UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie

MSc in Computer Science

EXPLOITING SHA-1 WEAKNESSES TO SPEED UP PBKDF2

Advisor: Prof. Andrea VISCONTI

Co-advisor: Prof. Giuliano GROSSI

Author:

Andrea Francesco IUORIO

ID. Number: 867039

Academic Year 2016-2017

Contents

Chapter 1

Introduction and problem description

Modern computer systems have the necessity to track the identities of the users requesting their services because identity is an important parameter in access control decisions. A system then has the need to authenticate users. We define as authentication the process of verifying the identity of an user. A common solution to this problem is to use a *secret*, an information known only to the legitimate user. By providing the secret the user can prove their identity. The secure handling and usage of secrets are an important aspect in computer security because anyone possessing the secret information can be identified as the legitimate user.

One of the most commonly used secrets are *passwords*. Passwords or passphrases are strings of text, a sequence of characters, known only to a specific user. Passwords can be used as secrets because if the user can prove to know the password they can prove their identity. Proving the knowledge of a password is trivial: the user only needs to provide it to the system, usually writing it with a keyboard. Since

they are extremely easy to use and implement they are one of the most used way to authenticate users even if they presents some problems from a security standpoint: users often reuse passwords on different systems and these passwords are short and presents low entropy [1].

Encryption algorithms encode an information in such a way that it can only be read by authorized people [2]. For doing so, the algorithm use a secret called *cryptographic key*. The key specify the transformation of plaintext into ciphertext and vice versa, so only using the correct key one can retrieve the original information. This cryptographic key is often implemented as a fixed-length sequence of bits. This presents several usability problems for an human user: these bit sequences are often quite long and they can be difficult to remember or provide in some context like on mobile devices. Both passwords and cryptographic keys are secrets but passwords are much easier to use than cryptographic keys. Is it possible to use a password as a cryptographic key? Passwords are character sequences, cryptographic keys are bit sequences: this means that it is necessary to use an algorithm that converts a password into a cryptographic key. This algorithm is called Key Derivation Function (KDF).

A Key Derivation Function is an algorithm that converts a password into a secure cryptographic key. Kerckhoffs's principle says that the security offered by a cryptographic algorithm depends only by the security of the cryptographic key itself. Because of this principle it is not enough to do a simple encoding of the password characters in bit: the conversion should generate the strongest possible key. KDF algorithms must presents 3 important proprieties:

- Given a password X, compute $KDF(X)$ should be "fast enough" for a legitimate users.

- Compute $KDF(X)$ should be as slow as possible without contradict the first point.
- Given $Y = KDF(X)$, there must be no significantly faster way to test q password candidates $X_1, X_2, ..., X_q$ than by actually computing $KDF(X_i)$ for each X_i.

If an algorithm respects these 3 proprieties, it can generate strong keys even when it uses a short and low-entropy password as input: for finding which password generates that key it is necessary to execute a brute force attack because of Propriety 3 but for Propriety 2 the computation of each single password candidate is slow, making the entire attack unpractical. Propriety 1 however means that, if a user knows the password, the computation of the key should be fast: the algorithm should not present a disadvantage to a legitimate user.

In 2000, RSA Laboratories defined in RFC 2898 [3] the Password Based Key Derivation v2 (PBKDF2). Today PBKDF2 is the most widespread KDF: it is used in WiFi Protected Access (WPA/WPA2), full disk encryption on iOS, macOS (Filevault) and Linux (LUKS) and by several applications like password managers. It is also worth noting that some applications, like Apacha Spark, uses PBKDF2 for client/server authentication. RFC 8018 [4], published in 2017, still recommends PBKDF2 for password hashing. NIST also released in 2017 a new standard for password authentication and storage that uses PBKDF2 [5].

PBKDF2 uses a pseudo-random function (PRF) for generate the output key. While it is possible to use any PRF, the standard suggests to use HMAC-SHA1 [6] as the PRF. Implementations that uses HMAC-SHA256 as their PRF are also common. For slowing down attackers, PBKDF2 uses a salt and an iteration count. The latter specifies the number of times the pseudo-random function is iterated to generate a key

of appropriate size. The iteration count is one of the most important parameters of PBKDF2. The choice of a high value slows attackers down but may negatively affect usability. In [7], NIST recommends a minimum of 1,000 iterations for general purpose applications but suggests to select the iteration count value as large as possible. Interestingly, many applications define such a value a priori — for example, WPA2 set the iteration count value to 4096 [8] — while others do not — e.g., the iteration count associated to iOS passcodes is calibrated to take about 80 milliseconds [9]. Since Key Derivation Functions transform user-provided passwords into secure keys, a great number of software and protocols uses PBKDF2 for authentication and key generation: as an example, Figure 1 shows how a Linux Unified Key Setup archive is decrypted using the password provided by the user.

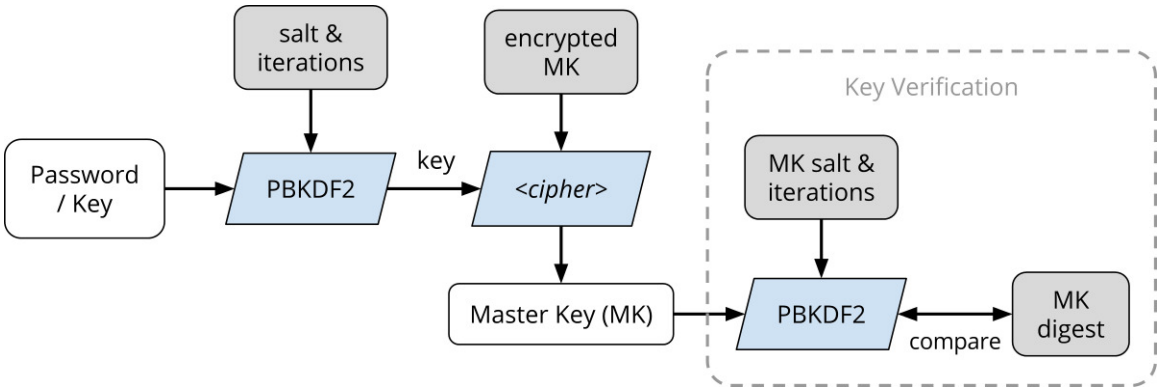

Figure 1: Linux Unified Key Setup verification flowchart

PBKDF2 is designed in such a way that the computation of the key cannot be done in parallel in an effort to reduce the amount of time for generate a key. However, in the recent years, thanks to the growing popularity of cryptocurrencies and the growing video-game industry, the market was flooded by cheap hardware architectures that

are able to execute several instances of PBKDF2 at the same time. An example of these architectures is presented by Graphic Processing Units (GPUs). GPUs are heavily specialized chips for image computation but, thanks to their programmable pipeline, in the last few years they became a cheap way to execute general-purpose parallel computations. Because of the low-memory requirements of PBKDF2, GPUs can execute a lot of instances of this KDF at the same time. Using a GPU, an attacker can accelerate by a lot the execution of PBKDF2 at a reasonable low cost.

In this thesis we focus on accelerating PBKDF2-HMAC-SHA1 on CPU and GPU architectures. While this is not something new, we wanted to concentrate on these novel issues:

- How much can we accelerate a PBKDF2 implementation? What can be learned while developing an highly optimized implementation from scratch?
- How much optimization techniques to the internal PRF can contribute to the acceleration of PBKDF2?
- Which are the computational possibilities offered by current consumer-grade hardware?
- For which kind of applications can we still consider PBKDF2-HMAC-SHA-1 secure?

We chose to focus on PBKDF2-HMAC-SHA1 because, despite there are some recent security concerns about SHA-1 [10], this version represents the current standard version of PBKDF2. In fact, even with these security concerns, HMAC-SHA1 is still considered secure to use as a pseudorandom function. PBKDF2-HMAC-SHA1 is used by several applications like full disk encryption softwares [11,12] and it is widely

deployed as an authentication method, being used in systems like WiFi Protected Access (WPA/WPA2) [8] and Apache Spark [13]. In addition, in the Internet of Things (IoT) era users want to be able to access to their accounts on all their devices, thus adopting password managers to remember and secure user-chosen passwords. Notice that several password manager applications [14–18] are based on PBKDF2 and a number of security and privacy concerns have to be addressed [19, 20].

For accelerating PBKDF2-HMAC-SHA-1 we first reviewed the current state of the art, selecting several optimization techniques used to speed up PBKDF2, HMAC and SHA-1 in a GPU/CPU context. We also tried and developed a couple of optimizations for the SHA-1 algorithm ourselves, when it is used in PBKDF2. Then, in order to measure the contributions provided by these optimizations, we developed from scratch an highly optimized implementation of PBKDF2-HMAC-SHA-1. Finally, we used this implementation to execute testing activities on consumer-grade hardware, trying to understand the impact each optimization had on the acceleration process.

This thesis is organized as follows.

- In Chapter 2 we introduce the necessary cryptographic background.
- In Chapter 3 we introduce the necessary background on GPGPU programming.
- In Chapter 4, we described the optimizations techniques that can be used to speed up PBKDF2, HMAC and SHA-1.
- In Chapter 5, we describe both our CPU and GPU implementations, the techiniques we have used and the problems we encountered during the implementation.
- In Chapter 6, we show the experimental result found by our implementation.

- Finally, in conclusions about our results and their implications for the security of PBKDF-HMAC-SHA-1 are drown in Chapter 7.

Chapter 2

Cryptographic preliminaries

In this Chapter we provide the necessary cryptographic background for understanding our testing activities. We start with a quick explanation of what a Key Derivation Function is, then we present PBKDF2, the KDF we concentrate in this thesis. Finally we explore the internal cryptographic primtives used by PBKDF2, more specifically HMAC and SHA-1 hashing algorithms.

2.1 Introduction to Key Derivation Functions

A Key Derivation Function (KDF) is a function that, using some initial keying material, it generates a secure cryptographic key, ie. an array of bits. In practice the most used type of KDF are password-based Key Derivation Function, functions that uses passwords as their keying material.

A password-based KDF generates a secure cryptographic key. Not all functions that converts passwords into arrays of bits can be considered good KDF. A KDF should satisfy 3 important proprieties to be considered secure. These three proprieties are:

- Given a password X, compute $KDF(X)$ should be "fast enough" for a legitimate users
- Compute $KDF(X)$ should be as slow as possible without contradict the first point
- Given $Y = KDF(X)$, there must be no significantly faster way to test q password candidates $X_1, X_2, ..., X_q$ than by actually computing $KDF(X_i)$ for each X_i.

