Understanding Optimizations and Measuring Performances of PBKDF2 [⋆]

Andrea Francesco Iuorio and Andrea Visconti[⋆⋆]

Department of Computer Science,
Università degli Studi di Milano
andreafrancesco.iuorio@studenti.unimi.it
andrea.visconti@unimi.it
http://www.di.unimi.it/visconti

Abstract. Password-based Key Derivation Functions (KDFs) are used to generate secure keys of arbitrary length implemented in many security-related systems. The strength of these KDFs is the ability to provide countermeasures against brute-force/dictionary attacks. One of the most implemented KDF is PBKDF2. In order to slow attackers down, PBKDF2 uses a salt and introduces computational intensive operations based on an iterated pseudo-random function. Since passwords are widely used to protect personal data and to authenticate users to access specific resources, if an application uses a small iteration count value, the strength of PBKDF2 against attacks performed on low-cost commodity hardware may be reduced. In this paper we introduce the cryptographic algorithms involved in the key derivation process, describing the optimization techniques used to speed up PBKDF2-HMAC-SHA1 in a GPU/CPU context. Finally, a testing activities has been executed on consumer-grade hardware and experimental results are reported.

Keywords: passwords , PBKDF2 , HMAC-SHA1 , optimizations , CPU-intensive operations , performance testing

1 Introduction

Although user-chosen passwords are often too short and lack enough entropy [1], they are still widely used for authentication purposes, thus making them vulnerable to brute force or dictionary attacks. A possible solution to this issue is to adopt a Key Derivation Function (KDF) which inputs a key material and generates a secure key [2]. In particular, [3] provides recommendations for the implementation of PBKDF2, a KDF which inputs a user-chosen password.

[⋆] A slightly different version of this paper appeared in the Proceedings of the 2nd International Conference on Wireless, Intelligent and Distributed Environment for COMmunication (WIDECOM 2019), Springer International Publishing, Lecture Notes on Data Engineering and Communications Technologies, Vol. 27, 2019, https://doi.org/10.1007/978-3-030-11437-4.

[⋆⋆] Corresponding author

Even though in 2015 the *Password Hashing Competition* [4] selected a number of hashing schemes — e.g., Argon2 [5] (the winner of the competition), Catena [6], Lyra2 [7], yescrypt [8], and Makwa [9] — currently, PBKDF2 [3] still remains the most widely implemented and used in practice. For example, it is used in WiFi Protected Access [10], iOS passcodes [11], LUKS [12], and many others. In addition, KDF has been usesd in Mobile Adhoc Network for securing the Zone Routing Protocol [13]. 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 [14,15,16,17,18] are based on PBKDF2 and a number of security and privacy concerns have to be addressed [19,20,21].

For slowing attackers down, PBKDF2 uses a random salt and an iteration count. The latter specifies the number of times a pseudo-random function (PRF) 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 [22], 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 sets the iteration count value to 4096 [10] — while others do not — e.g., the iteration count associated with iOS passcodes is calibrated to take about 80 milliseconds [11].

In this paper, we focus on PBKDF2-HMAC-SHA-1, presenting the state-of-the-art research results achieved in the last five years. In particular, we introduce all cryptographic algorithms involved in the key derivation process. Then, we describe the optimization techniques used to speed up PBKDF2, HMAC and SHA-1 in a GPU/CPU context. Finally, in order to measure the contributions provided by these optimizations, we develop an implementation of PBKDF2-HMAC-SHA-1 from scratch, execute our testing activities on consumer-grade hardware, and present the experimental results found.

The paper is organized as follows. In Section 2 we introduce the cryptographic algorithms involved in the key derivation process. In Section 3, we describe several optimizations published in literature that can be used to speed up a PBKDF2 implementation. In Section 4, we present our testing activities, describing both CPU and GPU implementations and showing the experimental result found. Finally, conclusions are drown in Section 5.

2 Cryptographic preliminaries

2.1 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 focus on PBKDF2-HMAC-SHA-1. Although in 2017 first practical technique for generating a collision of SHA-1 has been presented, HMAC-SHA-1 is still considered secure.

PBKDF2 is a Password-Based Key Derivation Function described in PKCS #5 [3], [22]. For providing better resistance against brute force attacks, PBKDF2 introduces 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

$$\begin{aligned} T_1 &= Function(p, s, c, 1), \\ T_2 &= Function(p, s, c, 2), \\ &\vdots \\ T_{len} &= Function(p, s, c, len). \end{aligned}$$

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

$$\begin{aligned} U_1 &= PRF(p, s||i), \\ U_2 &= PRF(p, U_1), \\ &\vdots \\ U_c &= PRF(p, U_{c-1}). \end{aligned}$$

The PRF adopted can be a hash function [24], cipher, or HMAC [25], [26], and [27]. In the sequel, we will refer to HMAC as PRF.

