A fast, GPU based, dictionary attack to OpenPGP secret keyrings

Fabrizio Milo\textsuperscript{b,∗}, Massimo Bernaschi\textsuperscript{a}, Mauro Bisson\textsuperscript{b}

\textsuperscript{a}IAC-CNR, Rome, Italy
\textsuperscript{b}Dipartimento di Informatica Università “La Sapienza”, Rome, Italy

\section*{A R T I C L E   I N F O}

Article history:
Received 30 September 2010
Received in revised form 5 April 2011
Accepted 18 May 2011
Available online 14 June 2011

Keywords:
Cryptanalysis
Graphics Processing Units
CUDA
OpenPGP

\section*{A B S T R A C T}

We describe the implementation, based on the Compute Unified Device Architecture (CUDA) for Graphics Processing Units (GPU), of a novel and very effective approach to quickly test passphrases used to protect private keyrings of OpenPGP cryptosystems.

Our combination of algorithm and implementation, reduces the time required to test a set of possible passphrases by three-orders of magnitude with respect to an attack based on the procedure described in the OpenPGP standard and implemented by software packages like GnuPG, and a tenfold speed up if compared to our highly tuned GPU implementation. Our solution can be considered a replacement and an extension of ppcreak, a utility used in the past for attacking PGP.

The optimizations described can be applied to other cryptosystems and confirm that the GPU architecture is also very effective for running applications that make extensive (if not exclusive) use of integer operations.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

The OpenPGP standard combines different technologies to provide confidentiality, key management, authentication and digital signatures services. It is defined by the OpenPGP Working Group in the Internet Engineering Task Force (IETF) Standard RFC 4880 (\textit{Group}, 2007). OpenPGP uses both strong symmetric and public-key cryptography algorithms and its protocol assumes that the private key portion of a public–private key pair is controlled and properly secured by the party or parties. Private keys are encrypted by using a symmetric encryption algorithm whose key is derived from a user defined passphrase and stored in a file called the \textit{secret ring} (\textit{secreting}). A copy of the secreting and the knowledge of the passphrase, by any means achieved, allows a malicious user to execute critical operations such as signature and decryption, on messages and files belonging to the legitimate owner of the public key. The OpenPGP standard does not provide any form of passphrase recovery. The passphrase is the unique and irreplaceable security token allowed by the system.

The attack we present targets the passphrase protecting the private keys as described by the OpenPGP standard. We base our work on GnuPG (\textit{Skala}, 2008), an open source software implementation of OpenPGP. GnuPG is widely used for private communication over the Internet and is considered by its users a solid and reliable software package.

A naive attack would enumerate all possible combinations of characters and test their validity as passphrase. This approach is computationally infeasible. A viable approach is, as with other cryptosystems, to select a set of possible passphrases, thus creating a \textit{dictionary} for the specific target.

Our attack is an \textit{improved} dictionary attack (Bernaschi \textit{et al.}, 2009) that runs on one or more Graphics Processing Units (GPU). The improvement is significant: the time required to check large sets of passphrases is drastically reduced if compared to the standard procedure. Showing that an attack to a crypto system can be much cheaper than expected, is one of the main goals of the present paper. In pursuing our main objective we introduce and describe a set of optimization techniques that can be easily applied to similar applications and can be of common interest. Their description and discussion is the other goal of this paper.

The GPU is, by now, considered much more than a simple graphic accelerator. Moreover, the Common Unified Device Architecture (CUDA) (\textit{Nvidia}, 2001b) proposed by Nvidia makes their programming reasonably simple. However, so far, GPUs have been employed in numerical computing mainly for their \textit{floating point} number crunching capabilities whereas their use for integer calculations and cryptography has been quite limited.

Recent studies, like those presented in Manavski (2007), Nguyen (2007) and NIST (2001) describe implementations of the Advanced Encryption Standard (AES) for GPU. Other papers explored the use of CUDA for the Serpent (Anas Mohd Nazlee and Ali, 2009) and RSA (Harrison and Waldron, 2009) algorithms. Open source imple-
mentations of the Message Digest 5 (MD5)\(^1\) and the Data Encryption Standard (DES) (Che et al., 2008) algorithm, are also available.

