Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2

Andrea Visconti and Federico Gorla

Department of Computer Science "Giovanni degli Antoni",
Università degli Studi di Milano
andrea.visconti@unimi.it
federico.gorla@studenti.unimi.it
http://www.di.unimi.it/visconti

Abstract. PBKDF2 [27] is a well-known password-based key derivation function. In order to slow attackers down, PBKDF2 introduces CPU-intensive operations based on an iterated pseudorandom function (in our case HMAC-SHA-1). If we are able to speed up a SHA-1 or an HMAC implementation, we are able to speed up PBKDF2-HMAC-SHA-1. This means that a performance improvement might be exploited by regular users and attackers. Interestingly, FIPS 198-1 [31] suggests that it is possible to precompute first message block of a keyed hash function only once, store such a value and use it each time is needed [43]. Therefore the computation of first message block does not contribute to slowing attackers down, thus making the computation of second message block crucial. In this paper we focus on the latter, investigating the possibility to avoid part of the HMAC-SHA-1 operations. We show that some CPU-intensive operations may be replaced with a set of equivalent, but less onerous, instructions. We identify useless XOR operations exploiting and extending Intel optimizations [26], and applying the Boyar-Peralta heuristic [12]. In addition, we provide an alternative method to compute the SHA-1 message scheduling function and explain why attackers might exploit these findings to speed up a brute force attack.

Keywords: HMAC-SHA-1, Password-Based Key Derivation Function 2, PKCS#5, Intel optimizations, Boyar-Peralta heuristic

1 Introduction

When faced with the problem of accessing a protected resource, many applications ultimately rely on the knowledge of one or more secrets. Such secrets are commonly passwords or passphrases that can lead to a number of security issues. Firstly, giving the user the choice of selecting the secret to be used, usually results in a very "humanly predictable" secret being chosen. The user will most likely choose a short, and easy to remember, passphrase which might undermine the security of the system to be protected [14], [39], [7], [19]. Secondly, password-based authentication can often be effectively attacked by employing techniques such as exhaustive search and dictionary attacks.

In order to avoid the use of user-chosen passwords as a key to cryptographic systems, a number of approaches have been developed. In particular, we are interested in password-based Key Derivation Functions, a set of algorithms which input a user-chosen password and provide a stream of pseudorandom bits presenting enough entropy and an adequate length to be used as key in real-world applications.

Password-based KDFs introduce CPU/memory intensive operations which avoid, or make dictionary and brute force attacks less feasible. At the same time, such operations are not so expensive to become a burden for a regular user.

In 2013, a Password Hashing Competition (PHC) [1] was held to develop a number of resistant password-based key derivation functions. PHC selected a winner (Argon2 [6]) and gave a special recognition to four algorithms (Catena [18], Lyra2 [40], yescrypt [34] and Makwa [35]). Although Argon2 is expected to be the password-based KDF of the next years, currently one of the most widely used functions is PBKDF2 [37]. Described by RSA Labs in PKCS#5 [37], PBKDF2 has been implemented in several real-world applications such as WPA/WPA2 [25], 1Password [2], Keeper [28], LUKS [20] [9], LassPass [30], Codebook [16], GRUB2[24], RAR archive format [36], FileVault Mac OS X [3], [15], Android's full disk encryption (since version 3.0 Honeycomb to 4.3 Jelly Bean), and many others. PBKDF2 uses a *salt* to prevent the construction of universal dictionaries and an *iteration count* to accurately tune the tradeoff between user perceived slowness of the key derivation process and bruteforce attack resistance [42].

Most of the PBKDF2 computations are performed by a pseudorandom function, and in this paper we focus on HMAC-SHA-1. Notice that not all the applications previously mentioned use PBKDF2-HMAC-SHA1 by default. For example, 1Password (Agile Keychain format) [23]) does, while 1Password (OPVault format) [22] does not. Cryptsetup version 1.6.8 (and above) does, while versions 1.7.0–1.7.5 do not and the hash function used is SHA-256 [13]. RAR (previous to version 5) does, while versions 5.00–5.50 do not and the pseudorandom function is HMAC-SHA-256 [36]. GRUB2 [24] implements PBKDF2-HMAC-SHA512 while Codebook [16] uses PBKDF2-HMAC-SHA-1.