2.2 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 [27] requires

- H: a cryptographic hash function;
- K: the secret key;

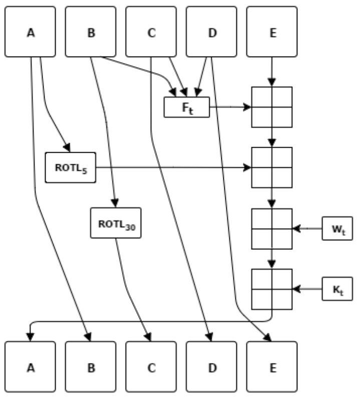

Fig. 1. A graphical representation of the SHA-1 algorithm

– *text*: the message to be authenticated.

As described in RFC 2104 [27], an HMAC can be defined as follows:

$$HMAC = H(K \oplus opad, H(K \oplus ipad, text)) \qquad (4)$$

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.

2.3 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 1. Notice that F_t is defined by

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

and K_t assume four constants value[1].

[1] In this section, we partially describe the SHA-1 algorithm. Further details can be found 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:

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

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 [28]. 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 (we are reducing the number of XORs) 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, [28] states that Equation 6 can be replaced with the unfolded version:

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

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.

3 Understanding Optimizations

PBKDF2 applies a pseudorandom function to generate cryptographically secure keys. Since in this process different cryptographic algorithms are involved (see Section 2), the optimization of one of these algorithms usually leads 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 section we briefly present the optimizations resulting to significant improvements.

3.1 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 fingerprints 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

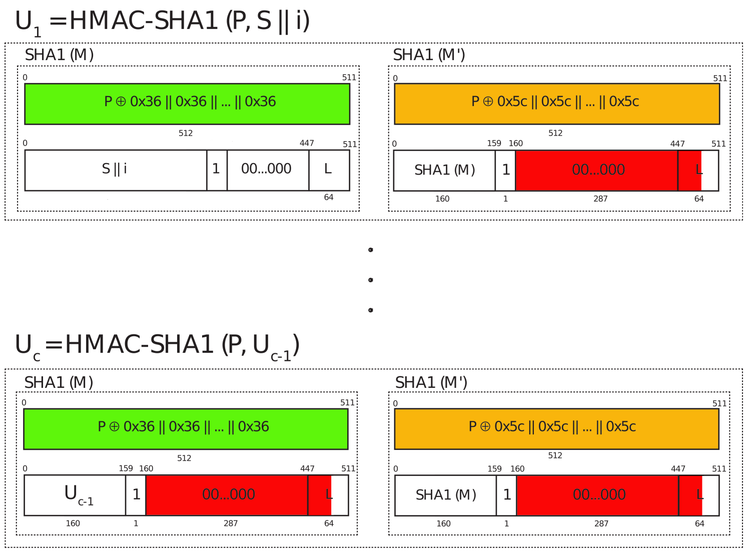
U_1 =HMAC-SHA1 (P, S || i)
SHA1 (M)
P ⊕ 0x36 || 0x36 || ... || 0x36
512
S || i
1
00...000
L
64
SHA1 (M')
P ⊕ 0x5c || 0x5c || ... || 0x5c
SHA1 (M)
00...000
160
287
U_c =HMAC-SHA1 (P, U_{c-1})
U_{c-1}

Fig. 2. PBKDF2-HMAC-SHA-1 optimizations

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 [29].

3.2 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 (light gray rectangles of Figure 2) and reuse such a value 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 [30,23,29].

[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 [31]. 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 2), we have not a generic HMAC implementation but a specific one. Indeed, we know 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 2), we have not to split the message to be authenticated in blocks. Therefore, this optimization provides 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 [30].

3.3 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 [28], 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 reduces 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 [28] 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 avoid 66 out of 192 XOR operations [28].

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 [30]. 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 1). 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.