2. GnuPG attack

Our attack is carried out in three phases (see Fig. 1). The first phase creates an initial set of candidate passphrases from a user provided list of text files. The second phase expands this initial set of candidates, by applying several transformation rules to each passphrase. The third phase tests every passphrase in the candidate set. Each phase receives as input the output of the preceding one. GnuPG employs the following Standard Authentication Procedure to validate a passphrase.

2.1. GnuPG Standard Authentication Procedure

The private key is encrypted by a symmetric algorithm, which requires, in turn, that a key is generated starting from the user-defined passphrase.

Since a generic string of characters cannot be directly used as a key to any conventional crypto system, the passphrase needs to be transformed. The procedure to transform the input passphrase into the key of the cipher is called string-to-key (S2K) and is computationally very expensive. The S2K procedure for the GnuPG system consists in calculating the SHA1 digest (Jones, 2001) of the input passphrase in an iterated and salted mode [Group, 2007, Section 3.6.1.3].

Part of the obtained digest is then used as the key for the symmetric encryption algorithm. CAST5 (Carlisle Adams, 1997) is the default block cipher\(^2\) used to protect the secret material of the securing including the private keys. If and only if, the right passphrase is used to derive the key, then the CAST5 algorithm produces a valid plaintext of z-bytes in size:

\[
[b_2b_{-1}|b_{-2} \ldots b_{21}|b_{20}b_{19} \ldots b_1]
\]

The plaintext is composed by a Multi Precision Integer (MPI) \((b_2 \ldots b_{21})\) (which is the private key) and by its SHA1 hash \((b_{20}b_{19} \ldots b_1)\).

After the decryption, the system checks if the result of SHA1 applied to the MPI \(b_2b_{-1} \ldots b_{21}\) matches the hash \(b_{20}b_{19} \ldots b_1\), included in the plaintext. If this test succeeds then GnuPG controls the fulfillment of the correct algebraic relationship between the MPI number and the corresponding public key. If this final check is successful then the passphrase under test is the correct one.

2.2. The fast test

The Multiple Precision Integer used in GnuPG consists of two parts. The first part is a two-byte scalar expressing the length of the MPI in bits (starting from the most significant non-zero bit); the second part is a string of bytes representing the actual integer.

We augment the standard verification process with a preliminary step: we test if the left part of the plaintext is a sound MPI (see Fig. 2).

The test verifies that the two-byte scalar \(b_2b_{-1}\) represents the bit length of the string \(b_{-2} \ldots b_{21}\). Since the size \(z\) of the plaintext is known (it equals the size of the ciphertext) we can compute the size, in bits, of the integer field of the MPI as

\[
l_{\text{MPI}} = (z - H_{\text{len}} - m) \times 8 - l_0(b_{-2})
\]

where \(H_{\text{len}}\) is the size of the SHA1 hash (20 bytes), \(m\) the MPI length field (2 bytes), \(8\) is the number of bits in a byte and \(l_0(b_{-2})\) is the number of leading zeroes of the most significant byte of the integer field.

From now on, we will refer to the check:

\[
b_2b_{-1} == l_{\text{MPI}}?
\]

as the fast test. If the fast test succeeds, we continue with the standard procedure (see Section 2.1).

Note that the fast test and the SHA1 hash check may produce false positives. The final test (verifying the algebraic relationship between public and private key) is exact but it is computationally very expensive.

GnuPG does not carry out our first control that is already very selective. Our first control discards around 95% of invalid passphrases. By following this multi-step procedure our validation test is much more cost-effective.

The value of the fast test is not in its mechanism (the comparison between two integer fields) but is in enabling a software to exploit very effectively the architecture and the processing power of the Graphics Processing Unit. An implementation of the whole standard authorization procedure, in terms of GPU technology, including the last step of verifying the private key, executing

\[\text{Fig. 2. Comparison between the standard procedure and our fast test.}\]

\[\text{Fig. 1. The three phases of the attack.}\]
multiprecision integer operations, would be overwhelming, ineffective and slow. Thanks to the fast test it is possible to develop an efficient GPU implementation able to discard a large number of invalid passphrases.