If the algorithm presents these proprieties, the generated key can be considered secure enough even when using a low-entropy password as keying material. In our threat model we assume that an attacker has the resulting key and they want to understand which password was used for generating it. Propriety 1 means that the execution of a single instance of the KDF should be fast: a legitimate user, which knows the correct password, has to execute at most one time the KDF for the correct conversion. However, because of propriety 2, executing several instances of the KDF should be as slow as possible. The careful balance between these two times is often a source of problems for implementers, as we will seen in the rest of this chapter. Propriety 3 is what permits to the generated key to be considered secure. Because of it, the attacker has to try all possible passwords, however because of propriety 2 this will require a considerable amount of time, making the attack too slow and expensive even if the pool of possible password is not big.

2.2 PBKDF2

PBKDF2 is a password-based key derivation function: starting from a password, the algorithm generates a key of fixed length. PBKDF2 can be described as a chain of

several instances of a pseudorandom function. In this paper we concentrate ourselves on the version that uses HMAC-SHA-1 as the pseudorandom function: even if SHA-1 is currently considered unsafe when we want to avoid hash collisions, the NIST still supports SHA-1 when used a key derivation function.

PBKDF2 is a Password-Based Key Derivation Function described in PKCS #5 [3, 21, 22], [7]. For providing better resistance against brute force attacks, PBKDF2 introduce CPU-intensive operations. These operations are based on an iterated pseudorandom function (PRF) which maps input values to a derived key. The most important properties to assure is that the iterated pseudorandom function is cycle free. If this is not so, a malicious user can avoid the CPU-intensive operations and, as described in [23], get the derived key by executing a set of functionally-equivalent instructions.

PBKDF2 inputs a pseudorandom function PRF, the user password p, a random salt s, an iteration count c, and the desired length len of the derived key. It outputs a derived key $DerKey$.

$$DerKey = PBKDF2(PRF, p, s, c, len) \tag{1}$$

More precisely, the derived key is computed as follows:

$$DerKey = T_1||T_2||\ldots||T_{len}, \tag{2}$$

where

$$T_1 = Function(p, s, c, 1),$$

$$T_2 = Function(p, s, c, 2),$$

$$\vdots$$

$$T_{len} = Function(p, s, c, len).$$

Each single block T_i — i.e., $T_i = Function(p, s, c, i)$ — is computed as

$$T_i = U_1 \oplus U_2 \oplus ... \oplus U_c, \tag{3}$$

where

$$U_1 = PRF(p, s||i),$$

$$U_2 = PRF(p, U_1),$$

$$\vdots$$

$$U_c = PRF(p, U_{c-1}).$$

The PRF adopted can be a hash function [24], cipher, or HMAC [25], [26], and [6]. In the sequel, we will refer to HMAC as PRF.

2.3 HMAC

An Hash-based Message Authentication Code (HMAC) is an algorithm for computing a message authentication code based on any iterated cryptographic hash function. The definition of HMAC [6] requires

- H: a cryptographic hash function;
- K: the secret key;
- $text$: the message to be authenticated.

As described in RFC 2104 [6], an HMAC can be defined as follows:

$$HMAC = H(K \oplus opad, H(K \oplus ipad, text)) \tag{4}$$

where H is the chosen hash function, K is the secret key, and *ipad*, *opad* are constant values — respectively, the byte 0x36 and 0x5C repeated 64 times. Recall that, Equation 4 can be expanded in the form:

$$h = H(K \oplus ipad \mid\mid text)$$

$$HMAC = H(K \oplus opad \mid\mid h)$$

In our performance tests, the hash function adopted will be SHA-1, thus making HMAC-SHA-1 the default pseudorandom function.

2.4 SHA-1

SHA-1 is a cryptographic hash function that inputs an arbitrarily long message M and outputs a 160-bit digest H. In order to provide the message digest, SHA-1 operates eighty times on five 32-bit words A, B, C, D, and E as shown in Figure 2. F_t is defined

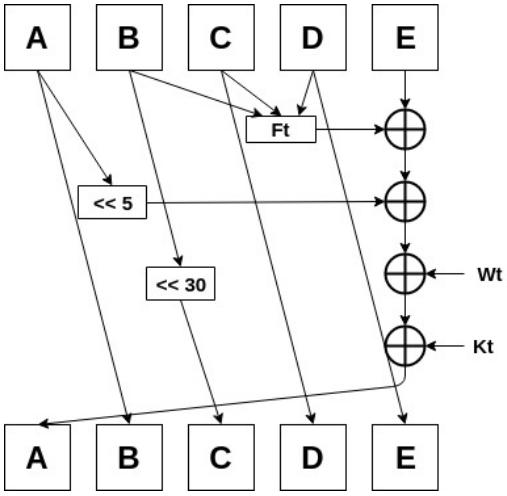

Figure 2: SHA-1

by

$$F_t = \begin{cases} F_0 = (B \wedge C) \vee ((\neg B) \wedge D) & t \in [0 \ldots 19] \\ F_1 = (B \oplus C \oplus D) & t \in [20 \ldots 39] \\ F_2 = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & t \in [40 \ldots 59] \\ F_3 = (B \oplus C \oplus D) & t \in [60 \ldots 79] \end{cases}$$

and K_t assume four constants value (see [24] for details).

Notice that expressions F_0 and F_2 of F_t are logical equivalent to

$$F_0 = \begin{cases} D \oplus (B \wedge (C \oplus D)) \\ (B \wedge C) \oplus ((\neg B) \wedge D) \end{cases}$$

$$F_2 = \begin{cases} (B \wedge C) \vee (D \wedge (B \vee C)) \\ (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D) \end{cases}$$

and that can be used instead of the expressions in [24].

Message M is processed in blocks of the size of 512 bits, namely, sixteen 32-bit words $W_0, \ldots, W_{15}$, eventually padding the last block. More precisely, the last block is padded with one bit **1** first then, zero or more bits **0** so that its length is congruent to 448, modulo 512. The remaining 64 bits of the last 512-bit block represent the message length L. The SHA-1 algorithm expands 32-bit words $W_0, \ldots, W_{15}$ into eighty words using the follow message scheduling function:

$$W_i = ROTL^1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \qquad i \in [16 \ldots 79] \tag{5}$$

where $ROTL(x, n)$ is the left rotation of x by n bits. Notice that Equation 5 requires to store eighty 32-bit words. If memory is limited (e.g. embedded devices and GPUs),

an alternative method should be adopted. NIST suggests to regard $W_0, \ldots, W_{15}$ as a circular queue [24] and substitute the Equation 5 with the following:

$$\begin{cases} s = i \wedge MASK & i \in [16 \ldots 79] \\ W_s = ROTL^1(W_s \oplus W_{(s+2)\wedge MASK} \oplus W_{(s+8)\wedge MASK} \oplus W_{(s+13)\wedge MASK}) \end{cases} \tag{6}$$

where $MASK$ is set to the value $0x0F$ in *Hex*. Equation 6 requires only sixteen words, thus saving sixty-four 32-bit words of storage.

Further improvements have been presented in [27]. In particular, the authors suggest to replace Equation 5 with

$$W[i] = \begin{cases} ROTL^1(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) & i \in [16 \ldots 31] \\ ROTL^2(W_{i-6} \oplus W_{i-16} \oplus W_{i-28} \oplus W_{i-32}) & i \in [32 \ldots 63] \\ ROTL^4(W_{i-12} \oplus W_{i-32} \oplus W_{i-56} \oplus W_{i-64}) & i \in [64 \ldots 79] \end{cases} \tag{7}$$

and then replace W_{29}, W_{30}, W_{31}, W_{60}, and W_{62} with the following and less expensive equations:

$$\begin{cases} W_{29} = ROTL^2(W_{23}) \oplus k[29] \\ W_{30} = ROTL^2(W_{24} \oplus k[16]) \\ W_{31} = ROTL^2(W_{25} \oplus k[17]) \oplus k[31] \\ W_{60} = ROTL^4(W_{48} \oplus W_{28} \oplus W_0) \\ W_{62} = ROTL^4(W_{50} \oplus W_{30} \oplus W_0) \end{cases} \tag{8}$$

where $k[29] = ROTL^2(W_5) \oplus ROTL^1(W_{15})$, $k[16] = W_0 \oplus W_2$ (previously computed in W_{16}), $k[17] = W_1 \oplus W_3$ (previously computed in W_{17}), and finally $k[31] = ROTL^1(W_{15}) \oplus ROTL^2(W_{15})$.

In addition, [27] states that Equation 6 can be replaced with the unfolded version:

$$\begin{cases} W_{16} = W_0^1 \oplus W_2^1 \oplus W_8^1 \oplus W_{13}^1 \\ W_{17} = W_1^1 \oplus W_3^1 \oplus W_9^1 \oplus W_{14}^1 \\ W_{18} = W_2^1 \oplus W_4^1 \oplus W_{10}^1 \oplus W_{15}^1 \\ W_{19} = W_0^2 \oplus W_2^2 \oplus W_3^1 \oplus W_5^1 \oplus W_8^2 \oplus W_{11}^1 \oplus W_{13}^2 \\ W_{20} = W_1^2 \oplus W_3^2 \oplus W_4^1 \oplus W_6^1 \oplus W_9^2 \oplus W_{12}^1 \oplus W_{14}^2 \\ W_{21} = W_2^2 \oplus W_4^2 \oplus W_5^1 \oplus W_7^1 \oplus W_{10}^2 \oplus W_{13}^1 \oplus W_{15}^2 \\ W_{22} = W_0^3 \oplus W_2^3 \oplus W_3^2 \oplus W_5^2 \oplus \cdots \oplus W_{11}^2 \oplus W_{13}^3 \oplus W_{14}^1 \\ W_{23} = W_1^3 \oplus W_3^3 \oplus W_4^2 \oplus W_6^2 \oplus \cdots \oplus W_{12}^2 \oplus W_{14}^3 \oplus W_{15}^1 \\ \cdots \\ W_{79} = W_0^8 \oplus W_0^{22} \oplus W_1^7 \oplus \cdots \oplus W_{15}^{14} \oplus W_{15}^{17} \oplus W_{15}^{18} \end{cases} \tag{9}$$

where $W_i^j = ROTL^j(W_i)$. Notice that, although Equation 9 increases the total number of XOR operations, if we compute PBKDF2-HMAC-SHA-1, it requires to store only five 32-bit words, namely $W_0, \ldots, W_4$, because $W_6, \ldots, W_{14}$ are equal to zero, and W_5, W_{15} are constant value. Therefore, this approach might be exploited by GPGPU programming.

