \begin{cases} 
0 & \text{if } s_i = 0, \\
1 & \text{if } s_i = 0, \\
s_i^{-1} & \text{otherwise}.
\end{cases}
\]

2. AT is defined over GF(2)\(^8\) by:

\[
\text{AT} : s_i \mapsto ps_i + c,
\]

where \( s_i \) is represented as a polynomial over GF(2), where \( p \) and \( c \) equal \( X^4 + X^3 + X^2 + X + 1 \) and \( X^6 + X^5 + X + 1 \) respectively, and where the product is computed in the polynomial field GF(2)[X]/(X\(^8\) + 1).

The software implementation of the SubBytes transformation depends on the countermeasures to set up. In the following, we assume that the SubBytes is performed in two steps: for every element \( s_i \in \text{GF}(2^8) \), the image of \( s_i \) through the S-Box is equal to AT \( \circ \) MIT\((s_i)\).

In the ShiftRows transformation, the last three rows of the State are cyclically shifted on the left over 1, 2 and 3 positions respectively.

The MixColumns transformation operates on the State column by column treating each of them as a four-term polynomial defined over GF(2)\(^8\)[X]/(X\(^4\) + 1), with elements of GF(2)\(^8\) being represented in GF(2)[X]/(X\(^8\) + X\(^4\) + X\(^3\) + X + 1). The MixColumns transformation can be written as a matrix multiplication:

\[
s' = M \cdot s
\]

with

\[
M = \begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}.
\]

B. Fault Attack Model

Many DFA published over the last decade assume that only a small number of State coordinates are disturbed since the attack complexity increases with the number of erroneous bytes. Therefore it seems pertinent to first thwart attacks disturbing at most one byte, then attacks disturbing at most two bytes, and so on, until focusing on attacks disturbing at most \( d \) bytes, where \( d \) is a security parameter. To analyze the security of our countermeasure, we assume that the output \( s' \) of an individual transformation is disturbed by an error vector \( \varepsilon = (\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{15}) \). We call \( d \)-byte Fault Model a model in which \( \varepsilon \) contains at most \( d \) non-zero coordinates. The errors \( \varepsilon_i \) are assumed to be independent and uniformly distributed over GF(2)\(^8\)*. In a software context, we think for the arguments given in [10], [11] that such a byte oriented model is more realistic than a bit Fault Model. Additionally, we assume that the attacker can choose the position of the byte(s) to disturb, but cannot avoid the execution of a small part of the algorithm (techniques exist to counteract such attacks [12], [13]). In the following, we denote by \( \rightsquigarrow \) the perturbation further to a fault injected during a computation.

The security provided by our countermeasures will be analyzed in the model above. Additionally, we will sometimes also consider faults disturbing both the output bytes and the digest values, since it can be of practical interest depending on the implementation.
III. OUR COUNTERMEASURE

In this section, we first give the outlines of our method. Then, we describe each step of the protection in details and we discuss its pertinence with regards to the fault model presented in the previous section.

A. Principle

To protect the AES against external disturbances exploitable by DFA, we choose a redundant fault detection scheme. Our method protects each operation independently and specifies how to combine the local protections to protect the AES globally. Our divide-and-conquer strategy allows us to deduce from the mathematical description of each operation the most appropriate digest value (e.g. additive for linear calculus and multiplicative for power functions). The principle of our method is first to associate the initial input. Then, after the transformation has been performed, the digest value of the output is computed from the digest value of the input. Therefore, before each AES internal transformation the expected digest value of the State with a digest value and then to update the non-zero coordinates of the State. We eventually formally prove that this strategy is sound to detect attacks disturbing one byte, two bytes and 16-byte Fault Models.

We present our results for the four main AES transformations. Firstly we introduce our method for the nonlinear part of the AES. Secondly we propose countermeasures for linear round layers. Finally, we show how to protect the KeyScheduling.

B. SubBytes Multiplicative Inverse

Whatever the representation \((R, +, \times)\) of the field \(GF(2^8)\), the power function is an automorphism of the set \((R^*, \times)\). Thus, for every \((x_1, x_2) \in R \times R\) we have:

\[
(x_1 \times x_2)^\alpha = x_1^\alpha \times x_2^\alpha .
\]  

Due to (3), it makes sense to choose a so-called Multiplicative Digest Value (MDV) to detect errors during the processing of power functions, and in particular during the MIT processing. However, as discussed hereafter, the MDV cannot be used alone because of the particular role played by 0 in a multiplication. We will therefore need to slightly modify it. Let us first define the digest value \(MDV(x)\) of a 16-tuple \(x\) by:

\[
MDV(x) = \prod_{i=0}^{15} x_i .
\]  

If no error has been injected during the MIT processing, then the digest value of the output \(s'\) is equal to the multiplicative inverse of the digest value of \(s\), \(i.e.\) the following relation is satisfied:

\[
MIT(MDV(s)) = MDV(s') .
\]  

However, even when only one byte of the MIT output State has been perturbed, checking the condition above does not always allow us to detect the error. Indeed, if at least one of the State coordinates equals zero, then we have \(MIT(MDV(s)) = MDV(s') = 0\) even if one of the non-zero coordinate of \(s'\) is erroneous. To deal with this issue, a solution consists in detecting attacks on zero-coordinates before computing MDV and then to process MDV only on the non-zero coordinates of \(s\) and \(s'\). This is the strategy that we have followed. First we use a method called Zero Test (ZT) to detect errors on the zero-coordinates of the State. Then, we process the multiplicative digest value only on the non-zero coordinates of the State. We eventually formally prove that this strategy is sound to detect attacks disturbing one byte, two bytes and \(d\) bytes for any order \(d\).

1) Zero-Test: Let us focus on the processing of a bijection that lets the element zero unchanged (which is the case of MIT). We present hereafter an efficient way to detect attacks resulting in the transformation of zero (resp. non-zero ones) coordinates into non-zero (resp. zero ones).

\[\text{Description:}\] We denote by \(ZTI\) the function defined from \(GF(2^8)^{16}\) into \(GF(2)^{16}\) such that for every \(x \in GF(2^8)^{16}\),

\[
ZTI(x) = \sum_{i=0}^{15} 2^i \delta_0(x_i) ,
\]  

where \(\delta_0\) equals 1 if \(x_i = 0\) and equals 0 otherwise.