3. GPU acceleration

In recent years the architectures of the Graphics Processing Unit have improved so much that it is viable to develop generic algorithms on them. There are classes of problems that may be expressed as data-parallel computations with high arithmetic intensity where a CPU is not particularly efficient. Multi-core CPUs excel at managing multiple discrete tasks and processing data sequentially, by using loops to handle each element. Instead, the architecture of GPU maps the data to thousands of parallel threads, each handling one element.

This architecture looks ideal for our fast test attack (Section 2.2) as our problem is embarrassingly data parallel, i.e. each passphrase is independently valid or not. The expected increase in execution speed motivated our efforts to develop a GPU version using CUDA.

3.1. CUDA

Programming a complex device like a Graphics Processing Unit is not simple. Many approaches have been proposed over the years to create a suitable programming model (brookGPU (Buck et al., 2004), Sequoia (Fatahalian et al., 2006), Sh (McCool, 2004; Fernando and Kilgard, 2003)). At this time, one of the most widely used solutions is the Compute Unified Device Architecture (CUDA) by Nvidia.

CUDA is a general purpose, scalable and parallel programming model to write highly concurrent applications. It provides several key abstractions including: a hierarchy of thread blocks, a hierarchy of memory and atomic increment primitives. This model has proven successful and effective at programming GPUs with a Single Instruction Multiple Threads (SIMT) paradigm and scales transparently to hundreds of cores.

3.2. Architecture

As already stated, our problem is embarrassingly data parallel. Each passphrase is independently valid or not. For this reason we expect to have a computational bottleneck, rather than a bandwidth one, because the number of operations per passphrase is very high.

We leverage our fast test algorithm by implementing a GPU passphrase filter. It reduces the set of candidate passphrases to a much smaller list, containing only those passphrases that fulfill our fast test.

The GPU filter follows the conventional execution flow of GPU applications (see Fig. 3) and it involves three steps:

(i) load the passphrases buffer into global memory;
(ii) process the data by using the GPU;
(iii) analyzes the results.

To take advantage of the GPU parallelism the procedure needs to collect a large number of possible passphrases, upload them into the GPU memory and process all of them at once.

Our CUDA version of the attack program has limited memory requirements and can run on any recent video card produced by Nvidia. We can increase the number of passphrases processed per second by using the distributed version of our attack (described in Bernaschi et al., 2009).

3.2.1. Initialization

Before starting the execution of the filter we need to setup its environment. The setup phase allocates and initializes in the GPU memory the structures required to work on a set of passphrases, including: the passphrase buffer, the result buffer and the keyring constant data.

3.2.2. Input

For our experiments, we decided to limit the maximum size of a passphrase to sixteen characters, a length that is fast to load on a GPU and that is usually over the average size of human-recallable passphrases.

To the fixed length passphrases, we apply a common attack technique, Password Transformation.\(^3\) The majority of users adopt passwords comprising easy-to-remember lowercase/uppercase letters, usually in English or in their first language, edited with common letter substitutions. This fact suggests to use transformation rules like: conversion to uppercase, conversion from letters to numbers (e.g., from “0” to “0” or from “i” to “1”). Note that such transformations do not change the size of the passphrase but only its charset. Our implementation allows an attacker to generate additional passphrases directly in GPU memory providing a better tuning for a specific attack.

3.2.3. Execution

Each thread fetches and processes only one passphrase. This is the simplest and most effective choice. It avoids complex logic and any kind of synchronization requirement. Each thread, upon completion, reports its outcome in the corresponding field inside the result structure. The unit of work executed by each thread is called a kernel. We developed two independent and complete kernels for the S2K and CAST5 steps and through a process of fine tuning and continuous testing, we merged them into a single and faster kernel. We apply this kernel to the set of independent passphrases stored in the input buffer.

3.2.4. Output

When the last thread terminates, the result structure is copied back to main memory and every marked passphrase is tested with the Standard Authentication Procedure by using the CPU. Note that only the correct passphrase can pass this last test. In case of success, the user is notified with the passphrase and the attack ends. In the

\(^3\) This technique is also used by the famous password cracker John the Ripper.
distributed version of the attack (Bernaschi et al., 2009), the final result is notified back to the root node.