Chapter 3

Introduction to GPGPU programming

Created initially for image processing, Graphic Processing Units (GPUs) became in the last few years a powerful platform for general-purpose parallel computations [28]. The fast-growing video game industry has motivated a rapid advancement of graphics hardware and architectures, making GPUs faster and cheaper: today is possible to buy a GPU that, for parallel computations, can outperform a CPU in the same price range by several order of magnitude [29].

GPUs possess a different architecture than general-purpose CPUs and they are optimized for parallel computation. A task should present these specific proprieties to be optimally executed on a GPU:

- The task must be divisible in smaller tasks.
- Each of these smaller tasks should execute the same instructions.
- Each of these smaller tasks should use as little memory as possible.

In this Chapter we give a brief description of the architecture of a GPU, how the software for this kind of hardware should be developed and how a GPU program is different from a CPU program. The purpose of this Chapter is to provide the necessary background for understanding the implementation we have developed from scratch for our testing activity . We have used OpenCL [30] as the language and framework for our implementation but the concepts in this chapter can be easily applied to other GPGPU framework and languages like CUDA [31].

3.1 Introduction to OpenCL

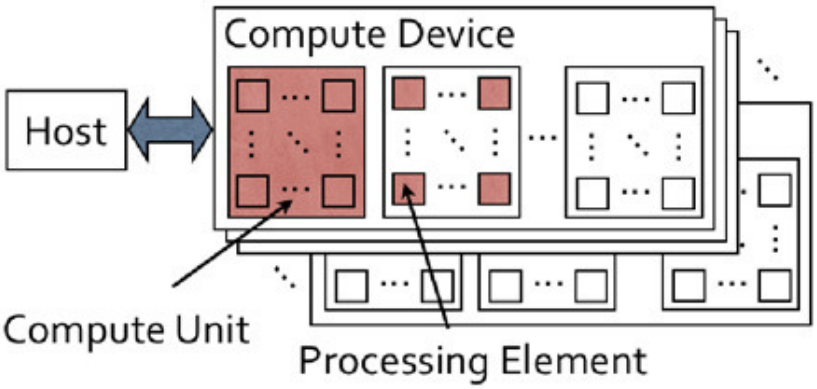

Figure 3: The OpenCL application model

The Open Computation Language (OpenCL) is a framework and programming language for programs that runs on heterogeneous platforms with CPUs, GPUs or FPGAs [32]. OpenCL, in contrast to other GPGPU frameworks, is an open standard maintained by Khronos Group [33] and it is supported by almost every GPU produced in the last 7 years.

An OpenCL application uses a client-server model, as shown in Figure 3: the application has a central host split the computation into several processing units and then it sends them to the Computation Devices (CD). These Computation Devices possess several Computation Units (CU). The CU executes the processing units, separated instances of the same GPU application. Often, in practical implementations, the host is a CPU application that handles how the workload should be executed on the GPUs connected to the system, making the GPUs the Computation Devices. OpenCL has APIs in several languages for writing the host application, while the kernel is written using the OpenCL language.

Kernels are written using in a C99-like programming languages created for making it easy to write efficient GPU applications. A work item is an instance of a kernel and it is executed on the Computation Devices. Since these devices can be different hardware with different architectures and different Instruction Set Architectures (ISA), an OpenCL kernel is often compiled just-in-time. This permits to execute the same kernel on different machines, however it presents problems for the distribution of the application. In 2015 Khronos Group has developed the Standard Portable Intermediate Representation (SPIR) [34], a binary intermediate language that can be executed by any OpenCL-supported Computation Devices.

3.2 Differences between CPU and GPU applications

A CPU is usually formed by a central Control Unit (CU) and by one or more Arithmetic Logic Unit (ALU). A CPU execution cycle can be simplified in these operations:

- Fetch the current instructions from the main memory.

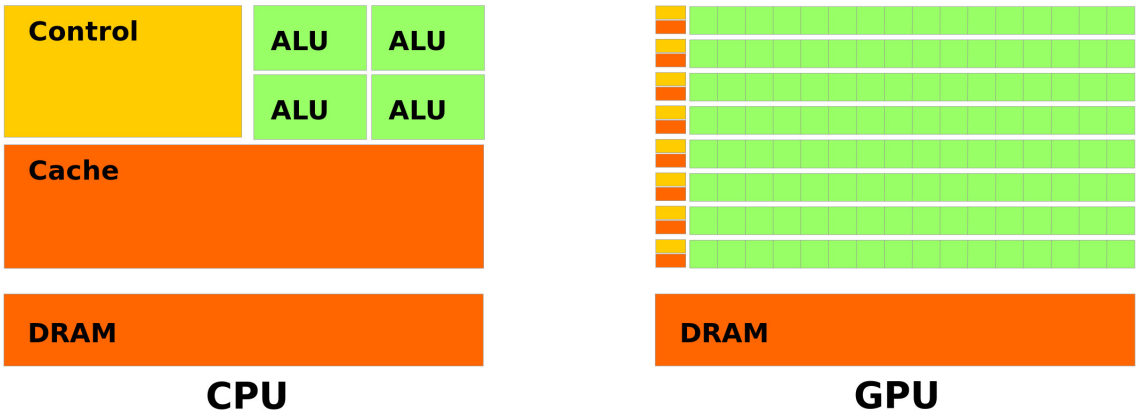

Figure 4: Differences between CPU and GPU architectures

- The CU decode the current instruction, setting the ALUs accordingly.
- The ALUs execute the operation.

The CPU possess a central memory. However the CPU does not access the memory directly because, for hardware reasons, each memory access is much slower than the execution of an instruction [35]. If an instruction needs to access the memory, it means that the CPU should wait for the data present in memory before continuing the execution, introducing latency during the execution of the instruction. For avoiding this problem, CPUs uses a hierarchy of caches for trying to reduce at a minimum a direct access to the central memory.

A GPU is usually formed by two main elements: an on-device memory and multiple streaming multiprocessors (SM) [36]. A streaming multiprocessor includes these elements:

- An Instruction Decoding Unit for decoding the instructions.

- A little cache memory, used as constant memory.
- A little, read-only, memory for the kernel's instructions.
- A bigger, on-chip memory used as local memory.
- Several thread processors (SP). Each of these thread processor contains an ALU for computations.
- Several Special Function Units (SFU), specialized ALUs that can execute in hardware several complex arithmetic operations.

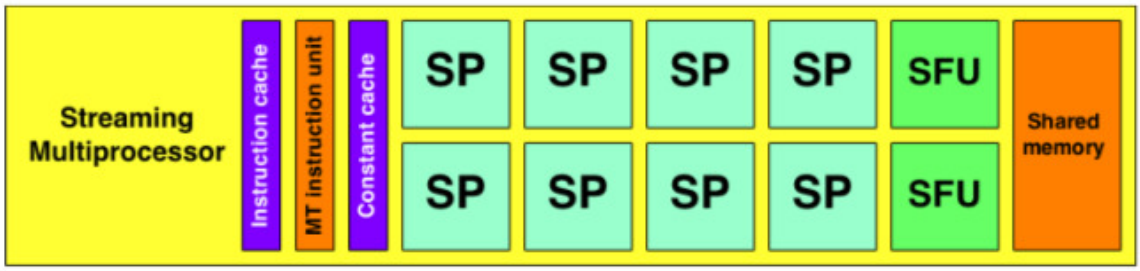

Figure 5: Structure of a Streaming Multiprocessor

Figure 5 shows the structure of a SM. For making the development of GPU application easier several framework and languages were created in the last decade. Taking the OpenCL model, a GPU application uses a client-sever model: the application possess a central CPU program to be uses as the host application that asks the execution of a set of work items to the GPU, the Computation Device. A work item is an instance, or thread, of a specific GPU program called kernel. The set of work items is then spitted into several blocks. Often a block has a size of 256 work items, however the

size of a block depends on the hardware and how well the application can be split into separated work items. Finally, blocks are spitted into warps: all threads in a warp execute the same instruction at the same time on different inputs: we will later see that each thread has its own set of registers. This execution approach takes the name of Single Instruction Multiple Data (SIMD) execution. Streaming multiprocessors are able to change the current active warp for reducing the impact of a memory access on performances.

The GPU can execute almost every instruction a CPU could: GPUs can be considered general purpose architectures. However there are some important differences. One difference we can already see is presented by conditional jumps. If only some threads in a warp must execute a block of instruction, the other threads must be put on wait, meaning that in that specific moment we do not use all the computation capabilities of the GPU. This phenomenon is called *warp divergence* and it must be considered during the development of GPU applications.

3.3 GPU memory hierarchy

One of the main aspects a programmer must consider during the development of a GPU application is memory handling. CPU and GPU communicate using the system bus, meaning that a GPU can access the CPU memory. However the system bus presents too much latency, meaning that the GPU cannot use the CPU memory in an efficient way. For this reason, GPUs often provide a separate memory on their card.

GPU memory use an hierarchy structure for hide as much as possible the latency introduced by memory accesses. On GPUs a memory access if often really costly: if

a thread in a warp needs to access memory, all the other threads in the warp must wait for that data before they can continue the computation.