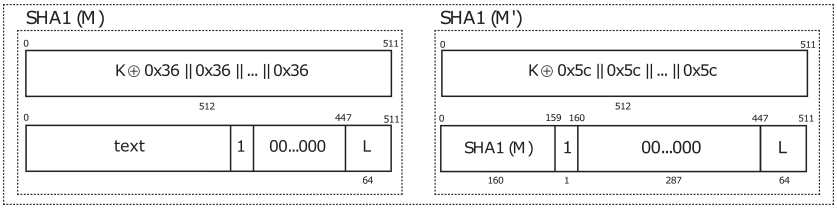

Fig. 1. A graphical representation of HMAC-SHA-1

If we are able to speed up SHA-1 or HMAC-SHA-1, we are able to speed up PBKDF2. Interestingly, when the HMAC function is implemented without following the performance improvement described in [29] and [31], it is possible to avoid 50% of PBKDF2's CPU intensive operations involved in the key derivation process. Moreover, as described in [41], [43] and [38], other critical flaws might be exploited, thus reducing the total number of CPU-intensive operations to be performed for computing a key. Therefore, the only requirement to maintain security is to increase the number of computations through the iteration count parameter of PBKDF2.

In this paper, we describe an HMAC-SHA-1 optimization which can be used to speed up PBKDF2. Focusing on the computation of second message block of HMAC-SHA-1, we investigate the possibility to avoid part of the CPU-intensive operations by executing a set of equivalent, but less onerous, instructions. By doing so, we identify useless XOR operations exploiting and extending Intel optimizations [26] and applying the Boyar-Peralta heuristic [12]. In addition, we provide an alternative method to compute the SHA-1 message scheduling function and explain why attackers might exploit these findings to speed up a brute force attack against PBKDF2-HMAC-SHA-1.

The remainder of this paper is organized as follows. In Section 2 we briefly introduce the Hash-based Message Authentication Code (HMAC) algorithm, referring in particular to HMAC-SHA-1. Password-based Key Derivation Function version 2 (PKBDF2) is described in Section 3. In Section 4 we present the original contribution of this paper, showing how performance improvements of SHA-1 and HMAC-SHA-1 might be exploited to speed up PBKDF2-HMAC-SHA-1. Finally, discussion and concluding remarks are drawn in Section 5.

2 HMAC-SHA-1

A Hash-based Message Authentication Code is an algorithm for computing a message authentication code based on any iterated cryptographic hash function. The definition of HMAC [29] requires

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

As described in RFC 2104 [29], an HMAC can be defined as follows:

$$HMAC = H(K \oplus opad \mid\mid H(K \oplus ipad \mid\mid text)) \tag{1}$$

where H is the chosen hash function, K is the secret key, $\oplus$ is the exclusive OR symbol, $\mid\mid$ is the concatenation symbol and *ipad*, *opad* are constant values — respectively, the byte 0x36 and 0x5C repeated 64 times. In order to better understand Equation 1, we can expand it in the form:

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

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

In this paper we will refer to HMAC-SHA-1 which is the default as per [37]. Therefore Equation 1 can be graphically represented as in Figure 1.

Readers who are not familiar with SHA-1 may find a detailed description of the cryptographic algorithm in [32] and [17]. However, it is worth recalling some basic concepts:

- SHA-1 [32] processes blocks of the size of 512 bits — i.e., sixteen 32-bit words $W_0, \ldots, W_{15}$,
- it iterates for 80 rounds in order to produce a 160-bit message digest,
- the original message is padded with one bit **1** first then, zero or more bits **0** so that its length is congruent to 448, modulo 512, and
- the last 64 bits of the last 512-bit block represent the message length L.