3.3. Implementation details

3.3.1. Input
To store N passphrases, the input buffer is divided into N elements, where each element is of exactly sixteen characters (see Fig. 4) and each element contains one passphrase. The trailing NULL delimiter\(^4\) is maintained only for the strings having fewer than sixteen characters. This convention allows the storage of exactly sixteen characters instead of fifteen plus the NULL delimiter. The only negative impact of having a fixed element length is in the case of short passphrases (i.e., four characters). In this case many of the buffer elements would be almost empty. We sacrifice memory space for speed and logic simplicity. Later in the paper, we will describe one optimization step to address this issue.

3.3.2. Uploading data into the GPU
To process the input dictionary with the GPU, we need to upload it into the global memory of the device. The number of passphrases in the input dictionary can be arbitrarily large and it could exceed the memory resources of the GPU. For this reason we split the input buffer into chunks. All the passphrases of a chunk are processed simultaneously by a single kernel invocation. The application continues to read the input dictionary in chunks until either a passphrase is found or the dictionary is depleted. The optimal chunk size depends on the capabilities of the GPU device as memory resources and number of processors determine how fast it can be transferred into the device memory.

We measured the bandwidth of the GPU by loading different buffers of increasing sizes (see Fig. 5). From the results we can see that to achieve ~99% of theoretical throughput we need, at least, a chunk of 4 MB in size.

3.4. Kernels

The arguments of each kernel are: the passphrase buffer, the passphrase length and the result buffer. The only other required values are from the keyring constant data. The keyring constant data contain the salt and byte count for the S2K step, the Initialization Vector (IV) used by CAST5 and the ciphertext the target of our attack. All of these additional values are copied to the GPU constant memory\(^5\) during the initialization step (see Section 3.2.1).

The ID of each passphrase is its index position inside the current chunk. We spawn a number of threads equal or larger than the number of passphrases. Each thread selects the input passphrase using the formula:

\[
pass_id = blockIdx \times blockDim + threadIdx
\]

\(^4\) Is a common convention in the C-Language to end a string of characters with a NULL delimiter.

\(^5\) The constant memory is a special read-only memory, very fast and directly referenceable by all the kernels.

where blockIdx is the ID of the thread block, blockDim is the block size and threadIdx is the ID of the thread within the block.\(^6\) If the thread ID is greater than the total number of passphrases, then it terminates.

3.4.1. S2K kernel
The S2K (string-to-key) mechanism requires generating the SHA1 digest of the passphrases in iterated and salted mode. After generating a byte sequence by repeatedly concatenating the salt with the passphrase, a digest of the byte sequence is produced by using the SHA1 algorithm (see Fig. 6).

Iterating and salting the passphrase is a security precaution commonly adopted by many modern cryptosystems. Iterating many times increases the testing time of the single passphrase in a way that is unnoticeable for an authorized user but that, for an attacker, could mean orders of magnitude of additional work. Salting the passphrase prevents precomputed rainbow tables attacks (Oechslin, 2003).

The salt and the byte count (usually \(2^{16}\) bytes) used to generate the byte sequence are both specified inside the secring (see Fig. 7). Allocating a buffer in local memory to store the whole byte sequence for each running thread is too expensive for the limited resources of the GPU. The solution is to store only enough space to handle the data required by a single round of the SHA1 algorithm.

\(^6\) See Section 4.3.1 for further details on the meaning of these variables.
Upon completion of all the rounds the digest is ready for the block cipher.

3.4.2. CAST5 kernel

The CAST5 procedure is divided in two steps. The first step transforms the input key by using a set of lookup tables. These tables are loaded inside the constant memory of the device before launching the kernel. Although the constant memory is limited, the tables are small enough to fit inside it. Each thread loads, from device memory, the key (produced by the previous S2K kernel) corresponding to its own global ID.

The second step reads and decrypts a CAST5 block-size (128 bytes) amount of encrypted data from the constant memory. If the length of the ciphertext is greater than that of the cipher block size, it requires multiple iterations to be entirely decrypted.

Once the first three bytes (i.e. the first part of the MPI number) are decrypted, we have all the data to execute the fast test. The procedure completes storing the result in global memory.