To verify that no perturbation occurred during the MIT processing for the zero-coordinates of \(s\) and \(s'\), we check that the following relation is satisfied:

\[
ZTI(s) \oplus ZTI(s') = 0 .
\]  

\[\text{Efficiency:}\] Computing \(ZTI\) requires 16 additions, 16 1-bit shifts and 16 comparisons with 0. The computation can be done efficiently in both hardware and software.

\[\text{Security Analysis:}\] We have \(ZTI(s) \oplus ZTI(s') = \sum_{i=0}^{15} 2^i(\delta_0(s_i) + \delta_0(s'_i)).\) Let \(j\) denote the index of a byte equals to zero and disturbed during the MIT calculation. Then we have \(\delta_0(s_j) = 1\) and \(\delta_0(s'_j) = 0\). On the other hand, If the computation of \(s_j \neq 0\) is disturbed such that \(s'_j = 0\), then \(\delta_0(s_j) = 0\) and \(\delta_0(s'_j) = 1\). In both cases, (7) is not satisfied and the fault is detected.
2) 1-byte Fault Detection:

Description: In this section, we assume that the Zero Test has been successfully passed and that we compute MDV only for the non-zero coordinates of \(s\) and \(s'\). In this case, (5) can be rewritten as

\[
\text{MIT}(\text{MDV}(\tilde{s})) = \text{MDV}({\tilde{s}}')
\]

(8)

where \(\tilde{s}\) (resp. \(\tilde{s}'\)) corresponds to the vector \(s\) (resp. \(s'\)) after replacing every zero coordinates by 1. This notation will be used throughout the paper.

Efficiency: We compute 2 MDV and 1 MIT to check (8). The MDV itself involves 15 multiplications. By using \(log\) and \(alog\) tables, those multiplications can be efficiently computed (see for instance [14]). On the other hand, applying the doubling method involves 16 MIT. So, if a multiplication in \(GF(2^{25})\) is twice less costly than MIT, then our countermeasure may be more efficient than the doubling method. Such a situation may for instance occur for SCA-secure implementations. Indeed, securing MIT using data masking is costly in terms of timing and memory space, particularly when masks are changed several times during the processing [3]. In Sections IV and V, we compare in details the doubling method and our countermeasure for a DPA-resistant AES implementation.

Security Analysis: If the computation of \(s'_j\) is disturbed by a fault injection, then there exists \(\epsilon_j \in GF(2^{25})^*\) such that \(s'_j \rightarrow s'_j + \epsilon_j\). In this case, we have:

\[
\text{MDV}(s') \rightarrow \text{MDV}(s') + \epsilon_j \prod_{i \neq j} s'_i.
\]

(9)

The perturbation is always detected since neither \(\prod_{i \neq j} s'_i\) nor \(\epsilon_j\) equals zero.

3) 2-byte Fault Detection: If two output bytes of indices \(j\) and \(k\) have been disturbed, then there exist \(\epsilon_j, \epsilon_k \in GF(2^{25})^*\) such that \(s'_j \rightarrow s'_j + \epsilon_j\) and \(s'_k \rightarrow s'_k + \epsilon_k\).

Behavior of our 1-byte Countermeasure Under a 2-byte Fault Attack: Since \(s'_j\) and \(s'_k\) are different from zero, we have:

\[
\text{MDV}(s') \rightarrow \text{MDV}(s') \cdot \frac{(s'_j + \epsilon_j)(s'_k + \epsilon_k)}{s'_j s'_k}.
\]

(10)

Thus the faults are not detected iff the following relation is satisfied:

\[
(s'_j + \epsilon_j)(s'_k + \epsilon_k) = s'_j s'_k.
\]

(11)

In this case, for every pair \((s'_j, s'_k)\), there are 254 pairs \((\epsilon_j, \epsilon_k)\) satisfying (11). So the probability to not detect the attack equals \(\frac{254}{255^2}\).

Description of a 2-byte Fault Countermeasure: To increase the detection probability, we propose to use a second digest value called Generalized Multiplicative Digest Value (GMDV) and defined by:

\[
\text{GMDV}(\tilde{x}) = \prod_{i=0}^{15} \tilde{x}_i^{i+1}.
\]

(12)

If the MIT processing is not disturbed, the following relation is satisfied:

\[
\text{MIT}(\text{GMDV}(\tilde{s})) = \text{GMDV}(\tilde{s}').
\]

(13)

Efficiency: Checking (13) requires to compute 1 MIT and 2 GMDV. The computation of GMDV involves itself 17 table accesses and 15 additions. Thus, checking both (8) and (13) requires 2 MIT, 2 MDV and 2 GMDV. Our countermeasure may be more efficient than the doubling method if 7 MIT calculations are more costly than the computation of the pair (MDV, GMDV).

Security Analysis: \(\tilde{s}'_j\) and \(\tilde{s}'_k\) Erroneous: Let \(j\) and \(k\) be two State indices such that \(j < k\). If two bytes \(\tilde{s}'_j\) and \(\tilde{s}'_k\) of the output have been disturbed, then \(\text{MDV}(s')\) satisfies (10) and \(\text{GMDV}(s')\) satisfies:

\[
\text{GMDV}(s') \rightarrow \text{GMDV}(s') \cdot \frac{(s'_j + \epsilon_j)(s'_k + \epsilon_k)^j}{s'_j s'_k} = 1\]

(16)

since neither \(\tilde{s}'_j\) nor \(\tilde{s}'_k\) are null. If \(\tilde{s}'_j + \epsilon_j = 0\), then the error is always detected. Otherwise, the error is not detected if the order of \(\frac{\tilde{s}'_k + \epsilon_k}{\tilde{s}'_j}\) is a divisor of \((k - j)\).