In addition, the SHA-1 algorithm uses the following message scheduling function to expand $W_0, \ldots, W_{15}$ into eighty words:

$$W[i] = ROTL^1(W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) \qquad i \in [16 \ldots 79] \tag{2}$$

This function requires to store eighty 32-bit words, therefore if memory is limited an alternative method should be adopted. In [32], NIST suggests to regard $W_0, \ldots, W_{15}$ as a circular queue and substitute the Equation 2 with the following:

$$\begin{aligned} &s \leftarrow i \wedge MASK \qquad i \in [16 \ldots 79] \\ &W[s] \leftarrow ROTL^1(W[s] \oplus W[(s+2) \wedge MASK] \oplus W[(s+8) \wedge MASK] \oplus W[(s+13) \wedge MASK]) \end{aligned} \tag{3}$$

where $MASK$ is set to the value `0000000f` (in hex), thus performing arithmetic modulo 16. This new equation requires only sixteen words, thus saving sixty-four 32-bit words of storage.

3 PBKDF version 2

Password Based Key Derivation Function version 2, PBKDF2 for short, is a key derivation function published by RSA Laboratories in PKCS #5 [37]. In order to face brute force attacks based on weak user passwords, PBKDF2 introduces CPU-intensive operations. Such operations are based on an iterated pseudorandom function which maps input values to a derived key. PBKDF2 is fully defined by the following parameters:

- a pseudorandom function (PRF) — e.g. a hash function, cipher, or HMAC;
- an iteration count c;
- a password p;
- a salt s;

– a desired output length $dkLen$.

and outputs a derived key DK.

PBKDF2 can derive keys of arbitrary length. More precisely, PBKDF2 generates as many blocks T_i as needed to cover the desired key length. Each block T_i is computed iterating the PRF as many times as specified by an iteration count c. NIST suggests to select the iteration count as large as possible, however a minimum of 1,000 iterations is recommended for general purpose applications while 10,000,000 may be appropriate for protecting very critical keys [42]. The length of each block T_i is bounded by $hLen$, which is the length of the underlying PRF output. This PRF can be a hash function [32], cipher, or HMAC [4], [5], [29]. Since the optimizations described in this paper only applies to HMAC-SHA-1, we use this HMAC as PRF.

PBKDF2 inputs a user password/passphrase p, a random salt s, an iteration counter c, and derived key length $dkLen$. It outputs a derived key DK.

$$DK = PBKDF2(p, s, c, dkLen)$$

The derived key is defined as the concatenation of $\lceil dkLen/hLen \rceil$-blocks:

$$DK = T_1 || T_2 || \ldots || T_{\lceil dkLen/hLen \rceil}$$

where

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

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

$$\vdots$$

$$T_i = Function(p, s, c, i)$$

$$\vdots$$

$$T_{\lceil dkLen/hLen \rceil} = Function(p, s, c, \lceil dkLen/hLen \rceil).$$

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

$$T_i = U_1 \oplus U_2 \oplus \ldots \oplus U_c$$

where (see Figure 2)

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

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

$$\vdots$$

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

Fig. 2. A real-world implementation [19][20] of PBKDF2 based on HMAC-SHA-1

4 Avoiding useless XOR operations

As shown in [41], [43] and [38], several performance improvements can be exploited to speed up the computation of PBKDF2 — e.g., precomputing the first message block of a keyed hash function, avoiding constant additions modulo 2^{32} in the first three rounds, avoiding SHA-1 length checks of input parameters, avoiding SHA-1 chunk splitting operations during the computation of $U_2,\ldots,U_c$, and so on. In this section we present a new improvement. In particular, we show how useless operations, computed by HMAC-SHA-1 and repeated c times by PBKDF2, might be exploited by users (regular or malicious).

When the HMAC function is computed in a standard mode — i.e., without following the performance improvements described in the implementation note

of RFC 2104 [29] and FIPS 198-1 [31] — attackers can precompute first message block of a keyed hash function only once (green and orange rectangles in Figure 2), store such a value and use it each time is needed. This suggests that computation of the first message block does not contribute to slowing down a brute force attack, thus the computation of second message block is crucial.