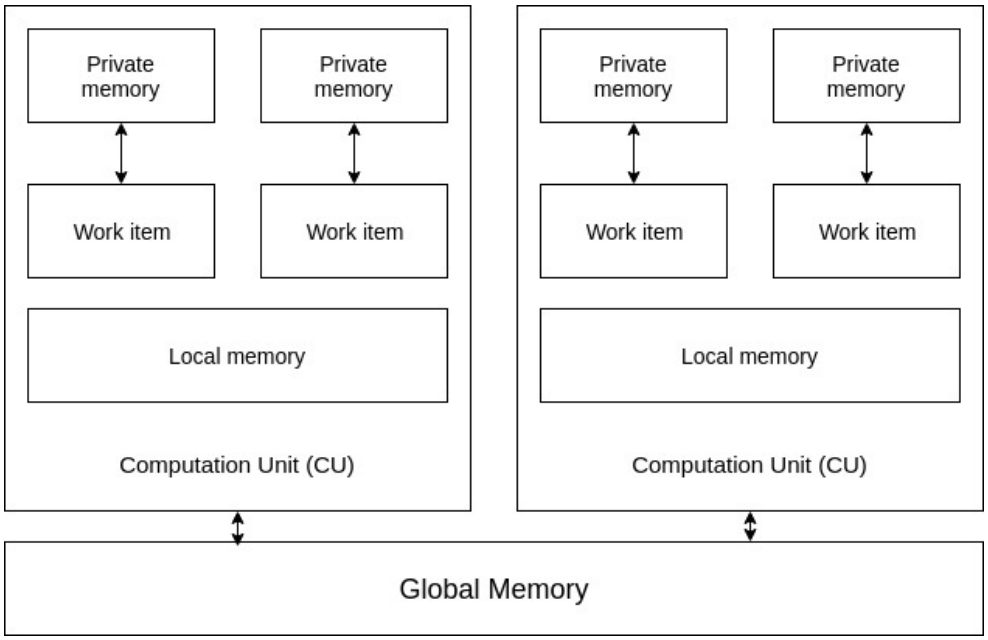

Figure 6: Memory hierarchy on a GPU

OpenCL presents 3 types of memory: Global Memory, Local Memory and Register Memory. Figure 6 shows the memory architecture of an OpenCL application.

3.3.1 Global memory

Global memory is the main memory of the GPU. It is often the bigger memory on the card, in the order of gigabytes, but it is also the slower. The Global memory works like the main memory for a GPU: each thread can access the global memory. There are however a couple of differences between the CPU memory and the GPU memory: for example GPU memory often does not have support for virtual addresses.

Since accessing the CPU memory from the GPU introduces a big latency because

of the system bus, often the host copy the input for the work items on the GPU main memory before starting the computation. At the same way, after the computation, the host must retrieve the results from the GPU memory. This operation can be done asynchronously if the hardware support DMA access.

A little part of the global memory is called Constant Memory. This memory buffer is a read-only area accessible by all work items for saving constants that are needed to the application. Since it is read-only, it can be copied internally to the SM without coherence problems, making the access to the data in the constant memory faster than the rest of the global memory.

3.3.2 Local memory

Local memory is a memory buffer internal to the streaming multiprocessor and it is shared between the work items in the same block. The purpose of this memory is to permit a faster communication between work items in a block, without requiring them to access the global memory. Since it is in the streaming multiprocessor, local memory presents a much lower latency that global memory but the size of this memory buffer is around the 16 to 32 Kb.

Local memory is often spitted into different hardware banks, memory buffers handled on different chips. The access to addresses in different banks can be done in parallel, while the access to addresses in the same bank must be serialized. How the addresses are separated into the memory banks is hardware-dependent, however often the following 32-bit words are in a different banks for trying to get as many accesses in different banks as possible

3.3.3 Registers

Register memory represents the faster memory on a GPU. The register memory contains the registers used by each work item in execution. Because of their architecture with thousands of streaming multiprocessors, registers on a GPU are in a buffer of fast memory. Each work item allocate in this memory the number of registers they require for the computation of the kernel. If a kernel needs lots of registers, however, presents a performance problem: the SM can occupy the entire register memory, meaning that some SM must be kept on slower memories and it increase the possibility of cache misses during a register access.

Chapter 4

Algorithmic optimizations

PBKDF2 applies a pseudorandom function to generate cryptographically secure keys. Since in this process different cryptographic algorithms are involved as we have seen in Chapter 2, the optimization of one of these algorithms usually lead to interesting performance improvements in the key derivation process. But this is not always true. Indeed, some optimizations described in this Section affect SHA-1 or HMAC-SHA-1 but have no effect on PBKDF2-HMAC-SHA-1. A crucial role is played by the context in which the code will be run, namely a GPU or CPU context. In fact, a specific algorithmic optimization may have no impact on GPU performances, while it has on CPU ones. Interestingly, however, the opposite is true as well.

Focusing on the state of the art of PBKDF2, HMAC, and SHA-1, in this Chapter we briefly present the optimizations that provided us any significant improvements.

4.1 PBKDF2 optimizations

[**OPT–01**] **Early exit**: The execution time spent for computing a derived key does not only depend on the iteration count values. Indeed, also the number of fingerprints

T_i required to compute a single iteration affects the total execution time. Assuming that we require a 256-bit derived key, two SHA-1 fingerprint are necessary — i.e., $DerKey = T_1||T_2$, with T_1 and T_2 160-bit length each. Since blocks T_i are independent of each other, firstly we generate a block T_1 and then we compute the second if and only if T_1 is equal to the first part of the 256-bit derived key. If not so, the chosen password p is certainly wrong. Therefore, the check of first 160 bits of the key is enough to discard the majority of invalid candidate passwords [37].

4.2 HMAC optimizations

[OPT–02] Block reduction: Since password p is an input parameter and it is not modified during the computation of PBKDF2, it is possible to precompute the first message block of a keyed hash function as showed by Figure 7) and reuse such values in all the subsequent HMAC invocations. Thus, the number of blocks that have to be computed is reduced from "$4*iteration\ count$" to "$2+2*iteration\ count$". This simple optimization saves about 50% of PBKDF2's CPU intensive operations [23, 37, 38].

[OPT–03] Input size: A generic HMAC implementation has to address the problems of the size of password p and message $text$. If the password length is bigger than 512 bits, it has to be reduced. Therefore, a hash algorithm is applied, namely $p = SHA{-}1(p)$, and then it is padded with enough zeros to reach a 512-bit length [39]. In addition, we have to address also the problem of the message size. If the message to be authenticated is bigger than 512 bits, it has to be split in several blocks and then each block managed separately. In PBKDF2, excluding the computation of U_1 (see Figure 7), we have not a generic HMAC implementation but a specific one.

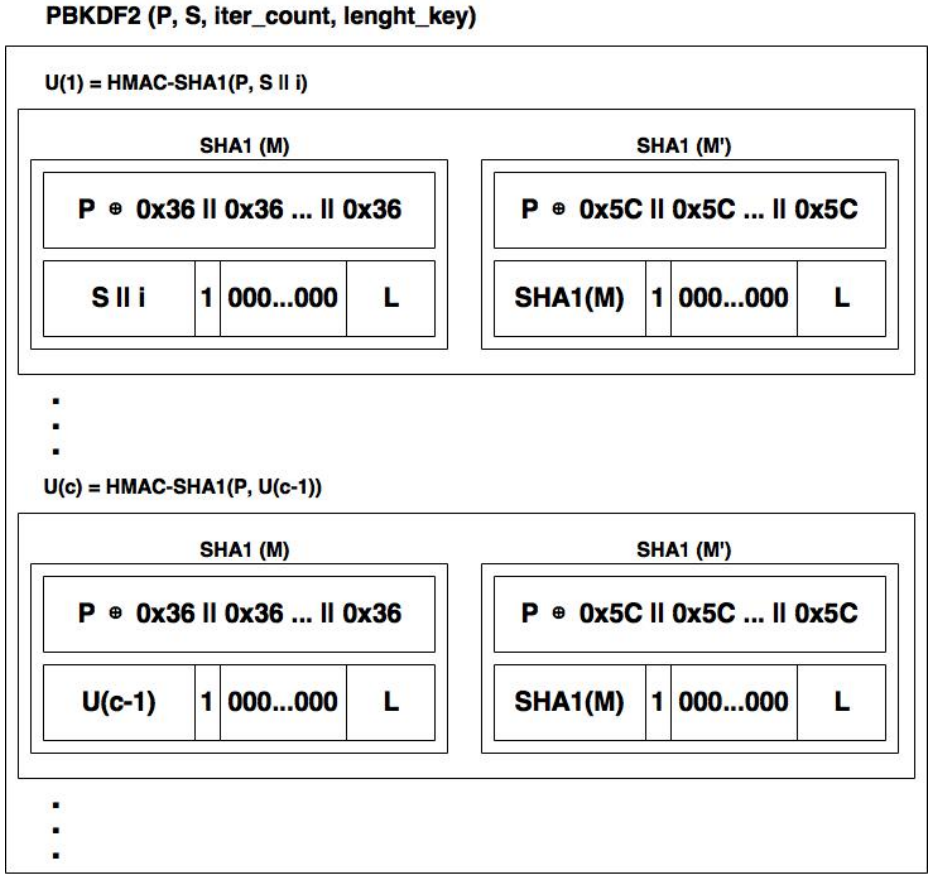

Figure 7: PBKDF2-HMAC-SHA-1 optimizations

Indeed, we known in advance the computation of the first message block (see [**OPT–02**] Merkle-Damgard block reduction), and we have to manage only the second one. Since the second message block always inputs a 160-bit message, namely *SHA-1(M)* or U_i (see Figure 7), we have not to split the message to be authenticated in blocks. Therefore, this optimization provide us the possibility to avoid length checks and the chunk splitting operations during the computation of $U_2,\ldots,U_c$, thus reducing the overhead necessary to compute an HMAC implementation [38].

4.3 SHA-1 optimizations

[OPT–04] Word expansion phase: Instead of using eighty 32-bit words for the word expansion phase (see Equation 5), SHA-1 can be implemented using a circular queue [24] of sixteen words (see Equation 6). This approach reduces the amount of memory required by the implementation, thus making this optimization a desirable feature in a GPU context.

A different approach has been introduced in [27], where the authors suggest the possibility to unfold the SHA-1 message scheduling function (see Equation 9). Although this approach increases the total number of XOR operations to be executed, it drastically reduce the amount of memory required to perform the SHA-1 message scheduling function, i.e., only five 32-bit words. Therefore, also this optimization may have an impact on GPU performances.

In addition, Visconti and Gorla [27] have also shown that Equation 5 can be replaced with Equation 7. This new approach does not reduce the amount of memory required to compute the word expansion phase but can be exploited to reduce the total number of XORs in a CPU context as suggested by **[OPT–05]**.

[OPT–05] Zero-based optimization: Due to a long run of several consecutive zeros, namely 287 bits, a number of 32-bit word W_t are set to zero. Since zero-based operations do not provide any contribution, they can be easily omitted. Therefore, exploiting Equations 7 and 8, we can ovoid 66 out of 192 XOR operations [27].

The same approach can be adopted to reduce the number of constant XORed twice — i.e., $0x36$ and $0x5C$ — when passwords p are short. We recall that XORing the same value twice does not provide any contribution and can be omitted [23].