The number of solutions \(\epsilon_j\) and \(\epsilon_k\) of (15) depends on the indices \(j\) and \(k\) (i.e. on the choice of the bytes to disturb) and on the values \(s'_j\) and \(s'_k\). In the most favorable case for the attacker (i.e. for some pairs of indices \((j, k)\) and some pairs of bytes \((s'_j, s'_k))\), the probability of the attack to succeed in equals \(\frac{14}{255^2}\). However in the majority of cases the probability is null.
Security Analysis: $\tilde{s}'_j$ and One Digest Value Erroneous:
Let us now slightly extend our security analysis to faults disturbing one State coordinate $\tilde{s}'_j$ and one digest value.

We first consider the case when MDV($\tilde{s}'$) has been disturbed in such a way that (8) does not allow us to detect the error on $\tilde{s}'_j$. Since $\tilde{s}'_j$ is different from 0 (otherwise the error is detected by the ZT test), we have:

$$\text{GMDV} (\tilde{s}') \sim \text{GMDV} (\tilde{s}') \cdot \left( \frac{\tilde{s}'_j + \varepsilon_j}{\tilde{s}'_j} \right)^{j+1}. \quad (17)$$

We deduce that the error $\varepsilon_j \in GF(2^8)^*$ is not detected by checking (13) iff $\left( \frac{\tilde{s}'_j + \varepsilon_j}{\tilde{s}'_j} \right)^{j+1}$ equals 1. The same analysis as the previous one can be applied here: if the order of $\frac{\tilde{s}'_j + \varepsilon_j}{\tilde{s}'_j}$ divides $j + 1$, then the error is not detected.

Now, let us consider that the second digest value GMDV($\tilde{s}'$) has been disturbed. The problem is brought down to detect a 1-byte fault with the digest value MDV($\tilde{s}'$). As proved in Section III-B2, this kind of perturbation is always detected.

Finally, let us consider that one byte of the Zero Test digest value ZTI($s'$) has been disturbed. This kind of fault is neither detected by MDV($\tilde{s}'$) nor GMDV($\tilde{s}'$) iff the disturbed byte $s_j$ equals 0 or 1 and is set to 1 or 0 respectively. The probability of such a scenario to occur equals $\frac{256 \times 255}{2^{32}} = 8.323 \times 10^{-8}$.

4) Generalization to the $d$-th Order: Let us denote by GMDV($t$) the following digest value:

$$\text{GMDV} (t) (\bar{x}) = \prod_{i=0}^{15} \tilde{x}_i^{\alpha_i (t)}, \quad (18)$$

where $\alpha_i (t) = (\alpha_0 (t), \ldots, \alpha_{15})$ is a fixed given vector. We assume that GMDV($0$) and GMDV($1$) equal MDV and GMDV respectively. To protect the MIT computation in a $d$-byte Fault Model, we suggest to use the $d$ multiplicative digest values GMDV($0$), ..., GMDV($d-1$) simultaneously. This results in the definition of a system like (15), but with $d$ equations instead of 2. The construction of vectors $\alpha_i (t)$ such that the system admits a very limited number of solutions $\varepsilon = (\varepsilon_0, \ldots, \varepsilon_{15})$ of weight lower than or equal to $d$ and strictly greater than 0 is left for further analysis.

C. SubBytes Affine Transformation

We recall that the AT transformation defined in Section II-A satisfies the following relation for every $(x_1, x_2) \in GF(2)^2 \times GF(2)^2$:

$$\text{AT}(x_1 + x_2) = \text{AT}(x_1) + \text{AT}(x_2) + c. \quad (19)$$

In the following we deduce from (19) different ways to associate the AT transformation with additive digest values.

1) 1-byte Fault Detection:
Description: As a consequence of (19), it makes sense to choose an additive digest value to check the integrity of AT. We thus associate AT with a digest value denoted by ADV and defined by:

$$\text{ADV}(x) = \sum_{i=0}^{15} x_i. \quad (20)$$

A perturbation on AT is detected if the following relation is not satisfied:

$$\text{ADV}(s') = \text{AT}(\text{ADV}(s)) + c. \quad (21)$$

Efficiency: Checking (21) requires 2 ADV, 1 AT and 1 addition. The computation of ADV itself involves 15 additions. On the other hand, the doubling method would require 16 AT. Thus if the cost of $31/15 \approx 2$ additions is lower than the cost of AT, then our countermeasures may be more efficient than the doubling method. It may for instance be the case when AT implements some SCA countermeasures (see Sections IV and V).

Security Analysis: If an error is injected during the computation of $s'_j$ then there exists $\varepsilon_j \in GF(2^8)^*$ such that $s'_j \sim s_j + \varepsilon_j$. In this case, we have

$$\text{ADV}(s') \sim \text{ADV}(s') + \varepsilon_j. \quad (22)$$

Since $\varepsilon_j \neq 0$, the perturbation is always detected in the 1-byte Fault Model by checking (21).

2) 2-byte Fault Detection:
Behavior of Our 1-byte Countermeasure Under a 2-byte Fault Attack: If we assume that two output bytes of indices $j$ and $k$ are erroneous, then we have:

$$\text{ADV}(s') \sim \text{ADV}(s') + \varepsilon_j + \varepsilon_k. \quad (23)$$

The faults are detected iff $\varepsilon_j + \varepsilon_k \neq 0$. So the probability to not detect a 2-byte fault is $\frac{1}{2^{255}}$. The same probability is obtained if ADV($s'$) and one byte of $s'$ are disturbed.

Description of a 2-byte Fault Countermeasure: We consider the following digest value to increase the detection probability:

$$\text{ADV}_\lambda (x) = \sum_{i=0}^{15} \lambda_i x_i \quad (24)$$

where the $\lambda_i$’s are non-zero constant values in $GF(2^8)$, where $\lambda = (\lambda_0, \ldots, \lambda_{15})$ and where the products are computed in $GF(2)[X]/(X^8 + 1)$.