Therefore we focus on the second message block with the aim to show how users might avoid part of CPU-intensive operations involved in the key derivation process. Doing so, we will exploit and extend Intel optimizations [26] and the Boyar-Peralta heuristic [12].

4.1 Exploiting Intel optimizations

As shown in [8], the SHA-1 algorithm is a vastly parallel algorithm in terms of instruction level parallelism allowing it to fully employ several ALU pipelines but it seems like it is not well suited to work with Single Instruction Multiple Data (SIMD) instructions.

One of the most interesting optimizations to SHA-1 has been described by Max Locktyukhin in [26]. Improving an idea by Dean Gaudet [21], the Intel researcher [26] shows how the SHA-1 hash algorithm can be implemented in order to take full advantage of SIMD instructions, providing a new equation to expand the sixteen 32-bit words $W[0], \ldots, W[15]$ into eighty words. More precisely, he suggests to replace the well-known Equation 2 with the following:

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

As discussed in [43], the second message block of a keyed hash function has a run of several consecutive zeros (red rectangle in Figure 2) — i.e., 287 zeros in the padding scheme and other 54 zeros in 64-bit message length — hence nine (out of sixteen) 32-bit words $W[i]$ are set to zero — $W[6] = W[7] = \cdots = W[14] = 0$. Therefore, 27 out of 192 XOR operations in Equation 2 are involved in zero based operations and can be easily omitted.

Making the expansion of Equation 4, it is easy to observe that words $W[6], \ldots, W[14]$ are presented in a greater number of times than in Equation 2. By simply replacing the original SHA-1 message scheduling function with Equation 4, we increase the number of useless XOR operations from 27 to 45.

4.2 Extending Intel optimizations

To improve the performance of the message scheduling $W[i]$, we extend the idea suggested by Max Locktyukhin [26] as follows:
Firstly, define $W[i]$ as a generic function

$$W[i] = ROTL^E(W[i-A] \oplus W[i-B] \oplus W[i-C] \oplus W[i-D]) \tag{5}$$

Secondly, set $i = i - A$ and compute $W[i - A]$:

$$\begin{aligned} W[i-A] &= ROTL^E(W[(i-A)-A] \oplus W[(i-A)-B] \oplus W[(i-A)-C] \oplus W[(i-A)-D]) \\ &= ROTL^E(W[i-2A] \oplus W[i-A-B] \oplus W[i-A-C] \oplus W[i-A-D]) \end{aligned} \tag{6}$$

Then repeat the same to $i = i - B$, $i = i - C$, and $i = i - D$ getting:

$$W[i-B] = ROTL^E(W[(i-B)-A] \oplus W[i-2B] \oplus W[(i-B)-C] \oplus W[(i-B)-D]) \tag{7}$$

$$W[i-C] = ROTL^E(W[(i-C)-A] \oplus W[(i-C)-B] \oplus W[i-2C] \oplus W[(i-C)-D]) \tag{8}$$

$$W[i-D] = ROTL^E(W[(i-D)-A] \oplus W[(i-D)-B] \oplus W[(i-D)-C] \oplus W[i-2D]) \tag{9}$$

Thirdly, substitute Equations 6, 7, 8, 9, in 5 obtaining:

$$\begin{aligned} W[i] = ROTL^E(&ROTL^E(W[i-2A] \oplus W[i-A-B] \oplus W[i-A-C] \oplus W[i-A-D]) \\ &\oplus ROTL^E(W[i-B-A] \oplus W[i-2B] \oplus W[i-B-C] \oplus W[i-B-D]) \\ &\oplus ROTL^E(W[i-C-A] \oplus W[i-C-B] \oplus W[i-2C] \oplus W[i-C-D]) \\ &\oplus ROTL^E(W[i-D-A] \oplus W[i-D-B] \oplus W[i-D-C] \oplus W[i-2D])) \end{aligned} \tag{10}$$