3.5. Output

After all the threads terminate, the result buffer is copied back in main memory and analyzed by using the CPU. The result structure is simply an array with one element for each analyzed passphrase. Each element contains the index of the passphrase in the chunk, if the fast test was successful or a negative integer (−1) otherwise. When a valid index is found, the CPU looks up the corresponding passphrase and tests it with the last two steps of the Standard Authentication Procedure (see Section 2.1).

The result buffer contains both valid and invalid ID. Reducing it to valid indexes only (for example by using the classic parallel technique of prefix-sum; Blelloch, 1990), does not offer any significant advantage.

4. Optimization phases

This section describes the various techniques we experimented with, to improve the execution speed on the GPU. The metric used is: passphrases tested per second. Due to the specific architecture and programming paradigm, different approaches have been tested in order to reach the best performance. Our main reference in this study has been the excellent CUDA Best Practices document provided by Nvidia (2001a).

We started by measuring the time required by a CPU to process a dictionary of one million randomly generated passphrases between five and sixteen characters long, not augmented with transforma-

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU version 1: loop unroll.</td>
</tr>
<tr>
<td>Version</td>
</tr>
<tr>
<td>CPU – single thread</td>
</tr>
<tr>
<td>GPU v.1 – loop unrolling</td>
</tr>
</tbody>
</table>

tion rules. The test performed on a Intel Xeon E5462 CPU (2.8 GHz, 6 MBytes of cache) by running a single thread resulted in a time of 505 s and has been used as reference.

Instead of launching the SHA1 kernel, storing its results in memory, launching the CAST5 kernel, fetching the results from SHA1 and computing the final result, we merged the two kernels into a single one, avoiding $2 \times N$ additional accesses to global memory, where $N$ is the number of passphrases processed in a chunk. This is feasible because the global memory of the device is not reset at each kernel invocation, whereas the local memory and the registers are.

The merged kernel performs the following steps:

1. Loads the constant information.
2. Loads the passphrase from global memory.
3. Computes the S2K.
4. Applies the CAST5 cipher to it.
5. Performs the fast test.
6. Saves the result of the test in global memory.

Of all the steps performed by the kernel, the S2K is the most expensive and thus is the main focus of our optimizations.

4.1. String-to-key optimization

To gain more insight into the mechanism of the byte-shuffling operations performed by the S2K procedure, we analyzed the device code generated by the GPU compiler.

The first thing that we noticed has been the large number of register spilling operations. The main kernel optimizations implemented to mitigate this behavior have been the loop unwinding of the SHA1 round logic and the preprocessing of the data access pattern for the iterated and salted buffer (see Section 3.4.1).

4.1.1. Loop unwinding

Loop unwinding (or loop unrolling) is a classic compiler optimization technique (Aho et al., 2006). Its goal is to increase the execution speed of a program by reducing or eliminating the instructions required to control the cycling logic, such as pointers arithmetic and exit tests, that occur on each iteration of the loop. Most loops can be re-written as a repeated sequence of equivalent and independent statements, thus eliminating this overhead and improving the data parallelism among register operations.

The SHA1 algorithm has different stages, each containing a series of loops. With the aid of a Python (van Rossum, 2011) program, we generate a CUDA-language source code of the unrolled version.

With this optimization the kernel becomes 20× times faster (see Table 1). It takes only 28 s to process our test case containing one million passphrases vs the CPU reference time of 505 s.

We think that the obtained speedup is not only due to the additional free registers but also due to the emerging instruction-level parallelism between the unrolled statements. Many of the SHA1 inner operations are independent from each other, allowing the threads to fill the hardware pipeline in a very effective way.

4.1.2. Improved iterated and salted algorithm

Our target hash is the result of SHA1 on the entire iterated and salted buffer (see Section 3.4.1). Storing the whole buffer would require a significant amount of memory ($2^{16}$ bytes for each thread). The solution is to store only the intermediate block required by a
single round (512 bit) and fill it iteratively with the salt and the passphrase.

Note that the bytes composing the input buffer of the S2K procedure are always the same, i.e. the salt and the passphrase. This suggests to create an even smaller buffer than the one needed by a single round, containing only the salt and the passphrase.