If no error has been injected during the AT computations, then the following relation is satisfied:

$$\text{ADV}_\lambda (s') = \text{ADV}_\mu (s) + c' \quad (25)$$

where the coordinates $\mu_i$ of $\mu$ satisfy $\mu_i = \lambda_i p$ and where $c' = c + \sum_{i=0}^{15} \lambda_i$. 

Efficiency: We compute $1 \text{ADV}_\lambda$, $1 \text{ADV}_\mu$ and 1 addition to check (25) (since $c'$ is a constant). The processing of $\text{ADV}_\lambda$ (and $\text{ADV}_\mu$) itself involves 15 additions and 16 multiplications. Thus our countermeasure may be more efficient than the doubling method if 32 multiplications and 31 additions in $\text{GF}(2^8)$ are less costly than 16 $\text{AT}$.

Security Analysis: $s'_i$ and $s''_k$ Erroneous: Two bytes $s'_i$ and $s''_k$ of the output have been disturbed, then $\text{ADV}(s')$ and $\text{ADV}_\lambda(s'')$ satisfy:

$$\begin{align*}
\text{ADV}(s') & \rightsquigarrow \text{ADV}(s') + \varepsilon_j + \varepsilon_k \\
\text{ADV}_\lambda(s') & \rightsquigarrow \text{ADV}_\lambda(s') + \lambda_j \varepsilon_j + \lambda_k \varepsilon_k
\end{align*}$$

(26)

with $\varepsilon_j, \varepsilon_k \neq 0$.

If we use together $\text{ADV}(s')$ and $\text{ADV}_\lambda(s')$, the errors $\varepsilon_j$ and $\varepsilon_k$ are not detected if they satisfy:

$$\begin{align*}
\varepsilon_j + \varepsilon_k &= 0 \\
\lambda_j \varepsilon_j + \lambda_k \varepsilon_k &= 0
\end{align*}$$

(27)

This system is equivalent to:

$$\varepsilon_j (\lambda_j + \lambda_k) = 0.$$ 

(28)

Therefore, by choosing the coefficient $\lambda_i$’s different from each other, the faults are always detected since $\varepsilon_j \neq 0$.

Security Analysis: $s'_i$ and One Digest Value Erroneous: We extend our security analysis and we consider that faults disturb one State coordinate $s'_i$ and one digest value.

Firstly, let us consider that an error has been injected during the computation of the first digest value $\text{ADV}(s')$. The problem is brought down to detect a 1-byte fault with the digest value $\text{ADV}_\lambda(s')$:

$$\text{ADV}_\lambda(s') \rightsquigarrow \text{ADV}_\lambda(s') + \lambda_j \varepsilon_j$$

(29)

with $\varepsilon_j \neq 0$. The faults are thus detected iff $\lambda_j \neq 0$.

We consider secondly the case when the second digest value $\text{ADV}_\lambda(s')$ has been disturbed. The problem is brought down to detect 1-byte fault with the digest value $\text{ADV}(s')$. As proved in Section III-C1, this kind of perturbation is always detected.

Conclusion: If $\text{ADV}$ and $\text{ADV}_\lambda$ are used conjointly, and if the $\lambda_i$’s are non-zero and pairwise distinct, then an attack on the $\text{AT}$ transformation in the 2-byte Fault Model is always detected.

3) Generalization to the $d$-th Order: Let us denote by $\text{ADV}^{(t)}_{\lambda}$ the following additive digest value:

$$\text{ADV}^{(t)}_{\lambda}(x) = \sum_{i=0}^{15} \lambda_i^{(t)} x_i.$$ 

(30)

Let us assume that $\text{ADV}^{(0)}_{\lambda}$ (resp. $\text{ADV}^{(1)}_{\lambda}$) equals $\text{ADV}$ (resp. $\text{ADV}_\lambda$). For the same reasons as in Section III-B3, we suggest to use $d$ digest values $\text{ADV}^{(0)}_{\lambda}, \ldots, \text{ADV}^{(d-1)}_{\lambda}$ simultaneously to protect the $\text{AT}$ transformation in a $d$-byte Fault Model. This results in the definition of a system like (27), but with $d$ equations instead of 2:

$$H \cdot \varepsilon^t = 0$$

(31)

with

$$H = \begin{bmatrix}
1 & 1 & \cdots & 1 \\
\lambda_0^{(1)} & \lambda_1^{(1)} & \cdots & \lambda_d^{(1)} \\
\lambda_0^{(2)} & \lambda_1^{(2)} & \cdots & \lambda_d^{(2)} \\
\vdots & \vdots & \cdots & \vdots \\
\lambda_0^{(d-1)} & \lambda_1^{(d-1)} & \cdots & \lambda_d^{(d-1)}
\end{bmatrix}$$

It can be noticed that $H$ is the matrix of a linear code on $\text{GF}(2^8)$ (see for instance [?]). Thus, to detect errors of weight smaller than or equal to $d$, coefficients $\lambda_i^{(t)}$ must be chosen such that every family of at most $d$ columns is composed of linearly independent elements. In this case, $H$ is the parity matrix of a $d$ detector linear code (see for instance [?] for references about instructions of such matrices).

D. ShiftRows

Digest value based on commutative operations is invariant for the $\text{ShiftRows}$ transformation, and in particular $\text{ADV}$ and $\text{MDV}$. So we can use either additive or multiplicative digest values to detect errors during the $\text{ShiftRows}$ transformation. Nevertheless, as it is less costly to implement an addition than a multiplication, we choose to associate $\text{ShiftRows}$ with additive digest values.

1) 1-byte Fault Model:

Description: The following relation is satisfied if no error has been injected in the $\text{ShiftRows}$ computation:

$$\text{ADV}(s') = \text{ADV}(s)$$

(32)

with $\text{ADV}$ defined in (20).

Efficiency: Checking (32) requires 2 $\text{ADV}$ that is 30 additions.

Security Analysis: A similar analysis as the one detailed in Section III-C1 can be applied here. The conclusion is that the perturbation is always detected in the 1-byte Fault Model.