Recalling that

$$ROTL^X(Y \oplus Z) = ROTL^X(Y) \oplus ROTL^X(Z) \tag{11}$$

and

$$ROTL^X(ROTL^X(Y)) = ROTL^{2X}(Y) \tag{12}$$

apply Equation 11 to 10, then remove the values XORed twice and finally, apply Equation 12 getting:

$$W[i] = ROTL^{2E}(W[i-2A]) \oplus ROTL^{2E}(W[i-2B]) \oplus ROTL^{2E}(W[i-2C]) \oplus ROTL^{2E}(W[i-2D]) \tag{13}$$

Equation 13 can be applied to 32-bit words $W[i]$ with $i > 63$ of the Equation 4, generating a new SHA-1 message scheduling function:

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

Again, in Equation 14 words $W[6], \dots, W[14]$ are presented a greater number of times than Equations 2 and 4, thus using the new message scheduling function, the number of useless XOR operations is further increased from 45 to 61 (out of 192).

4.3 Exploiting Boyar-Peralta heuristic

In Sections 4.1 and 4.2 we described how to speed up the computation of the SHA-1 message scheduling function. The approach adopted iteratively computes 32-bit words $W[i]$, $16 \leq i \leq 79$, as a function of the previous thirty-two words (see Equation 4) or sixty-four words (see Equation 14), but none of them compute such $W[i]$ by using only words $W[0], \ldots, W[15]$. In this section, we (a) provide an alternative method to compute the SHA-1 message scheduling function that may be adopted in systems with strong memory constraints and then, exploiting this alternative method, (b) investigate the possibility to further reduce the number of XORs used in Equation 14. More precisely, we:

1. define $W[16]$,...,$W[79]$ as a function of $W[0], \ldots, W[15]$;
2. write 32-bit words $W[i]$ as a linear system of sixty four equations over $GF(2)$;
3. reduce the size of such a linear system;
4. apply heuristics for minimizing the total number of XORs of a given function;
5. identify a subset of 32-bit words $W[i]$ that are computed with a small number of XORs.

Let us explain such five steps in details.

Firstly, unfold Equation 14, defining it as a function of $W[0], \ldots, W[15]$

$$
\begin{cases}
W[16] = W[0]^1 \oplus W[2]^1 \oplus W[8]^1 \oplus W[13]^1 \\
W[17] = W[1]^1 \oplus W[3]^1 \oplus W[9]^1 \oplus W[14]^1 \\
W[18] = W[2]^1 \oplus W[4]^1 \oplus W[10]^1 \oplus W[15]^1 \\
W[19] = W[0]^2 \oplus W[2]^2 \oplus W[3]^1 \oplus W[5]^1 \oplus W[8]^2 \oplus W[11]^1 \oplus W[13]^2 \\
W[20] = W[1]^2 \oplus W[3]^2 \oplus W[4]^1 \oplus W[6]^1 \oplus W[9]^2 \oplus W[12]^1 \oplus W[14]^2 \\
W[21] = W[2]^2 \oplus W[4]^2 \oplus W[5]^1 \oplus W[7]^1 \oplus W[10]^2 \oplus W[13]^1 \oplus W[15]^2 \\
W[22] = W[0]^3 \oplus W[2]^3 \oplus W[3]^2 \oplus W[5]^2 \oplus \cdots \oplus W[8]^3 \oplus W[11]^2 \oplus W[13]^3 \oplus W[14]^1 \\
W[23] = W[1]^3 \oplus W[3]^3 \oplus W[4]^2 \oplus W[6]^2 \oplus \cdots \oplus W[9]^3 \oplus W[12]^2 \oplus W[14]^3 \oplus W[15]^1 \\
\ldots \\
W[79] = W[0]^8 \oplus W[0]^{22} \oplus W[1]^7 \oplus \cdots \oplus W[15]^{13} \oplus W[15]^{14} \oplus W[15]^{17} \oplus W[15]^{18}
\end{cases}
\tag{15}
$$