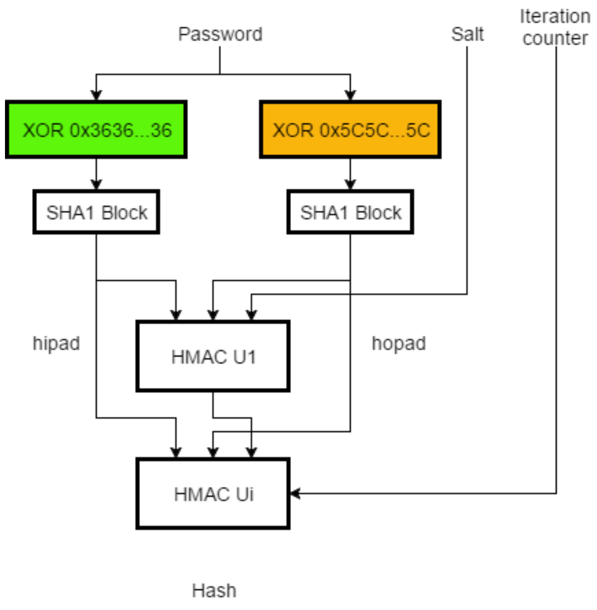

Fig. 3. PBKDF2 GPU implementation

4 Measuring Performances

To evaluate the contribution of the optimizations described in Section 3, we (a) implemented from scratch both CPU and GPU version of PBKDF2, (b) performed our testing activities, measuring PBKDF2 performances, and finally (c) compared our results with well-known implementations — e.g. OpenSSL version 1.1.0e [32], libgcrypt version 1.7.6 [33], hashcat 3.5.0 [34].

4.1 GPU testing

In order to run the same code on several devices with different architectures, our implementation has been written using the OpenCL framework [35]. The implementation uses a classic host-device approach. The host (a CPU) generates a set of passwords and sends them to the device (a GPU). In order to exploit [OPT–03], our code executes PBKDF2 as a two-step process (see Figure 3): firstly it computes U_1, storing the intermediate results (*hipad* and *hopad*) of the compression function — i.e., we are computing the light gray rectangles of Figures 2 and 3 — and secondly computes the remaining $U_2, \ldots, U_c$. Doing so, we can compute U_1 with a generic HMAC implementation and the remaining U_i with a specific one, thus avoiding length checks and the chunk splitting operations. In addition, after the computation of U_1, the CPU is able to transfer (asynchronously) a new set of candidate passwords to the GPU, reducing the overhead generated by read/write memory operations.

Table 1. Number of Kilohashes per second (KH/s) on different GPUs

GPU	Naive	[OPT–4] only	All SHA-1 opt.	All HMAC, SHA-1 opt.	Full version	hashcat
AMD R9 390	244.72	359.56	377.73	755.57	1553.34	1469.6
AMD HD6870	7.32	98.16	99.18	198.45	398.15	156.4
Nvidia GTX 670	75.28	84.14	90.06	191.20	393.83	410.7
Nvidia GTX 960	180.32	206.41	212.62	504.06	1048.44	992.2
Nvidia GTX 1060	324.26	351.64	381.34	1048.40	1678.30	1710.2

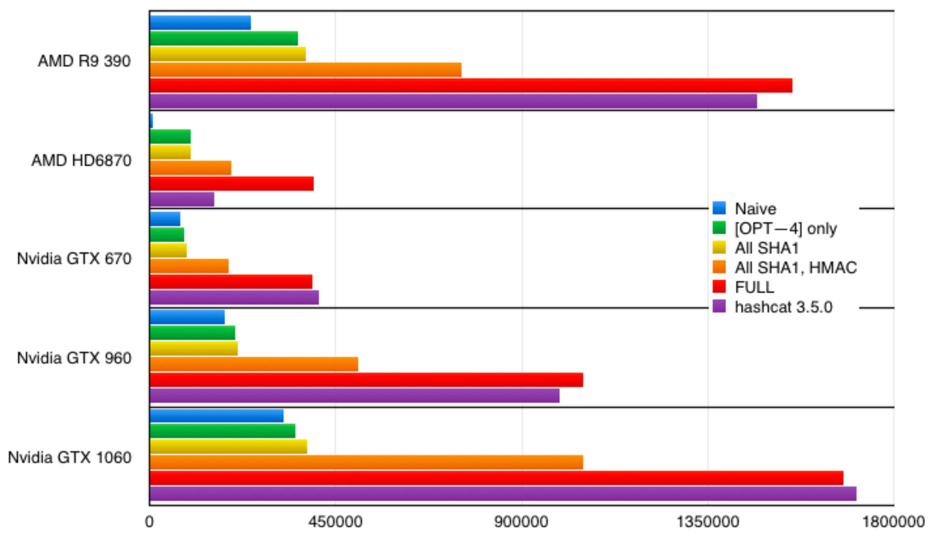

Fig. 4. Number of hashes per second on GPU

All the tests were executed on a machine equipped with an AMD FX 8320 4GHz processor, 8 GB RAM, Microsoft Windows 10 Home 64-Bit Operating System. In addition, we installed five consumer-grade GPUs with different architectures, memory structures and price ranges: AMD R9 390, AMD HD6870, Nvidia GTX 960, Nvidia GTX 1060, and Nvidia GTX 670.