[OPT–06] Three-round optimization: During the computation of the message digest, SHA-1 operates on several 32-bit words such as constants K_t, registers A, B, C, D, E, functions f_t and W_t too. However, in the first three rounds a number of these words are known a priori and some operations can be omitted [38]. For example, in the first round we have to compute the following equation: $f_0 + E + ROTL(A, 5) + W_0 + K_0$ (see Figure 2). The content of 32-bit word W_0 is unknown but those of f_0, E, the circular shift of A, and K_0 are not. Therefore, we can precompute $f_0 + E + ROTL(A, 5) + K_0 = 0x9FB498B3$ and reduce the first round to a single operation, namely $W_0 + 0x9FB498B3$, thus saving 3 operations out of 4. This approach can be also applied to second and third round, where the unknown values are A, W_1, and A, B, W_2, respectively.

Chapter 5

Implementation optimizations

For our testing activities described in Chapter 6 we have used our implementation of PBKDF2. This implementation was developed from scratch instead of using off the shelves tools and libraries like hashcat [40]. We decided to take the effort to implement PBKDF2 and its cryptographic primitives so that we were able to easily implement al the optimizations we have presented in Chapter 4 and observe the impact each optimization have on the acceleration of PBKDF2.

We have developed two versions of PBKDF2 for two different architectures: a GPU version using OpenCL and a CPU version. We have implemented PBKDF2 on both architectures because of the diversities we have presented in Chapter 3. Since the purpose of this work is to understand how much impact the optimizations have on the acceleration of PBKDF2, it is reasonable to expect that some of them can have a different impact based on the architecture used for the execution of the algorithm. In this Chapter we present both implementations and the difficulties we have encountered during their development.

5.1 Details of GPU implementation

Using modern GPU hardware it is possible to execute several thousand instances of PBKDF2 at the same time on a single GPU. Since we want to accelerate PBKDF2, we have two ways to achieve this goal:

- Reduce the execution time of a single PBKDF2 instance.
- Increase the number of instances we can execute in parallel on the GPU.

Interestingly, these approaches are interconnected: reducing the execution time of the single PBKDF2 instance can reduce the execution time of the execution warps, meaning that the GPU can run more warps per second. Also an important factor of our implementation will be how the kernel handle the several layers of GPU memory and how this memory is accessed.

As we have seen in Chapter 3, an OpenCL application usually possess an host application and several kernels. These kernels are executed on the Computation Devices. In our case the computation devices are GPUs, so for the rest of this Chapter we will assume that the kernels will be executed on this kind of hardware. The purpose of our application is to execute PBKDF2 instances on a GPU, so we need to implement this algorithm using kernels. This does not mean that PBKDF2 must be implemented using a single kernel: as we have seen big kernels uses more memory and registers. This can reduces the number of concurrent instances that can be efficiently executed on the device. In our testing activities we have tried several approaches:

- We have implemented PBKDF2 using a kernel that implements HMAC. Then PBKDF2 is executed using a queue of these HMAC kernels. We have found

that this approach generates kernels that uses very little memory and registers, however the overhead of starting and stopping kernels became extremely costly if the iteration count was big, making this implementation non viable.

- We implemented PBKDF2 using several kernels, each implementing only a certain number of HMAC iterations. The problem of this approach was finding the best amount of HMAC iterations that can be executed on the same kernel while keeping the memory consumption and the overhead of finishing and starting a new thread as little as possible.

- We implemented PBKDF2 as a single kernel. The resulting kernel used the most amount of memory and registers but it was the simplest to implement.

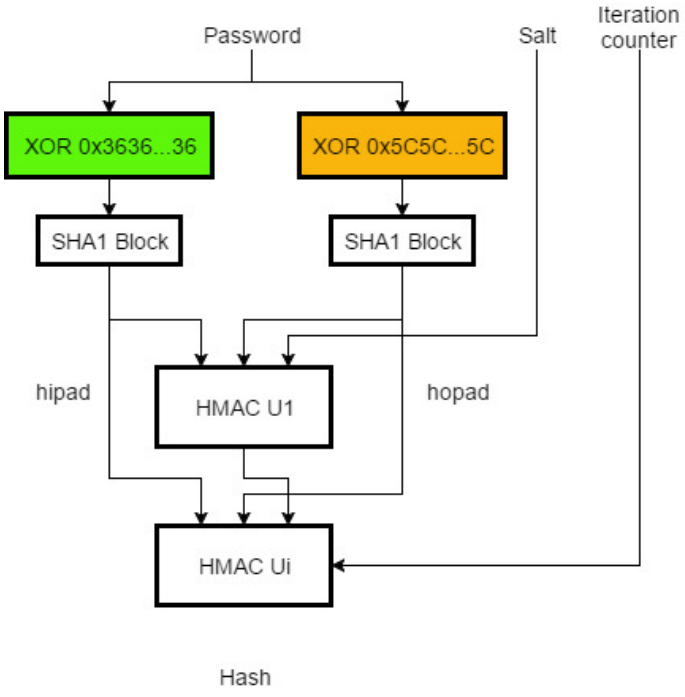

Figure 8: PBKDF2 GPU implementation

In our implementation we used an union of these approaches. Figure 8 shows how we have implemented PBKDF2. For this implementation we decided to use two GPU kernels:

- A kernel that executes only the first iteration of HMAC.
- A kernel that executes the remaining $i - 1$ iterations.

We used this division for hiding the memory transfer time between the host and the computation device: the biggest part of the input of PBKDF2 is the password and the salt. However when we use the [**OPT–02**] these values are only used during the first iteration. This means that after the execution of the first iteration we can update the memory buffer containing the passwords and salts without race conditions with the GPU computation. This can be done with a DMA transfer between the host and the computation device. Figure 9 shows how the host does this transfer. Thanks to this solution the device is able to start the execution of the next set of password candidates immediately, hiding the transfer time of the input between host and device.

5.1.1 OpenCL host implementation

The host application is a CPU program that must handle the execution of the kernels on our GPUs. This application must:

- Send on the GPU Global memory the input of the kernels.
- Put the kernels instructions in the execution queue of the GPU.
- Recover from the GPU Global memory the output of the computation.

```
1    temp = clEnqueueMapBuffer(ctx->dataQueue, ctx->pinned_buffer,
2        CL_FALSE, CL_MAP_WRITE, 0,global_size*64,0,NULL,&writePinned,&err);
3    ...
4    //Buffer ready, let's see if the GPU has finished the first iteration
5    clWaitForEvents(1,&writePinned);
6    memcpy(temp,ptr,global_size*64);
7    clEnqueueUnmapMemObject(ctx->dataQueue, ctx->pinned_buffer,temp,0,NULL,NULL);
8    clReleaseEvent(writePinned);
9  ...
10   clEnqueueCopyBuffer(ctx->dataQueue,ctx->pinned_buffer, ctx->key_buffer,0,0,
11       global_size*64,1,precList,eventList);
12   /* The dictionary is on GPU, let's enqueue PBKDF2*/
13   clEnqueueNDRangeKernel(ctx->queue, ctx->kernel_precomp, 1, NULL, &global_size,
14       &local_size, 1, eventList, precList);
15   clEnqueueNDRangeKernel(ctx->queue, ctx->kernel_hmac_sha1, 1, NULL, &global_size,
16       &local_size, 0, NULL, NULL);
17   clEnqueueNDRangeKernel(ctx->queue, ctx->kernel_compare, 1, NULL, &global_size,
18       &local_size, 0, NULL, NULL);
19   /*Enqueue a non-block transfer for the output*/
20   clEnqueueReadBuffer(ctx->queue, ctx->out, CL_FALSE, 0, 80, output, 0, NULL,
21       &readOutput);
22 ..
```

Figure 9: The OpenCL application model

Since our GPU will execute a queue of kernels that implements PBKDF2, our input is the set of password candidates and the salt. In our application we consider that the user provides the set of password candidates in what we call *password dictionary*. Since the amount of memory on the device is limited it is often impossible to store the entire dictionary on the GPU memory. The first job of the host application is Then to split the dictionary into subsets. The dimension of this subset depends on the amount of the Global Memory the current GPU possess. For simplicity we have considered the passwords as character arrays of length 64 with padding if necessary. Even if this means that we consume a little more memory, this is not usually a problem since HMAC-SHA-1 requires to extend the password to 64 byte [39]: the kernel still needs to use that much memory for each instance.

For the transfert of the password candidates into the device memory we have used

a DMA transfer, as we have showed in 9. If our implementation is using [**OPT–2**], PBKDF2 only needs to access the password buffer only during the first iteration. While the GPU executes the other iterations the host application can send, with a DMA transfer, the next subset of password candidates to the device. Then, when the GPU ends the execution, it will already have the next subset in memory so that it can immediately start the next execution.

```
1  //While we wait, let's prepare the next dictionary
2  memset(ptr,0,64*global_size);
3
4  for(int p = 0; p < global_size*64; p+=64){
5                  if(fscanf(dict,"%s",ptr+p) == EOF){
6                          break;
7                  }
8  }
9
10 /* Must convert all word into big endian */
11 for(int i = 0; i < global_size*16; i++){
12         ((uint32_t*)ptr)[i] = toBE(((uint32_t*)ptr)[i]);
13         ((uint32_t*)ptr)[i] ^= 0x36363636;
14 }
15 //Buffer ready
```

Figure 10: Host precomputation of the dictonary

One of the first problems we had during the development of our host application was the endianess: our machine uses an AMD CPU so the words in memory were in little endian order however HMAC-SHA-1, the PRF used by our implementation of PBKDF2, uses the network endianess [39]. We had to convert the passwords from little endian to big endian before starting the execution of our kernels. We could do the conversion on the host application and then transfert the passwords or we could transfer the passwords and do the conversion on the GPU before starting the execution of PBKDF2 using a specific kernel. We decided to do this conversion on the host application: after some extensive profiling we have noticed that the CPU

was often in idle waiting for the next transfer so we have decided to use this time for convert in network endian the next subset of password candidates. Our profiling also showed that we still had some idle time on the CPU, so we decided to do the first HMAC operation, the XOR of each byte of the password with the byte 0x36, on the host application. When the host application transfers the password candidates subset, these passwords will be already in the correct endianess. Figure 10 shows how the host application does this precomputation of the password candidates subset.