where $ROTL^X(W[i])$ is denoted, for short, as $W[i]^X$. Then, we delete 32-bit words $W[i]$ equal to zero — i.e, $W[6]^X, \ldots, W[14]^X$ — getting:

$$\begin{cases} W[16] = W[0]^1 \oplus W[2]^1 \\ W[17] = W[1]^1 \oplus W[3]^1 \\ W[18] = W[2]^1 \oplus W[4]^1 \oplus W[15]^1 \\ W[19] = W[0]^2 \oplus W[2]^2 \oplus W[3]^1 \oplus W[5]^1 \\ W[20] = W[1]^2 \oplus W[3]^2 \oplus W[4]^1 \\ W[21] = W[2]^2 \oplus W[4]^2 \oplus W[5]^1 \oplus W[15]^2 \\ W[22] = W[0]^3 \oplus W[2]^3 \oplus W[3]^2 \oplus W[5]^2 \\ W[23] = W[1]^3 \oplus W[3]^3 \oplus W[4]^2 \oplus W[15]^1 \\ W[24] = W[0]^2 \oplus W[2]^2 \oplus W[2]^3 \oplus W[4]^3 \oplus W[5]^2 \oplus W[15]^3 \\ W[25] = W[0]^4 \oplus W[1]^2 \oplus W[2]^4 \oplus W[3]^2 \oplus W[3]^3 \oplus W[5]^3 \\ \ldots \\ W[79] = W[0]^8 \oplus W[0]^{22} \oplus W[1]^7 \oplus W[1]^8 \oplus \cdots \oplus W[15]^{18} \end{cases} \tag{16}$$

Secondly, we write the 64 rows of Equation 16 as a $m \times n$ matrix M over $GF(2)$, where m is the number of rows — i.e., 64, one for each equation $W[i]$ — and n is the number of columns — i.e., $224 = 7 \times 32$, seven 32-bit words $W[0]$, $\ldots$, $W[5]$, $W[15]$ which can be left-rotated 32 times, $W[i]$, $ROTL^1(W[i])$, $\ldots$, $ROTL^{31}(W[i])$. For the sake of simplicity, we denote $W[0]$ as $j = 0$, $ROTL^1(W[0])$ as $j = 1, \ldots, W[1]$ as $j = 32$, $ROTL^1(W[1])$ as $j = 33$, and so on. For each row i of the matrix M, we place 1 in position i, j if a 32-bit word j appears in equation $W[i]$. Therefore, we represent the linear system 16 with a 64×224 matrix M with coefficients over $GF(2)$, and each row of such a matrix is interpreted as the XOR of words (columns) containing 1. Then, we reduce the size of M deleting the columns which do not provide any contribution — i.e., columns filled with zeros. We get a sparse 64×149 matrix M.

Thirdly, in Equation 16, we substitute all occurrences of $ROTL^X(W[5])$ and $ROTL^X(W[15])$, $0 \leq X \leq 31$, with constant values — notice that $W[5]$ and $W[15]$, are set to well-known values, respectively padding and message length,

and can be computed in advance. We call such values $const[i]$ and substitute them in each row i of the linear system 16, obtaining:

$$\begin{cases} W[16] = W[0]^1 \oplus W[2]^1 \\ W[17] = W[1]^1 \oplus W[3]^1 \\ W[18] = W[2]^1 \oplus W[4]^1 \oplus const[18] \\ W[19] = W[0]^2 \oplus W[2]^2 \oplus W[3]^1 \oplus const[19] \\ W[20] = W[1]^2 \oplus W[3]^2 \oplus W[4]^1 \\ W[21] = W[2]^2 \oplus W[4]^2 \oplus const[21] \\ W[22] = W[0]^3 \oplus W[2]^3 \oplus W[3]^2 \oplus const[22] \\ W[23] = W[1]^3 \oplus W[3]^3 \oplus W[4]^2 \oplus const[23] \\ W[24] = W[0]^2 \oplus W[2]^2 \oplus W[2]^3 \oplus W[4]^3 \oplus const[24] \\ W[25] = W[0]^4 \oplus W[1]^2 \oplus W[2]^4 \oplus \cdots \oplus const[25] \\ \cdots \\ W[79] = W[0]^8 \oplus W[0]^{22} \oplus W[1]^7 \oplus \cdots \oplus const[79] \end{cases} \quad (17)$$