In order to show the contribution of the optimizations described in Section 3, we implement four different versions 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]);

Then, we set the following PBKDF2 input parameters:

– iteration count $c = 1,000$ (the minimum value suggested in [22]);

Table 2. Number of Kilohashes per second (KH/s) on CPU

Library	Naive	[OPT–4] only	All SHA-1 opt.	All HMAC, SHA-1 opt.	Full version
Our version	0.797	0.974	1.005	1.761	1.791
OpenSSL ver.1.1.0e	-	-	-	-	0.851
Libgcrypt ver.1.7.6	-	-	-	-	0.683

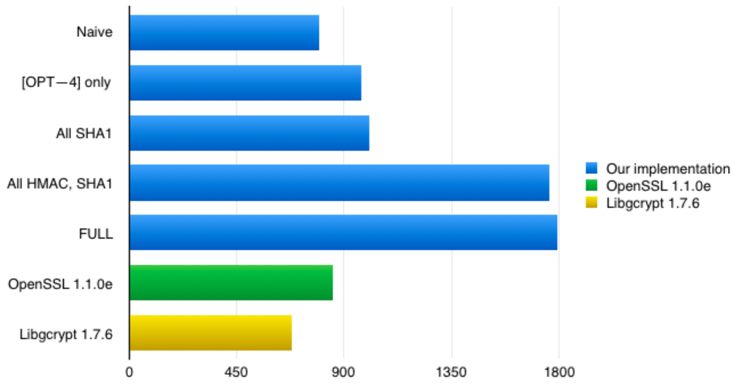

Fig. 5. Number of hashes per second on AMD FX 8230

– derived key length $derkey = 256\ bits$;
– a random salt s.

Finally, we run our implementations and collected data. Testing results are shown in Table 1 and Figure 4. Then, we compare the performance of our code with a well-known password recovery utility [34].

4.2 CPU testing

Since real-world applications may define an appropriate iteration count value at runtime by executing a CPU-test performance — e.g. LUKS [12,36] — we also implemented a CPU-based version of PBKDF2. 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 shown in Figure 3. 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 no memory constraints, while GPU-based has. Therefore, the circular queue is a desirable feature only in a GPU context.

Our testing activities were executed on an AMD FX-8320 8-Core 4 Ghz, setting iteration count c, derived key length $derkey$, and salt s as described in

Section 4.1. Table 2 and Figure 5 show the data collected by running our implementation, OpenSSL version 1.1.0e [32], and Libgcrypt ver.1.7.6 [33].

5 Conclusions

User-chosen passwords are widely used to protect our sensitive information and to gain access to specific resources. They should be strong enough to prevent dictionary and brute-force attacks but usually are short and lack enough entropy and cannot be directly used as keys. A possible solution to these issues is to adopt a password-based Key Derivation Functions. Although Argon2 [5], Catena [6], Lyra2 [7], yescrypt [8], Makwa [9] and scrypt [37] are expected to supersede PBKDF2 in the next years, currently PBKDF2 is still widely used to derive keys in many security applications. In this paper we described the state-of-the-art of PBKDF2 with the aim to raise the reader's awareness of security issues facing new applications — for example, it is not difficult to find apps on Android market which implement PBKDF2 with poor security parameters. Thus, focusing on PBKDF2-HMAC-SHA-1, we described and tested a number of optimization techniques used by malicious users to speed up PBKDF2, HMAC and SHA-1. In addition, we showed that these optimizations should be implemented in crypto libraries in order to speed up the performances, and accordingly, increasing the level of security of those applications which define the iteration count at runtime.

An interesting future work will be to analyse the performance of these optimizations on several graphics cards equipped with different chips from fastest to slowest.