2) 2-byte Fault Model: Digest values $\text{ADV}^{(t)}_{\lambda}$ and $\text{GMDV}^{(t)}$ are not invariant for the $\text{ShiftRows}$ transformation since their values change when the coordinates of their input are permuted. We therefore must here consider a new digest value to increase the detection capacity of our scheme.

Description: Let us consider the following Generalized Additive Digest Value (GADV) which has already been proposed in a similar context [6]:

$$\text{GADV}(x) = \sum_{i=0}^{15} x_i^3.$$ 

(33)

If no error has been injected during the $\text{ShiftRows}$ transformation, then the following relation is satisfied:

$$\text{GADV}(s') = \text{GADV}(s)$$

(34)
Efficiency: Checking (34) requires 2 GADV, that is 30 additions and 32 table accesses. The doubling method would involve one ShiftRows operation. Our countermeasure may therefore be more efficient than the doubling method if the cost of ShiftRows is greater than 30 additions and 32 table accesses (which may happen if ShiftRows implements some SCA countermeasures).

Security Analysis: \( s'_j \) and \( s'_k \) Erroneous: If we assume that the two output bytes of indices \( j \) and \( k \) have been disturbed, we have:

\[
\begin{align*}
\text{ADV}(s'_j) &\rightsquigarrow \text{ADV}(s'_j) + \varepsilon_j + \varepsilon_k \\
\text{GADV}(s'_j) &\rightsquigarrow \text{GADV}(s'_j) + s'_j^3 + s'_k^3 + (s'_j + \varepsilon_j)^3 + (s'_k + \varepsilon_k)^3
\end{align*}
\]

where \( \varepsilon_j, \varepsilon_k \neq 0 \).

If we use together the additive digest values ADV and GADV, the fault injection on 2 bytes succeeds in if the errors \( \varepsilon_j \) and \( \varepsilon_k \) in \( GF(2^8)^* \) are such that:

\[
\begin{align*}
\varepsilon_j + \varepsilon_k &= 0 \\
(s'_j + \varepsilon_j)^3 + (s'_k + \varepsilon_k)^3 + s'_j^3 + s'_k^3 &= 0
\end{align*}
\]

We computed that the probability to find such \( \varepsilon_j \) and \( \varepsilon_k \) equals \( \frac{1}{2^{255}} \).

Security Analysis: \( s'_j \) and One Digest Value Erroneous:

Let us extend our security analysis and consider that \( s'_j \) and ADV\((s'_j)\) have been disturbed. Thus we have:

\[
\text{GADV}(s'_j) \rightsquigarrow \text{GADV}(s'_j) + (s'_j + \varepsilon_j)^3 + s'_j^3 .
\]

The error \( \varepsilon_j \in GF(2^8)^* \) is not detected with the second digest value GADV\((s'_j)\) if:

\[
(s'_j + \varepsilon_j)^3 + s'_j^3 = 0 .
\]

The probability of such a scenario to occur equals \( \frac{1}{2^{255}} \).

Now, let us consider that the second digest value GADV\((s'_j)\) has been disturbed. The problem is brought down to detect 1-byte fault with the digest value ADV\((s'_j)\). As proved in Section III-C1, this kind of perturbation is always detected.

3) Generalization to the \( d \)-th Order: Let us denote by GADV\(^{(t)}\) the following additive digest value:

\[
\text{GADV}^{(t)}(x) = \sum_{i=0}^{15} x_i^{2^t+1} .
\]

It can be checked that GADV\(^{(0)}\) and GADV\(^{(1)}\) equal ADV and GADV respectively. For the same reasons as given in Section III-B3, we suggest to use \( d \) digests values GADV\(^{(0)}\), \ldots, GADV\(^{(d-1)}\) simultaneously to protect the ShiftRows transformation in a \( d \)-byte Fault Model. The working-out of such a system is similar to the elaboration of the matrix of a BCH code.

E. MixColumns

Let us notice the following property of the MixColumns transformation:

\begin{proposition}
Let \( M \) be the AES MixColumns matrix. Then the sum of the four coordinates of each column (resp. each row) of \( M \) equals \( \{01\} \).
\end{proposition}

As a consequence of Proposition 1 we have:

\[
\sum_{i=0}^{15} s'_i = \sum_{i=0}^{15} s_i \quad (40)
\]

where \( s \) and \( s' \) respectively correspond to the input and the output of the MixColumns.

We deduce from (40) an efficient way of performing digest values for the MixColumns.

1) 1-byte Fault Detection:

\begin{description}
\item[Description:] As a consequence of (40), it makes sense to choose an additive digest value to check the integrity of the MixColumns transformation. We thus use the same ADV as defined by equation (20).
\end{description}

If no error has been injected during the MixColumns transformation, then the relation

\[
\text{ADV}(s') = \text{ADV}(s) \quad (41)
\]

is satisfied.

\begin{description}
\item[Efficiency:] Checking (41) requires 2 ADV that is 30 additions.
\end{description}

Security Analysis: The same analysis as the one described in Section III-C1 can be applied here. This analysis leads to the fact that the perturbation is always detected in the 1-byte Fault Model by checking (41).

Remark. The way how the MixColumns is processed may render our countermeasure ineffective. We therefore suggest to use the implementation originally proposed by Daemen and Rijmen in [15, §5].

2) 2-byte Fault Detection:

\begin{description}
\item[Behavior of Our 1-byte Countermeasure Under a 2-byte Fault Attack:] For the same argument as presented in the first paragraph of Section III-C2, the probability to not detect a 2-byte fault by using ADV only is \( \frac{1}{2^{255}} \).
\end{description}

\begin{description}
\item[Description of a 2-byte Fault Countermeasure:] To increase the detection probability, let us now consider the second digest value ADV\(_{\lambda}\) as defined in (24).
\end{description}

If no error has been injected during the MixColumns transformation, then the following relation is satisfied:

\[
\text{ADV}_{\lambda}(s') = \text{ADV}_{\lambda}(s) \quad (42)
\]

where \( \lambda_{C_i} = \lambda_{C_i} \cdot M, \forall i \in 0, \ldots, 3 \).