Our kernels executes PBKDF2 on several passwords, in the hope of finding which password generates the key we want to break. After the execution of the kernels, we have on the GPU memory the keys generated from the password candidates subset. We then need to compare the generated key with the one we want to know the password: if this key is in the output of the GPU PBKDF2 kernel, we have found a collision. As for the input, this search can be done on the host or on the GPU itself. We have decided in this case to do the comparison on the GPU. The host application puts the desired key on the constant memory of the GPU. The GPU, after the execution of PBKDF2, executes a little kernel that compare each generated key with the one in the constant memory: if there is a collision, it sets a flag in a little memory buffer with some informations for retrieving the input password from the dictionary. This little buffer is then transferred from the GPU to the host application: since it is a little memory buffer, its transfer time is often negligible. The host application then checks the returned memory: if the flag is set, it can then use the informations in the buffer itself for getting the correct password from the dictionary.

5.1.2 GPU kernel implementation

Our PBKDF2 implementation uses 3 different GPU kernels:

- A kernel that implements the first iteration of HMAC-SHA-1.
- A kernel that implements the remaining $i - 1$ iterations of HMAC-SHA-1.
- A kernel for checking the hash collision.

As we have already described, when we are using the **[OPT–2]** optimization only the first iteration needs to access the password and the salt values: all the other iterations will use the precomputated blocks instead of using the password. The memory buffer containing the password candidates is needed only during the execution of the first iteration. The host can then update the password memory buffer with the next subset of password candidates while the second kernel is executed by the GPU without race conditions problematics, hiding as much as possible the transfer time of the password candidates from the host to the GPU. After we have adopted this configuration our implementation saw a great improvement in the number of computed hashes per second even if we did not actually changed in a significant manner the number of instruction executed.

PBKDF2 uses HMAC and SHA-1 as cryptographic primitives. We have implemented them from scratch on our GPU. As we have described in Chaper 3, one of the biggest differences between a CPU and a GPU architecture is in their memory structure. SHA-1 uses a considerable amount of memory for keeping track of its internal state: we need at least 85 32-bit words for keeping the state between the 80 rounds. This can be a huge problem in an high performance GPU implementation, so memory handling was one of the primary things we concentrate ourselves during the development. In fact, as one can see, **[OPT–4]** optimization gives us a great improvement with some minimal code changes since it can reduces to 21 the number of 32-bit words necessary for keep track of the internal state of SHA-1.

While [**OPT–04**] reduces the number of words necessary for the internal state of SHA-1, [**OPT–02**] presents the problem that each instance must also keep the precomputed values from the first iteration. These values are what we call hash-ipad and hash-opad on Figure 8 and they must be read at each iteration. We experimented several times on where we should keep these 10 words: if in the registers, the Local memory or in the Global memory, maybe even transfer them to the Constant memory inside it. In the end we decided to keep these values in the registers: while it is true that we are using 10 extra registers we have found that this led to the best performances overall.

The implementation of PBKDF2 requires the execution of several iterations of HMAC. While it could be seems logical to implement HMAC ad a single kernel and then put in the execution queue these kernels to implement PBKDF2 we have found that this slows down too much the execution because of the overhead for starting and finishing a kernel. We then use a single kernel that execute a loop of HMAC-SHA-1 execution.

Figure 11 shows an extract of our implementation of HMAC-SHA-1. We have implemented this primitives by inling the SHA-1 instructions in the loop itself. As we have seen in Chapter 3 one of the biggest penalties in execution time on a GPU is presented by conditional jump instructions. In our kernel we are doing a loop, so we have a conditional jump. A way to mitigate this problem is to use loop unrolling: we put the code for several iterations of the loop in the loop itself, reducing the number of conditional jumps required. Often with this techniques it is possible to just avoid the conditional jump instructions altogether but in our case we need to execute too many iterations. The main disadvantage of using loop unrolling is that we need to increase the number of istructions in the compiled kernel, increasing its size. When

```
1  for(int i=1; i < iter_count; i++){
2          W0 = A; W1 = B;
3          W2 = C; W3 = D;
4          W4 = E; W5 = 0x80000000;
5          Wf = 672;
6
7          A = rotAI; B = rotBI;
8
9          SHA_PASSES;
10         SUM_IPAD;
11
12         W0 = A; W1 = B;
13         W2 = C; W3 = D;
14         W4 = E; W5 = 0x80000000;
15         Wf = 672;
16
17         A = rotAO; B = rotBO;
18
19         SHA_PASSES;
20         SUM_OPAD;
21
22         tot[0] ^= A; tot[1] ^= B;
23         tot[2] ^= C; tot[3] ^= D;
24         tot[4] ^= E;
25 }
```

Figure 11: HMAC-SHA-1 GPU implementation

we tried to loop unroll all the iterations of PBKDF2 we obtained a kernel so big that it was not possible to execute on our GPUs. We have found the best performances by putting 3 iterations on the same loop iteration on our main testing GPU, however different GPUs can have a different optimal value for loop unrolling.

5.2 Details of CPU implementation

While it is quite known that a GPU implementation of PBKDF2 can be several times faster than a CPU implementation, we decided to developed one from scratch anyway. We had several reasons behind our decisions: while we wanted to understand

which impact our optimizations had on PBKDF2, it is important to remember that CPU implementations are those used more often by users. Several softwares like cryptsetup [41] for example uses PBKDF2 for deriving the encryption keys using a CPU implementation as a benchmark. This is done for understanding the value the iteration count should have on the current machine, avoiding to use one that can be computed fast enough for providing brute force resistance. If one can accelerate a CPU implementation, these benchmark are no longer significant of the amount of time required to execute an instance of the algorithm.

The CPU implementation was simpler than the GPU implementation: we did not have to work on a multi architecture, client-server application and, because of the importance of PBKDF2, it currently exists a lot of documentations and implementations of this KDF on this kind of architecture. Almost every cryptographic library like OpenSSL [42] or libgcryt [43] provides an implementation of PBKDF2. Some of these implementations, like fastpbkdf2 [44] are specifically developed for acceleration attacks to the algorithm itself. We decided to implement PBKDF2 as a native library and then to develop a little application that uses this library for out testing activities.

While it is common today to have multi-core CPUs on consumer hardware, our implementation only use one of them. We took this decision because PBKDF2 cannot be executed efficiently in a multi-threading environment, especially if we implement [**OPT–1**], except for the computation of the T_i terms, which can be done in parallel. However we have more than one T_i term only if the required key is longer than the internal PRF output length, which is often enough for a lot of use cases. We decided it was better to use a single core for each instance of PBKDF2 and, in case, the main application can execute several instances on different threads.

```
1  void pbkdf2(const char *password, unsigned char* salt,
2                  uint32_t iter, unsigned char* key){
3  ...
4  /* Precompute hash_ipad and hash_opad */
5    memcpy(buffer, password, strlen(key));
6    XOR_BUF(buffer, 0x36);
7    sha1_block(buffer,64, hash_ipad);
8
9    memset(buffer,0,64);
10   memcpy(buffer,password,strlen(password));
11   XOR_BUF(buffer, 0x5C)
12   sha1_block(buffer,64, hash_opad);
13
14 /* Precompute costants from hash_ipad and hash_opad */
15   precompConst(constIpad, hash_ipad);
16   precompConst(constOpad, hash_opad);
17
18 /* Compute U1 */
19   HMAC_SHA(salt, end_hash);
20   memcpy(key, end_hash, 20)
21
22 /* Iterations */
23   for(int i=1;i < iter;i++){
24         HMAC_SHA(end_hash, end_hash);
25         XOR_HASH(key, end_hash);
26   }
27 }
```

Figure 12: PBKDF2 CPU implementation

Figure 12 shows an extract of our CPU implementation of PBKDF2. In this implementation we did not use different kernels or procedures for the implementation but, thanks to the amount of memory we had at our disposal, we were able to introduce as many values precomputation as possible as showed by Figure 12. While it was easier to implement compared to the GPU implementation, we still had a couple of problems to solve. Our CPU application, since it is developed for x86 and x64 architectures, presented the same endianess problem we have discussed before for the GPU implementation. If on the GPU implementation we were able to use the idle

time of the host application to convert the password candidates, on the CPU version we do not have this possibility. In our implementation we opted for precomputed dictionary: before starting the computation the entire dictionary of password candidates is converted into the network endianess. These converted dictionary can then be used by different executions of our CPU implementation, trying to mask the conversion time. This however presents an usability problem, since the dictionary files must be processed before the actual execution.

As for the GPU implementation, the majority of execution time is spent during the execution of the SHA-1 procedure. For this reason, we concentrate a lot on how we should implement this procedure. We wanted something that could achive high performances but it was also easily modifiable for implementing the optimizations we have presented in Chapter 4. We opted for a C implementation, giving us the ability to easily implement our optimizations while having good performances.

Chapter 6

Testing activities

To evaluate the contribution of the optimizations described in Chapter 3, we have:

- implemented from scratch both CPU and GPU version of PBKDF2, following the optimizations presented in Chapters 4 and 5,
- performed our testing activities, measuring PBKDF2 performances, and finally
- compared our results with well-known implementations — e.g. OpenSSL version 1.1.0e [42], libgcrypt version 1.7.6 [43], hashcat 3.5.0 [40].

We have discussed about the implementation of both CPU and GPU version of PBKDF2 in Chaper 5. In this Chapter we present our testing activities, both on GPU and CPU, and the results we have obtained. It is important to notice that during our testing activities we focused not only on raw performances but on understanding which impact each optimization had in our implementation of PBKDF2. In order to show the contribution of the optimizations described in Chapter 4, we implement four different version of our code:

1. based on [**OPT–04**];

2. based on all SHA-1 optimizations ([**OPT–04**], [**OPT–05**], [**OPT–06**]);
3. based on all HMAC and SHA-1 optimizations ([**OPT–02**], ..., [**OPT–06**]);
4. full version ([**OPT–01**], ..., [**OPT–06**]);

All results in this Chapter are based on the same machine, equipped with an AMD FX 8320 8-Core CPU, 8 GB of RAM and running Ubuntu 16.04. For the GPU testing we have used several different GPUs: we will give the details about the hardware while we will present the GPU testing activities. Finally, all the tables assume the following PBKDF2 input parameters:

- iteration count $c = 1,000$ (the minimum value suggested in [7]);
- derived key length $derkey = 256\ bits$;
- a random salt s.