To clearly comprehend this optimization, it is necessary to further describe the mechanism of a SHA1 round. Each round requires exactly sixteen input values, each of four bytes, for a total of 512 bit of data (4 bytes × 16 = 64 bytes = 512 bit). Assuming that the byte sequence is 216 bytes long, in each round is possible to pre-compute which combination of bytes from the salt and passphrase buffer will compose the next sixteen input values. For each passphrase we will have 1024 iterations (65,536/64). We extend the reduced buffer with an additional 4 bytes, taken from the first part of the salt, to align the buffer, and treat it as a circular buffer. In this way is easy to extract the required sixteen values for one round of SHA1 x00, x01, ..., x15 and then execute a SHA1 round (see Code 1).

By following this approach, we achieved a 22× speed up over the original implementation with a speed up of ~1.1× over the previous kernel (see Table 2). This change does not have a large impact on the overall performances but it plays an important role when combined with another optimization technique described later on: kernel specialization (see Section 4.5.3).

**Theorem 1.** C++ code of the improved S2K logic.

```cpp
unsigned int shift = SALT_SIZE + passphrase_len;
for (int i = 0; i < 1024; i++) {
    int offset = (i * 64);
    // compute input value 1/16
    start = 0 * 4 + offset;
    start = start + shift * (start / shift);
    x00 = (buff + start)[0] + 24
    (buff + start)[1] = 16;
    (buff + start)[2] = 8;
    (buff + start)[3];
    // compute input value 2/16
    start = 1 * 4 + offset;
    start = start + shift * (start / shift);
    x01 = (buff + start)[0] + 24;
    (buff + start)[1] = 16;
    (buff + start)[2] = 8;
    (buff + start)[3];
    ...
    // compute input value 16/16
    start = 15 * 4 + offset;
    start = start + shift * (start / shift);
    x15 = (buff + start)[0] + 24;
    (buff + start)[1] = 16;
    (buff + start)[2] = 8;
    (buff + start)[3];
    // execute one SHA1 round
    SHA1(x00, x01, x02,...,x15);
}
```

4.2. Memory access optimizations

4.2.1. Data transfer between host and device

One of the issues that need to be considered during the development of a GPU application is the limited data bandwidth available between the CPU main memory and the GPU global memory. We tried different configurations of data transfers provided by the CUDA API: (i) SimpleMalloc Pageable Memory, (ii) Pinned Memory, (iii) Write Combined and (iv) Mapped Write Combined. The best performances have been obtained by utilizing the Pinned Memory with only the Write Combined flag enabled (with a speed up of 4× if compared to the Simple Malloc Pageable Memory) (see Fig. 8). Enabling the Mapped flag (which delegates the responsibility of memory transfers to the hardware driver), did not provide any additional benefit and occasionally resulted in even lower performances.

4.2.2. Coalesced accesses

After the data are stored in global memory, each thread needs to issue instructions to fetch it. The address alignment and the size of these requests can affect performances significantly.

The GPU scheduler divides the thread pool in warps, groups of thirty-two threads. All the memory requests from threads belonging to the same half-warp (i.e. sixteen threads) are collected and processed at once. The global memory of the GPU is divided into segments of 128 bytes. If all the threads of a half-warp request a global memory address belonging to the same memory segment, then all the requests are coalesced into one and served by executing only a single memory transaction. Otherwise, the set of requests is uncoalesced and needs to be serialized resulting in multiple (thus slower) memory transactions.

The simplest way to fetch the passphrases from global memory is by requesting one character at time. However, given our data layout, this access pattern results in a severe performance drop because only eight threads out of sixteen access the same segment, generating a high number of uncoalesced accesses (see Fig. 9).

**Access Metrics.** We used the excellent CUDA profiler tool to measure the total number of memory transactions issued by the kernel.

We built a small test that runs on a grid, composed by a single block of 32 threads. Each thread loops over all the 16 characters. On a device with compute capability 1.1, the profiler reports 512 uncoalesced accesses or, in other words, for each of the thirty-two threads, each kernel does a serial request each time (32 × 16 = 512).