Efficiency: Checking (42) requires 1 ADV\(_{\lambda}\) and 1 ADV\(_{\sigma}\). The cost of ADV\(_{\lambda}\) (and so ADV\(_{\sigma}\)) is already detailed in Section III-C2. If the cost of checking (42) is lower than the one of MixColumns, then our countermeasure is more efficient.
Security Analysis: The same analysis as the one detailed in Section III-C2 can be applied here. This analysis shows that the faults are always detected by choosing pairwisely distinct and non-zero coefficients $\lambda_i$.

3) Generalization to the d-th Order: In view of (42), we can apply the same strategy as the one in Section III-C3 to ensure the integrity of MixColumns.

F. AddRoundKey

Since the AddRoundKey transformation is a bitwise addition, the most natural choice to protect it is to choose additive digest values.

1) 1-byte Fault Detection:

Description: If no error has been injected during the AddRoundKey transformation, then the following relation is satisfied:
\[
ADV(s') = ADV(s) + ADV(RK)
\]

where $ADV$ defined as in (20) and where $RK$ denotes the corresponding round key.

Security Analysis/efficiency: A similar analysis as the one detailed in Section III-C1 can be applied here. The conclusion is that the perturbation is always detected in the 1-byte Fault Model by checking (43).

2) 2-byte Fault Detection:

Behavior of Our 1-byte Countermeasure Under a 2-byte Fault Attack: The probability to not detect 2 faults by using $ADV$ only is $\frac{1}{256}$ (the analysis is already detailed in the first paragraph of Section III-C1).

Description: To increase the detection probability, let us now consider the second digest value $ADV_\lambda$ as defined in (24).

If no error has been injected during the AddRoundKey transformation, then the following relation is satisfied:
\[
ADV_\lambda(s') = ADV_\lambda(s) + ADV_\lambda(RK)
\]

Security Analysis: We can do a similar analysis as the one described in Section III-C2: faults are always detected if coefficients $\lambda_i$ are pairwisely distinct and non-zero.

G. KeyScheduling

The AES KeyScheduling is composed of two steps. First it operates on the fourth column of the previous key state. The elements of this column are shifted and then, they are transformed by calling the SubBytes function. Eventually, a round constant is added to the transformed column, which results in the construction of a new column $C'$. The second step designs the new key state from the previous one by simply replacing each column $C_i$ of the state by the column $C_i + C'$. The first step can be protected in the similar way as the combining of the ShiftRows and SubBytes transformations of the AES State (namely by using the GMDV, $ADV_\lambda^{(l)}$ and $GADV_\lambda^{(l)}$ digest values). The second step being a simple addition, it can be protected in the 1-byte Fault Model by using the $ADV$ value. To protect it in the $d$-byte Fault Model the digest values $ADV_\lambda^{(0)}, \ldots, ADV_\lambda^{(d-1)}$ can be used simultaneously as described in Section III-C3.

H. Summary

We have proved in previous sections that our method offers a perfect resistance in the 1-byte Fault Model. In the 2-byte Fault Model, our method enables to detect every kind of 2-byte errors disturbing either $AT, MixColumns$ or $AddRoundKey$. For the MIT and ShiftRows transformations, the probability that a 2-byte Fault Attack succeeds in are upper bounded by $\frac{14}{256}$ and $\frac{1}{256 \cdot 256}$ respectively. Generalizations of our work to $d$-byte Fault Model have been suggested. For the linear transformations, detecting errors in a $d$-byte Fault Model is brought down to build a system of $d$ equations making use of correcting code matrix. In a 1-byte Fault Model, it can be checked that for the sequence of transformations ($AT, then \ ShiftRows, then MixColumns$), only 1 $ADV$ verification is required to check the integrity of the computations. Thus, if the $ADV$ verification is less costly than re-computing ($AT$, then ShiftRows, then MixColumns), our countermeasure is more efficient than the doubling method.

In the next section, we focus on an AES implementation combining our method and SCA countermeasures using data masking. We exhibit masked versions of relations introduced above that detect errors in the basic implementation.

IV. Application of our Countermeasure to a DPA-Resistant AES

DPA-resistant implementations of AES (and more generally of any block cipher) involves data masking [16]. The way how the mask is introduced and how it is modified throughout the ciphering depends on the block cipher primitives. In fact, like our FA countermeasures, DPA countermeasures based on data masking apply differently according to the property of the operation to protect. For linear operations, dealing with mask propagation and correction is straightforward. At the opposite, no obvious solution exists for nonlinear operations and several techniques have been proposed that greatly increase the memory/timing complexity of the implementation.

In the following, we assume that the input and the output of the AES subfunctions (MIT, $AT, ShiftRows, MixColumns$ and AddRoundKey) are masked additively. Namely, every function $Op$ is implemented such that it takes as input a masked state $(s + m) \in (GF(2^8))^16$ and outputs the masked state $(s' + m') \in (GF(2^8))^16$, where $m$ and $m'$ are randomly generated data called masks. By combining this masking and the countermeasures presented in previous sections, the goal is to get an AES-implementation thwarting both 1O-SCA and FA. In the following we show how to do such a mixing of the two techniques to define an FA-SCA secure implementation of the SubBytes operation. Due
to length constraints we do not describe the way how to combine the FA countermeasures with additive masking for the linear layers of the AES. Regarding the definition of the digest values $ADV$, $ADV_3$ and $GADV$ and due to the properties (e.g. linearity) of the $ShiftRows$, $MixColumns$ and $AddRoundKey$ operations, defining such a combination is less tricky than it does for the $SubBytes$ operation (it is therefore let for an extended version of this paper).

As in Section III-B we split our analysis into two parts, the first one dedicated to the MIT transformation and the second one dedicated to the AT transformation.