6.1 CPU testing

The main difference between a CPU and GPU implementation is that, in the first one, we have not to transfer a set of candidate passwords from host to device, hence we have not to split the algorithm in two phases as we did on the GPU implementation. In addition, the CPU version implements Equations 7 and 8 as [**OPT–4**] instead of Equation 6. In this case, the circular queue used to implement the word expansion phase of the GPU version does not provide any performance improvement. Indeed, a CPU-based approach has less memory constraints than a GPU-based one. Therefore, the circular queue is a desirable features only in a GPU context.
Table 1 and Figure 13 shows the data collected by running our implementation, OpenSSL version 1.1.0e [42], and Libgcrypt ver.1.7.6 [43].

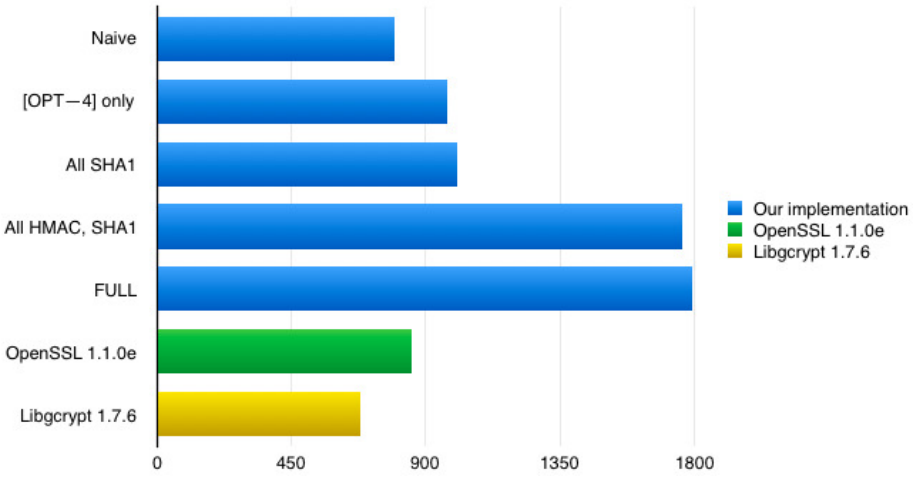

Figure 13: Number of hashes per second on AMD FX 8230

Library	Naive	[OPT–4] only	All SHA-1 opt.	All HMAC, SHA-1 opt.	Full version
Our version	0.797	0.814	1.005	1.761	1.791
OpenSSL ver.1.1.0e					0.851
Libgcrypt ver.1.7.6					0.683

Table 1: Number of Kilohashes per second (KH/s) on CPU

[OPT–4] doesn't present a great contribution to the general performances of the algorithm. All other SHA-1 optimizations brings about a 20% increase in the number of computed hashes. During our testing activities we have noticed that this improvement is primarily caused by **[OPT–6]**, since it reduces the number of CPU instructions required to compute an instance of SHA-1.

As expected, the most important influential optimization are the HMAC one, more precisely **[OPT–2]**. Because of this optimization, the implementation has to execute a little more than half the expected number of SHA-1 block instances. For

this reason, some high performance, CPU-based implementations like fastpbkdf2 [44] already use this optimization.

6.2 GPU testing

For our GPU testing we used the same machine described at the start of this Chapter equipped with 5 different GPUs: AMD R9 390, AMD HD6870, Nvidia GTX 960, Nvidia GTX 1060, and Nvidia GT 670. These are consumer-grade GPUs with different architectures, memory structures and price ranges. Table 2 provides some details of these GPU, like their price and the amount of memory they possess.

GPU	Release date	GPU clock (Mhz)	Memory (MB)	Price ($)
AMD R9 390	18 June 2015	1050	8192	329
AMD HD6870	22 October 2010	900	1024	179
Nvidia GTX 670	10 May 2012	915	2048	399
Nvidia GTX 960	22 January 2015	935	2048	199
Nvidia GTX 1060	16 July 2016	1500	6144	299

Table 2: Details of our GPUs

We used different GPUs for showing the capabilities these hardware have in different price ranges. It's important to notice that, thanks to our OpenCL implementation, we were able to execute our kernels on both AMD and Nvidia GPUs after some little, implementation-specific changes: more precisely, we had to change how the memory pinning is done on the host application for permits to do a DMA transfer of the password dictionary between the CPU and GPU.

Testing results on those GPUs are shown in Table 3 and Figure 14. For each GPU we have executed our implementation for guessing several 6 to 8 character long passwords. We provide the average number of hashes per second our implementation was able to perform. All values in these tables are in Kilohashes (1 KH = 1000 keys) per second (KH/s). Finally, in Table 4 we compare the performance of our code with a well-known password recovery utility [40].

GPU	Naive	[OPT–4] only	All SHA-1 opt.	All HMAC, SHA-1 opt.	Full version
AMD R9 390	244.72	359.56	377.73	755.57	1553.34
AMD HD6870	7.32	98.16	99.18	198.45	398.15
Nvidia GTX 670	75.28	84.14	90.06	191.20	393.83
Nvidia GTX 960	180.32	206.41	212.62	504.06	1048.44
Nvidia GTX 1060	324.26	351.64	381.34	1048.40	1678.30

Table 3: Number of Kilohashes per second (KH/s) on different GPUs

GPU	Full version	hashcat
AMD R9 390	1553.34	1469.6
AMD HD6870	398.15	156.4
Nvidia GTX 670	393.83	410.7
Nvidia GTX 960	1048.44	992.2
Nvidia GTX 1060	1678.30	1710.2

Table 4: Comparison between our implementation and hashcat (KH/s)

Table 3 reports interesting results. First, if we compare these results with those presents in Table 1 we can see that the GPU implementation is around 2000 times faster than the CPU implementation when we consider the number of hashes we can produce and test. This isn't something new since it is known that a GPU architecture is perfect for parallel computations and our problem, the computation and collision checking of hashes, it is extremely easy to execute in parallel. What it is surprising is how old and cheap GPUs are able to compete with modern CPUs even in the naive version. As Table 2 shows us, the AMD HD6870 is the oldest GPU we used for our testing. It was released in 2010 for a 179$ retail price. Even so, this GPU was able to compute almost 4 times more hashes than our AMD FX8320 even if we do not consider any optimization. This shows the amount of performances that GPUs can provide for the right problem.

An interesting results was the impact on performances of [**OPT–4**] on our GPU implementation. This optimization does not reduce the number of operations to execute for the computation: instead it can actually increase them since it means that we cannot implement some of the optimizations presented in [27] that we have discussed in Chapter 2. However this optimization alone provides an increment of around 5% in the number of hashes our implementation can compute. Of particular interesting is the case of the AMD HD6870 with this optimization: because of the small amount of memory equipped on this GPU we were able to have an increment of around 10 times in the number of computed hashes. This is probably because our naive implementation wastes too many registers for the word expansion, causing too many accesses to the slower memory. The other SHA-1 optimizations provided, while not as much as [**OPT–4**], a nice increase in performances, especially [**OPT–06**] because it reduces the number of instruction executed for each iteration of PBKDF2.

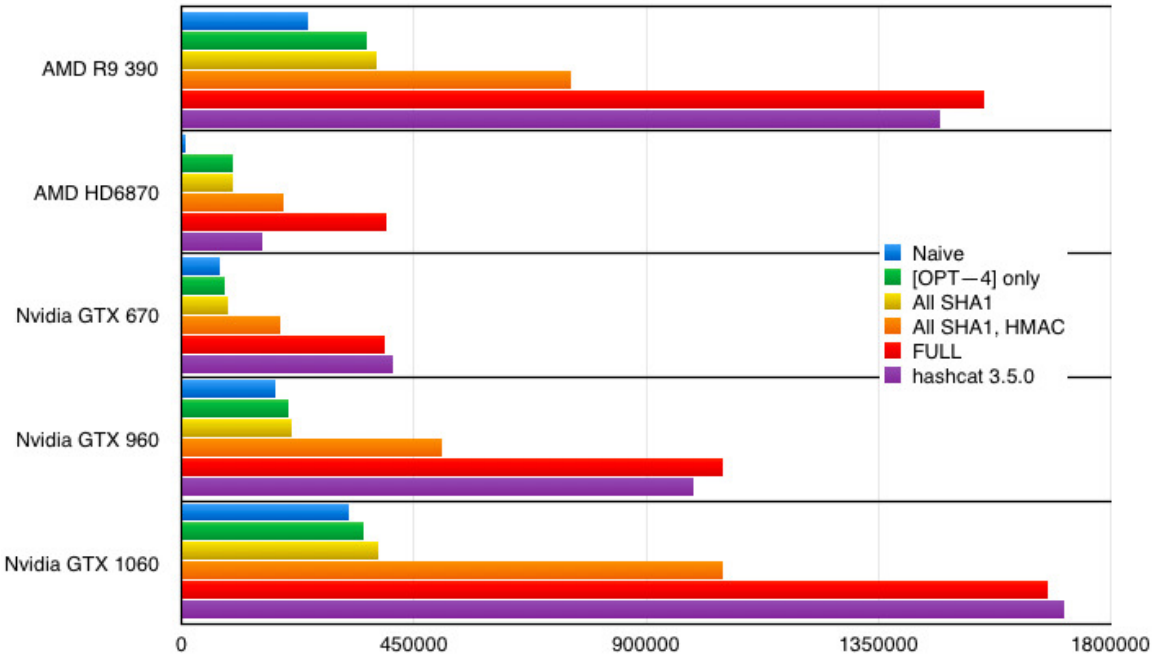

Figure 14: Number of hashes per second on GPU

The major improvement, as expected, was given by [**OPT−2**]. This optimization almost reduces in half the number of SHA-1 blocks the implementation must execute for compute the final key. During development we had some doubts about the effectiveness of this optimization: for using [**OPT−2**], each instance of PBKDF2 must keep the precomputed values, resulting in extra memory consumption. However, as we have described in Chapter 5, we were able to find the best compromise for reducing the memory latency of this version, resulting in a great improvement in the number of computed hashes per second.