![Fig. 8. Different speed of Pinned Memory vs Pageable Memory on a C1060.](image)

![Fig. 9. Uncoalesced access of a half warp. The global memory is composed of elements of 16 bytes. Eight threads access 8 × 16 bytes = 128 bytes. Sixteen threads (half warp) access 2 × 128 bytes or two segments.](image)
A first possibility to lower this value is to fetch four characters at once instead of one. Another important factor is the total number of requests made by each kernel even if the memory transactions are coalesced. If we would request all the 16 characters at once, as a single vector of four unsigned integers, we could achieve a coalesced access and reduce the number of fetches to only four, resulting in an optimal memory transaction. We analyzed the resulting binary with decuda, a third-party disassembler. This tool produces, from a CUDA compiled binary file, the native instruction set instead of the abstract PTX [Nvidia, 2001c]. In this way we are sure that the kernel actually issues a 128 bit (16 chars × 8 bit) request at once with a single instruction

```plaintext
mov.b128 r0, global[r0]
```

This is the maximum memory transaction size that can be requested by a single instruction.

By optimizing the global memory access and lowering the number of segment transfers, we reached a speed up of 63× over the original implementation and of 3× over the previous kernel (see Table 3).

### 4.3. Execution optimizations

#### 4.3.1. Grid size and kernel occupancy

The GPU threads can be scheduled in different layouts, allowing us to choose the best one according to the data parallelism of the problem. This layout is specified through a set of values that define the grid. Threads are organized in blocks. The number of threads per block is called the block size, whereas the number of blocks is called grid size. The total number of threads is equal to block size × grid size. It is possible to shape both the block and the grid in different ways without changing the total number of running threads. A grid is fully specified by a block size, a block shape, a grid size and a grid shape. The GPU hardware is organized as a set of multiprocessors and each handles up to eight blocks of threads.

The occupancy value indicates the percentage, per multiprocessor, of resident active threads with respect to the maximum possible number, that is 1536 on the latest generation NVIDIA cards.

The optimal size of the grid depends on both the number of multiprocessors available on the target device and the amount of resources (number of registers, shared memory) used by the active kernel.

The Nvidia documentation recommends at least 192 threads per block. Our experiments reveal that for our arithmetic intensive kernels, it is better to specify 64 threads per block with a number of blocks equal to four times the number of multiprocessors available on the GPU.

#### 4.4. Thread branch divergence

In the GPU Single Instruction Multiple Threads architecture, each thread has its own instruction address counter and register state. Therefore each thread is free to branch and execute independently from the others.

### Table 3

<table>
<thead>
<tr>
<th>GPU version 3: coalesced global memory accesses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>CPU – single thread</td>
</tr>
<tr>
<td>GPU v.2 – reduced buffer</td>
</tr>
<tr>
<td>GPU v.3 – coalesced accesses</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>Thread divergence with different passphrase length inputs and time required by our application for processing the corresponding dictionary layout.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passphrase length</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td>Fixed 8</td>
</tr>
<tr>
<td>Fixed 12</td>
</tr>
<tr>
<td>Fixed 16</td>
</tr>
<tr>
<td>Mixed 8, 12, 16</td>
</tr>
<tr>
<td>Mixed 1–16</td>
</tr>
</tbody>
</table>

Each thread of a warp that executes different logic paths, due to a data-dependent conditional branch, is said to be divergent. Branch divergence occurs only within a warp. In the presence of divergent threads, the warp executes each branch serially. The warp continues to execute in a concurrent fashion only when all its threads converge back to the same execution path. This behavior, affects the execution speed of the whole warp by up to 11%. Our kernel could be dramatically affected by branch divergence, for example when threads in the same warp process passphrases of different lengths. This would result in a distinct branch logic for most of the threads, leading to a full serialization of all the threads inside the warp.

To investigate the impact of this architecture constrain, we performed the following experiment: We generated five dictionaries: three with the same fixed passphrase length (8, 12 and 16 characters) and two with passphrases of mixed length. Each dictionary held one million passphrases. The mixed 1–16 dictionary is a special passphrase arrangement that generates sixteen different branches on each warp: a block of sixteen consecutive passphrases contains sixteen passphrases each of different lengths. Note that we could not test for the full thirty-two different branches because of the kernel restriction on the passphrase length (sixteen characters).