A. MIT Transformation

To use the multiplicative digest values $MDV$ and $GMDV$ when data are additively masked, we have to define a way to efficiently compute the product of two elements $s_1$ and $s_2$ of $GF(2^8)$ from their masked representation $(s_1)_{m_1} = s_1 + m_1$ and $(s_2)_{m_2} = s_2 + m_2$, with $m_1$ and $m_2$ in $GF(2^8)$. Moreover, this computation and the processing of $MDV$ and $GMDV$ must be themselves resistant to first order SCA. Our solution is composed of two steps. First, since $MDV$ and $GMDV$ are multiplicative morphisms, we transform the additive masked data into multiplicative masked data. Secondly, we propose a way to compute every multiplication involved in $MDV$ and $GMDV$ without leaking information exploitable in 1O-SCA.

Let us denote by $[s_i]_\beta$ the value $\beta(s_i + \delta_0(s_i))$, with $\beta$ in $GF(2^8)^\ast$. It is the multiplicative masked representation of $s_i$. When applied to the coordinates $s_i$ of a vector $s$ defined over $GF(2^8)$, the resulting vector is denoted by $[s]_\beta$. Let $r$ be a random value in $GF(2^8)$ and let $T$ be a table filled with 256 bytes equal to $r$. To transform every $(s_i)_{m_i}$ into $[s_i]_\beta$ without leaking information about $s_i$, we use the following algorithm:

**Algorithm 1** Masking conversion

**Inputs:** The table $T$, the additive masked value $(s_i)_{m_i}$, the additive mask $m$, the multiplicative mask $\beta$

**Output:** The multiplicative masked value $[s_i]_\beta$

1. $T[m] \leftarrow T[m] \oplus 1$
2. $res \leftarrow \beta \cdot (r \oplus (s_i)_{m_i} \oplus m)$
3. $res \leftarrow res \oplus \beta \cdot T[(s_i)_{m_i}]$
4. $T[m] \leftarrow T[m] \oplus 1$

To insure that the calculation is secure against 1O-DPA, the operation order must be respected.

**Remark.** An alternative algorithm is proposed in Appendix A.

The product of two elements $[s_1]_{\beta_1}$ and $[s_2]_{\beta_2}$ in $GF(2^8)^\ast$ satisfies:

$$[s_1]_{\beta_1}[s_2]_{\beta_2} = \begin{cases} 
\beta_1 \beta_2 s_1 s_2 & \text{if } s_1, s_2 \neq 0 \\
\beta_1 \beta_2 s_1 & \text{if } s_1 \neq 0 \text{ and } s_2 = 0 \\
\beta_1 \beta_2 s_2 & \text{if } s_1 = 0 \text{ and } s_2 \neq 0 \\
\beta_1 \beta_2 & \text{if } s_1, s_2 = 0
\end{cases}$$

Eventually, for every vector $\beta \in GF(2^8)^\ast$ and for every state $s$ we get:

$$MDV([s]_\beta) = \prod_{i=0}^{15} (\beta_i s_i + \delta_0(s_i) \beta_i) = \prod_{i=0}^{15} \beta_i \prod_{i=0}^{15} (s_i + \delta_0(s_i)),$$

that is

$$MDV([s]_\beta) = MDV(\beta)MDV(\tilde{s}),$$

with the notations introduced in Section III-B2.

On the other hand, from (45) and (8), one deduces that the output state $s'$ after the MIT transformations satisfies:

$$MDV([s']_{\beta'}) = MDV(\beta')MIT(MDV(\tilde{s})).$$

From (45) and (46), we conclude that the following relation holds:

$$MIT(MDV([s]_{\beta})) = \alpha \times MDV([s']_{\beta'})$$

where $\alpha$ equals $MIT(MDV(\beta))MIT(MDV(\beta'))$. (47) is a masked version of (8).

In a similar way, we get

$$MIT(GMDV([s]_{\beta})) = \alpha \times GMDV([s']_{\beta'})$$

where $\alpha$ equals $MIT(GMDV(\beta))MIT(GMDV(\beta'))$. (48) is a masked version of (12).

B. AT Transformation

When applied to an additively masked value $(s_i)_{m_i}$, the AT transformation satisfies:

$$AT((s_i)_{m_i}) = AT(s_i) + AT(m_i) + c.$$  

We denote by $(s)_m = ((s_0)_{m_0}, \ldots, (s_{15})_{m_{15}})$ the masked state and by $(s')_m'$ the masked state after the AT transformations of the masked coordinates $(s_i)_{m_i}$. Due to (49), we have $m' = (m_0, \ldots, m_{15}) + (c, \ldots, c)$. 

**Remark.** The multiplicative masking defined above is a non-reversible masking. Indeed, it is not possible to correct it unambiguously when $s_i$ equals 0 or 1. However, in our context such a property is not damageable. Indeed, we apply the multiplicative masking only to securely compute $MDV$ and $GMDV$ and there is no need to correct it (i.e. to retrieve $s_i$ from $[s_i]_\beta$ for every $s_i$). In the following, we will refer to the operation $s_i, \beta \rightarrow [s_i]_\beta$ as a multiplicative masking of $s_i$. However the latter must not be mixed up with the maskings presented in [14], [17].
It can be checked that the digest values $ADV$ and $ADV_\lambda$ satisfy:

$$ADV((s)_m) = ADV(m) + ADV(s) \quad (50)$$

and

$$ADV_\lambda((s)_m) = ADV_\lambda(m) + ADV_\lambda(s) \quad (51)$$

Of course relations above also hold for $(s')_{m'}$. Thus, from (21) and (50), we deduce:

$$ADV((s')_{m'}) = \lambda T(ADV(s)_m) + c \quad .$$

In the next section, we implement our method on a smart card and we compare its efficiency with the doubling method.

V. IMPLEMENTATION ON A SMART-CARD

In this section, we present the characteristics of AES-128 implementations in order to compare the efficiency of our method with the traditional doubling method. This comparison is done in two cases: the one where the implementation is not protected against DPA attacks and the one where we counteract DPA by using the technique presented in Section IV.

The corresponding algorithms have been implemented on an 8-bit CPU. The timing where obtained with a clock at 30 MHz.