Finally, Table 4 presents a comparison between our implementation and hashcat, an open source tool for password cracking. It's worth noting that hashcat is a general-purpose password guesser: it implements several KDF and hash algorithms and it is

often use for password recovery. During our testing activities we have noticed that our implementation is a little faster than hashcat on some GPUs. On our main testing card, the AMD R9 390, our implementation was able to compute around 5% more hashes than hashcat in the same amount of time. We did not explore in details the implementation of PBKDF2-HMAC-SHA-1 provided by hashcat but we can probably trace back this increase in the number of computed hashed to our transferring protocol and the optimizations our implementation is using. It is also worth notice that hashcat probably possess some overheads caused by the many features offered by this application that our implementation does not provide like password masking.

Chapter 7

Conclusions and future works

User-chosen passwords are widely used to protect sensitive information and to gain access to specific resources. They should be strong enough to prevent dictionary and brute-force attacks but usually they are short and lack enough entropy. They cannot be directly used as cryptographic keys. A possible solution to these issues is to adopt a password-based Key Derivation Functions. Currently the Password Based Key Derivation Function v2 [3], defined by RSA Laboratories, is the de-facto standard for KDF. In January 2017 RSA Laboratories updated the PCKS-5 standard [4], which is the current standard for PBKDF2. NIST currently has a standard draft for the storage of passwords and other kind of secrets using PBKDF2 as a response for the numerous password breaches of the last few years [5]. iOS currently uses PBKDF2 for the implementation of the full disk encryption capabilities [9] with an iteration count specific for each device. Android, since version 4.4, uses scrypt instead of PBKDF2 for their full disk encryption. Also, several full disk encryption software like LUKS and FileVault [11] uses a KDF for managing their encryption keys.

In these thesis we have presented a PBKDF2 implementation we have developed

from scratch. We had three main objectives for our implementation:

- Understand which impact several algorithmic optimizations to the internal cryptographic primitives have on the performances of PBKDF2
- Understand the differences between CPUs and GPUs architectures in the context of accelerating KDFs.
- Obtain the fastest possible implementation for PBKDF2.

Our PBKDF2 implementation is able to attack in a few days a 6-7 character long password using a cheap, consumer grade GPU. This means that, even with PBKDF2, the user should use long and complex passwords for avoiding brute force attacks, diminishing the usability gains that the usage of this KDF should give, especially to non technical users. We obtained these results thanks to the several optimization techniques to the main algorithm and its internal cryptographic primitives. We have showed that, in a counterintuitive way, some optimizations brings a great impact only on one architecture and a very modest one on another one: for example, [**OPT–04**] provides a big advantage on the GPU implementation, while it is almost insignificant to the CPU one because of the different memory hierarchy on this architecture.

Even if PBKDF2 received a revision in the January 2017, the security community agrees that PBKDF2 requires an iteration count so big and passwords so long to be considered secure that it can be difficult to use by human users. The problem arises not from a weakness to the algorithm itself or its internal cryptographic primitives but from the idea of using CPU intensive operations for slowing down attackers: the presence of cheap parallel architectures like GPUs and ASICs permits to reduce the cost and the time required for there CPU operations, making the algorithm less resistant to dictionary-based attacks.

As a response to these new architectures, the cryptography field saw a renovated interest in the development of new Key Derivation Functions. In 2009 Colin Percival proposed scrypt [45], a KDF based on a new concept: instead of using CPU-intensive operations as a countermeasure against brute force attacks, scrypt is based on the concept of memory hard operations. In the original definition, a memory hard function is a function that use $S(n)$ memory locations and $T(n)$ instructions on a RAM machine, where:

$$S(n) \in \Omega(T(n)^{(1-\epsilon)})$$

A memory hard algorithm is an algorithm which asymptotically uses at least as many memory locations as instructions. We can see it as an algorithm that uses almost every memory locations it can. The reason behind the development of this kind of functions is because memory is often the most expensive thing to replicate in hardware for general purpose computations. If we took the example of a GPU or an FPGA, it is easy to see that the memory space and speed are a bigger constraint than the number of operations executed per second.

In 2013 the Password Hashing Competition (PHC) [46] was announced. The PHC was an open competition for selecting new password-based hashing and key derivation functions. In 2015 argon2 [47] was announced the winner of the competition, with special mention of the other four finalists: Catena [48], Lyra2 [49], yescrypt [50], Makwa [51] because of their qualities. All these algorithms use sophisticated ideas for preventing brute force attacks by GPU and parallel hardware in general. Argon2, the winner of the competition, is a memory hard function like scrypt but it also presents the possibility to tweak several parameters for better adapting the algorithm to the usage context. MARKVA, instead, uses computations on big integers for making an hardware implementation too much expensive because of the elevated number of

transistor needed. Currently there is not a complete and in-depth analysis of the new algorithms and their brute force resistance algorithms. A possible direction for future works based on this thesis could be the cryptanalysis of these new Key Derivation Functions and their brute force resistance algorithms.

Bibliography

[1] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[2] B Anandampilai and K Haribsakar. *Cryptography and Network Security: A Practical Approach*. Laxmi Publications, 2013.

[3] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, RFC Editor, September 2000.

[4] Burt Kaliski Moriarty, Kathleen and Andreas Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, RFC Editor, January 2017.

[5] Paul A Grassi, James L Fenton, Elaine M Newton, Ray A Perlner, Andrew R Regenscheid, William E Burr, Justin P Richer, Naomi B Lefkovitz, Jamie M Danker, YeeYin Choong, et al. Nist special publication 800 63b digital identity guidelines, June 2017.

[6] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997.

[7] Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen. SP 800-132. Recommendation for Password-Based Key Derivation. Part 1: Storage Applications, December 2010.

[8] IEEE 802.11 WG. Part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Std 802.11 i-2004. 2004.

[9] ios security guide. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`, March 2017.

[10] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1.

[11] Apple Inc. Best Practices for Deploying FileVault 2, 2012.

[12] C. Fruhwirth. LUKS On-Disk Format Specification Version 1.2.2, 2016.

[13] Apache spark: Lightning-fast cluster computing. `http://spark.apache.org`. [Online, accessed 4-February-2018].

[14] Enpass. `https://www.enpass.io`. [Online, accessed 4-February-2018].

[15] F-Secure Key. `https://www.f-secure.com/en/web/home_global/key`. [Online, accessed 4-February-2018].

[16] AgileBits. How PBKDF2 strengthens your Master Password. `https://support.1password.com/pbkdf2/`. [Online, accessed 4-February-2018].

[17] LassPass. Password Iterations (PBKDF2). `https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2/`. [Online, accessed 4-February-2018].

[18] Keeper's Best-In-Class Security. `https://keepersecurity.com/security.html`. [Online, accessed 4-February-2018].

[19] Andrey Belenko and Dmitry Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? *Blackhat Europe*, 2012.

[20] Luca Casati and Andrea Visconti. Exploiting a Bad User Practice to Retrieve Data Leakage on Android Password Managers. In *Proceedings of the 11th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2017*. Springer, 2017.

[21] K Moriarty, B Kaliski, and A Rusch. Pkcs# 5: Password-based cryptography specification version 2.1. Technical report, 2017.

[22] RSA Laboratories. PKCS #5 V2.1: Password Based Cryptography Standard. 2012.

[23] Andrea Visconti, Simone Bossi, Hany Ragab, and Alexandro Caló. On the weaknesses of PBKDF2. In *Proceedings of the 14th International Conference on Cryptology and Network Security, CANS 2015*. Springer International Publishing, LNCS 9476, 2015.

[24] NIST. FIPS PUB 180-4. Secure Hash Standard (SHS). 2012.

[25] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of Advances in Cryptology—CRYPTO96*, pages 1–15. Springer, 1996.

[26] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions—the HMAC construction. *RSA Laboratories CryptoBytes*, 2(1):12–15, 1996.

[27] Andrea Visconti and Federico Gorla. A further weakness of PBKDF2. `https://eprint.iacr.org/2018/097.pdf`.

[28] Mark Harris. Gpgpu: General-purpose computation on gpus. *SIGGRAPH 2005 GPGPU COURSE*, pages 1–51, 2005.

[29] Christopher Cullinan, Christopher Wyant, Timothy Frattesi, and Xinming Huang. Computing performance benchmarks among cpu, gpu, and fpga. page 55, 2013.

[30] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[31] CUDA Nvidia. Programming guide, 2010.

[32] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[33] Khronos group. `https://www.khronos.org/`.

[34] John Kessenich. An introduction to spir-v. 2015.

[35] L Hennessy John and A Patterson David. Computer organization & design: The hardware/software interface, 1999.

[36] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[37] Andrew Ruddick and Jeff Yan. Acceleration attacks on pbkdf2: or, what is inside the black-box of oclhashcat? In *10th USENIX Workshop on Offensive Technologies*, 2016.

[38] Jens Steube. Optimising Computation of Hash-Algorithms as an Attacker. `https://hashcat.net/events/p13/js-ocohaaaa.pdf`. [Online, accessed 4-February-2018].

[39] NIST. FIPS PUB 198-1. The Keyed-Hash Message Authentication Code (HMAC). July 2008.

[40] hashcat. `https://hashcat.net/hashcat/`. [Online, accessed 4-February-2018].

[41] Milan Brož. Cryptsetup 2.0.1 Release Notes. `https://www.kernel.org/pub/linux/utils/cryptsetup/v2.0/v2.0.1-ReleaseNotes`, 2018. [Online, accessed 4-February-2018].

[42] Openssl. `https://www.openssl.org/`. [Online, accessed 4-February-2018].

[43] Libgcrypt. `https://www.gnupg.org/software/libgcrypt/index.html`.

[44] fastpbkdf2. `https://github.com/ctz/fastpbkdf2`. [Online, accessed 4-February-2018].

[45] Colin Percival. Stronger key derivation via sequential memory-hard functions. `https://www.tarsnap.com/scrypt/scrypt.pdf`, May 2009. [Online, accessed 4-February-2018].

[46] Password hashing competition. `https://password-hashing.net/`, 2015.

[47] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 (version 1.2). University of Luxembourg, Luxembourg, July 2015.

[48] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena : A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525, 2013.

[49] Marcos A Simplicio Jr, Leonardo C Almeida, Ewerton R Andrade, Paulo CF dos Santos, and Paulo SLM Barreto. Lyra2: Password hashing scheme with improved security against time-memory trade-offs. Cryptology ePrint Archive, Report 2015/136, 2015.

[50] Alexander Peslyak. yescrypt – password hashing scalable beyond bcrypt and scrypt. May 2014.

[51] Thomas Pornin. The MAKWA Password Hashing Function. April 2015. [Online, accessed 4-February-2018].