By varying the input data, the branch divergence metric changes (see second column of Table 4). As expected, the worst case is with the mixed 1–16 dictionary. It is clear that fixed length dictionaries allow a 11% performance increase. Regardless of the passphrase length, if it is fixed, the kernel speed will not be affected, whereas with a non fixed passphrase length, the speed decreases (see last column of Table 4).

With all the passphrases being of the same number of characters, each thread in every warp executes the same length-dependent code branch, resulting in no serialization.

### 4.5. Fixed length password optimizations

A set of the tests, whose results are reported in Table 5, indicates that kernels specialized for dictionaries containing only passphrases with the same length, produce a noticeable speedup. As a consequence, we decided to investigate an effective way of developing such kernels for any possible length of the passphrase.

#### 4.5.1. Meta-programming

To easily create and maintain length-optimized kernels, we employ function templates. Function templates are a form of meta-programming that allows a programmer to write generic functions in a data-type independent fashion. CUDA supports function templates to the full extent of the C++ standard [Abrahams and Gurtovoy, 2004].

#### Table 5

<table>
<thead>
<tr>
<th>GPU Version 4: passphrases of fixed length.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>CPU – single thread</td>
</tr>
<tr>
<td>GPU v.3 – coalesced access</td>
</tr>
<tr>
<td>GPU v.4 – fixed length input</td>
</tr>
</tbody>
</table>

---

input values are constant for each SHA1 iteration. This insight leads to the more efficient Code 3.

The pre-computed access pattern is very effective, reaching a speedup of 100× over the reference CPU implementation (see Table 8).

5. Final results and discussion

Through a process of continuous improvements and analysis we managed to tremendously improve our OpenPGP attack. The final version of our application is able to perform 100× times better than the original reference implementation (Table 9).

Our implementation scales well with respect to the number of multicores available on the GPU. By simply replacing the previous generation device (Tesla C1060) with a latest generation card (based on the new Fermi architecture), we doubled the performances (see Fig. 10).

5.1. A faster CPU version inspired by the GPU one

In the development process of the GPU kernels we found new ways to optimize parts of the attack. Since these optimizations were independent from the architecture, we back-ported them into the CPU code producing a new CPU-version, that is much faster than the reference one (see Fig. 11). Most of this speed up is due to the SHA1 loop unroll and to the improved iterated and salted step (described in Section 4.1.2) that has a better access pattern to the cache memory.

![Fig. 10. Speedup of Fermi vs Tesla c1060.](image-url)
6. Conclusions

The presented results should discourage anyone from using simple dictionary-based passphrases/passwords, even in different languages or with simple transformations. We are confident that many of the techniques and optimizations used and implemented for our GnuPG secret ring attack may be found useful and applicable to other cryptosystems. As a matter of fact, we are already working on a similar fast technique to check passwords used to protect zip archives.

Moreover, we demonstrated that a GPU may be a very effective platform to run applications that require integer-based calculations. The optimization techniques developed during the tuning process may be easily applied to other GPU applications and in some cases to a standard CPU as well.

Finally, although we are still far from reaching the computing power required to carry out a complete brute force attack, the security community can hardly ignore that a single, latest generation, GPU is able to test 500k passphrases per second.

Acknowledgment

We would like to thank Massimiliano Fatica (Nvidia) for providing early results on a Fermi GPU.

References


Nvidia, 2010. PTX ISA 1.4.


Fabrizio Milo received his MS in Computer Science from the University of Rome “Sapienza” in 2011. He is currently doing applied research for a private company in Los Angeles.

Massimo Bernaschi has been 10 years with IBM working in the field of parallel and distributed computing. Currently he is with the National Research Council of Italy (CNR) as Chief Technology Officer of the Institute for Computing Applications. He is also an adjunct professor of Computer Science at “La Sapienza” University in Rome.

Mauro Bisson received his PhD in Computer Science from the University of Rome “Sapienza” in 2011. He is currently a postdoctoral fellow at the School of Engineering and Applied Sciences, Harvard University. He is also a research fellow at Brigham and Women’s Hospital in Boston.