<table>
<thead>
<tr>
<th>Method</th>
<th>Timings (%)</th>
<th>RAM (%)</th>
<th>ROM (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic implementation*</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>PA-resistant AES</td>
<td>Doubling</td>
<td>203</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>This paper</td>
<td>195</td>
<td>140</td>
</tr>
<tr>
<td>PA-resistant AES</td>
<td>Doubling</td>
<td>100</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>This paper</td>
<td>200</td>
<td>133</td>
</tr>
<tr>
<td>DPA-FA-res AES</td>
<td>Doubling</td>
<td>145</td>
<td>548</td>
</tr>
</tbody>
</table>

Table I
COMPARISON BETWEEN THE DOUBLING COUNTERMEASURE AND OUR METHOD TO COUNTERACT FA ON AES.

One can observe that our method is faster than doubling in both cases: in the non-DPA-resistant case we have a gain of 4% and in the other case, we have a gain of 28%.

VI. CONCLUSION

Our scheme associates appropriate digest values to each AES individual transformation, which enables to protect each operation independently. To theoretically analyze the efficiency of our countermeasures in terms of detection capability, we have introduced a fault model called byte Fault Model. As expected, the efficiency of our countermeasures depends on the order of the byte Fault Model. It is perfect in the 1-byte Fault Model and achieves a high detection rate in the 2-byte Fault Model. For orders $d$ greater than 2, we have suggested a way to extend our detection scheme to $d$-byte Fault Model.

The timing/memory cost of our scheme increases with the order of the Fault Model. Our experimental results show that in the 1-byte Fault Model, our method is faster than the doubling method. In this model, $\lambda T$, ShiftRows and MixColumns use the same digest value, thus we only need to test the digest value once to verify the integrity of these three transformations. The improvement is much noticeable for a SCA secure implementation.

ACKNOWLEDGEMENT

We would like to thank Matthieu Rivain for his useful comments.

REFERENCES


2The performances for a basic implementation are 18.5 ms, 30 RAM bytes and 1900 ROM bytes.

3The performances for DPA-resistant AES are 122 ms, 66 RAM bytes and 4100 ROM bytes.
B. A 1B-FA Resistant AES Implementation

Firstly, let us recall the definition of ZTI, MDV and ADV digest values. For \( x \in GF(2^8) \), we have:

\[
\text{MDV}(x) = \prod_{i=0}^{15} x_i ,
\]

\[
\text{ZTI}(x) = \sum_{i=0}^{15} 2^i \delta_0(x_i) ,
\]

\[
\text{ADV}(x) = \sum_{i=0}^{15} x_i .
\]

Algorithm 3 describes a way of implementing an 128-bit AES resistant against one-byte fault attacks. If the function \text{Compare} fails, then the 16-byte temporary value \( s \), the 8-bit digest values \( DV_0, DV_1, DV_2, DV_3 \) and the 16-bit digest values \( DV_4 \) and \( DV_5 \) are erased and the algorithm returns an detection error.

One may note that since \( AT, \text{ShiftRows} \) and \( \text{MixColumns} \) use the same digest value, we only need to test the digest value once to verify the integrity of these three transformations.

Algorithm 3 1B-FA-resistant implementation of the 128-bit AES

\begin{verbatim}
  INPUTS: A 128-bit message \( m \), a 128-bit key \( k \)
  OUTPUT: The 128-bit ciphertext
  1. \( s \leftarrow m \)  \[DV computations to ensure \text{AddRoundKey} integrity\]
  2. \( DV_0 \leftarrow \text{ADV}(s) \)
  3. \( DV_1 \leftarrow \text{ADV}(RK_0) \)
  4. \( s \leftarrow s \oplus RK_0 \)  \[AddRoundKey\]
  5. for \( i \) from 1 to 10 do
    [DV computations to ensure \text{MIT} integrity]
    6. \( DV_4 \leftarrow \text{ZTI}(s) \)
    7. \( DV_5 \leftarrow \text{MIT}(MDV(s)) \)  \[Verification of \text{AddRoundKey} integrity\]
    8. \( DV_2 \leftarrow \text{ADV}(s) \)
    9. Compare \( DV_2 \) with \( DV_0 \oplus DV_1 \)
    10. \( s \leftarrow (\text{MIT}(s_0), \ldots, \text{MIT}(s_{15})) \)  \[MIT\]
    [DV computation to ensure AT, ShiftRows and MixColumns integrity]
    11. \( DV_0 \leftarrow AT(\text{ADV}(s)) + c \)
    [Verification of \text{MIT} integrity]
    12. \( DV_5 \leftarrow \text{ZTI}(s) \)
    13. \( DV_1 \leftarrow MDV(s) \)
    14. Compare \( DV_5 \) with \( DV_1 \)
    15. Compare \( DV_4 \) with \( DV_5 \)
    16. \( s \leftarrow (AT(s_0), \ldots, AT(s_{15})) \)  \[AT\]
    17. \( s \leftarrow SR(s) \)  \[ShiftRows\]
    [Test if last round or not]
  18. if \( i \neq 10 \) then
    19. \( s \leftarrow MC(s) \)  \[MixColumns\]
    [Verification of AT, ShiftRows and MixColumns integrity]
  20. \( DV_1 \leftarrow \text{ADV}(s) \)
  21. Compare \( DV_0 \) with \( DV_1 \)
\end{verbatim}
[DV computations to ensure AddRoundKey integrity]
22. $DV_1 \leftarrow ADV(RK_i)$
23. $s \leftarrow s \oplus RK_i$ [AddRoundKey]
24. else
[Verification of AT and ShiftRows integrity]
25. Compare $DV_0$ with $DV_1$
26. $DV_1 \leftarrow ADV(RK_{10})$ [AddRoundKey]
27. $s \leftarrow s \oplus RK_{10}$
[Verification of AddRoundKey integrity]
28. $DV_2 \leftarrow ADV(s)$
29. Compare $DV_2$ with $DV_0 \oplus DV_1$
30. return $s$