References

1. Shannon, C.E.: Prediction and entropy of printed english. Bell system technical journal **30**(1), 50–64 (1951)
2. Krawczyk, H.: Cryptographic Extraction and Key Derivation: The HKDF Scheme. Cryptology ePrint Archive, Report 2010/264 (2010)
3. Moriarty, K., Kaliski, B., Rusch, A.: PKCS# 5: Password-Based Cryptography Specification Version 2.1. RFC 8018 (2017)
4. Password hashing competition. `https://password-hashing.net/`. Cited 10 Nov 2018
5. Biryukov, A., Dinu, D., , Khovratovich, D.: Argon2 (version 1.2). University of Luxembourg, Luxembourg. `https://password-hashing.net/submissions/specs/Argon-v3.pdf`. Cited 10 Nov 2018
6. Forler, C., Lucks, S., Wenzel, J.: Catena : A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525 (2013)
7. Simplicio Jr, M.A., Almeida, L.C., Andrade, E.R., dos Santos, P.C., Barreto, P.S.: Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs. Cryptology ePrint Archive, Report 2015/136 (2015)
8. Peslyak, A.: yescrypt – password hashing scalable beyond bcrypt and scrypt. Openwall, Inc. (2014). `http://www.openwall.com/presentations/PHDays2014-Yescrypt/`. Cited 10 Nov 2018

9. Pornin, T.: The MAKWA Password Hashing Function (2015). http://www.bolet.org/makwa/makwa-spec-20150422.pdf. Cited 10 Nov 2018
10. Wi-Fi Alliance: Discover Wi-Fi: Specifications. https://www.wi-fi.org/discover-wi-fi/specifications. Cited 10 Nov 2018
11. iOS Security Guide (2017). https://www.apple.com/business/docs/iOS_Security_Guide.pdf. Cited 10 Nov 2018
12. Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.2.2 (2016). https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf. Cited 10 Nov 2018
13. Rahman, M.T., Mahi, M.J.N.: Proposal for SZRP protocol with the establishment of the salted SHA-256 Bit HMAC PBKDF2 advance security system in a MANET. In: 2014 International Conference on Electrical Engineering and Information Communication Technology, pp. 1–5 (2014)
14. Enpass. https://www.enpass.io. Cited 10 Nov 2018
15. F-Secure Key. https://www.f-secure.com/en/web/home_global/key. Cited 10 Nov 2018
16. AgileBits: How PBKDF2 strengthens your Master Password. https://support.1password.com/pbkdf2/. Cited 10 Nov 2018
17. LassPass: Password Iterations (PBKDF2). https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2/. Cited 10 Nov 2018
18. Keeper: Keeper's Best-In-Class Security. https://keepersecurity.com/security.html. Cited 10 Nov 2018
19. Belenko, A., Sklyarov, D.: "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Blackhat Europe (2012)
20. Casati, L., Visconti, A.: 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. Casati, L., Visconti, A.: The Dangers of Rooting: Data Leakage Detection in Android Applications. Mobile Information Systems, Article ID 6020461 (2018). DOI 10.1155/2018/6020461
22. Turan, M.S., Barker, E.B., Burr, W.E., Chen, L.: SP 800-132. Recommendation for Password-Based Key Derivation. Part 1: Storage Applications (2010). http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf. Cited 10 Nov 2018
23. Visconti, A., Bossi, S., Ragab, H., Caló, A.: 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). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Cited 10 Nov 2018
25. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Proceedings of Advances in Cryptology—CRYPTO96, pp. 1–15. Springer (1996)
26. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions—the HMAC construction. RSA Laboratories CryptoBytes **2**(1), 12–15 (1996)
27. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104
28. Visconti, A., Gorla, F.: Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2. IEEE Transactions on Dependable and Secure Computing (2018). DOI 10.1109/TDSC.2018.2878697

29. Ruddick, A., Yan, J.: Acceleration attacks on PBKDF2: or, what is inside the black-box of oclHashcat? In: Proceedings of the 10th USENIX Workshop on Offensive Technologies (2016)
30. Steube, J.: Optimising Computation of Hash-Algorithms as an Attacker. https://hashcat.net/events/p13/js-ocohaaaa.pdf. Cited 10 Nov 2018
31. NIST: FIPS PUB 198-1. The Keyed-Hash Message Authentication Code (HMAC) (2008). http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf. Cited 10 Nov 2018
32. Openssl, version: 1.1.0e. https://www.openssl.org/. Cited 10 Nov 2018
33. Libgcrypt, version 1.7.6. https://www.gnupg.org/software/libgcrypt/index.html. Cited 10 Nov 2018
34. hashcat, version 3.30. https://hashcat.net/hashcat/. Cited 10 Nov 2018
35. OpenCL. https://www.khronos.org/opencl/. Cited 10 Nov 2018
36. Bossi, S., Visconti, A.: What users should know about Full Disk Encryption based on LUKS. In: Proceedings of the 14th International Conference on Cryptology and Network Security, CANS 2015. Springer International Publishing, LNCS 9476 (2015)
37. Percival, C.: Stronger key derivation via sequential memory-hard functions (2009). https://www.tarsnap.com/scrypt/scrypt.pdf. Cited 10 Nov 2018