where $const[18] = ROTL^1(W[15])$, $const[19] = const[18]$, $const[21] = ROTL^1(W[5]) \oplus ROTL^2(W[15])$, and so on. Again, we write the 64 rows of Equation 17 as a $m \times n$ matrix M' over $GF(2)$ and delete the columns which do not provide any contribution, thus getting a new sparse 64×107 matrix M'.

Fourthly, finding the minimum number of XORs necessary to compute Equation 14 corresponds to the problem of finding a gate-optimal Boolean circuit that computes the linear systems 16 or 17. This problem is known to be a hard problem [10] [11], thus sub-optimal solutions can be computed applying polynomial- or exponential-time heuristics — e.g., [33],[12], and [44]. The Paar heuristic [33] usually performs well on large matrices but does not consider cancellation[1] over GF(2). The Visconti-Schiavo-Peralta heuristic [44] considers cancellation over GF(2) but performs well on dense matrices. The Boyar-Peralta (BP) heuristic [12] considers cancellation over GF(2) and works well with sparse matrices. Since both M and M' are sparse, we can apply the BP heuristic to find a small circuit that implements Equations 16 and 17. Unfortunately, we are not able to solve a large linear system due to the exponential time complexity of BP, thus applying BP to several sub-matrices of M and M' with size smaller than 64×39 (that is the maximum allowed by our hardware). We repeat this point many times, choosing different sub-matrices and obtaining several circuits that implement the linear systems 16 and 17. Of course, at a cost of tripling the running time, one can also run [33] [44] heuristics and then pick the best circuit. We did this, but with the same or worse results.

Fifthly, since $W[16],\ldots,W[79]$ are computed in Equation 14 using at most three XOR operations, we try to identify a subset of 32-bit words $W[i]$ computed with less than three XORs. Exploiting the Boolean circuits provided by

[1] The fact that for all x in GF(2), we have $x \oplus x = 0$.

BP heuristic, we note that $W[29]$, $W[30]$, $W[31]$, $W[60]$, and $W[62]$ defined in Equation 14 as

- $W[29] = ROTL^1(W[26] \oplus W[21] \oplus W[15])$
- $W[30] = ROTL^1(W[27] \oplus W[22] \oplus W[16])$
- $W[31] = ROTL^1(W[28] \oplus W[23] \oplus W[17] \oplus W[15])$
- $W[60] = ROTL^2(W[54] \oplus W[44] \oplus W[32] \oplus W[28])$
- $W[62] = ROTL^2(W[56] \oplus W[46] \oplus W[34] \oplus W[30])$

can be replaced with the following, and less expensive, equations:

- $W[29] = ROTL^2(W[23]) \oplus k[29]$
- $W[30] = ROTL^2(W[24] \oplus k[16])$
- $W[31] = ROTL^2(W[25] \oplus k[17]) \oplus k[31]$
- $W[60] = ROTL^4(W[48] \oplus W[28] \oplus W[0])$
- $W[62] = ROTL^4(W[50] \oplus W[30] \oplus W[0])$

where $k[29]$ is set to constant value $ROTL^2(W(5)) \oplus ROTL^1(W(15))$, $k[16]$ is equal to $W(0) \oplus W(2)$ — that is a value computed in $W[16]$, $W[16] = ROTL^1(W(0) \oplus W(2)) = ROTL^1(k[16])$ — $k[17]$ is equal to $W(1) \oplus W(3)$ (previously computed in $W[17]$), and finally $k[31]$ is set to constant value $ROTL^1(W(15)) \oplus ROTL^2(W(15))$.

The final SHA-1 message scheduling function will be computed using Equation 14 in which $W[29]$, $W[30]$, $W[31]$, $W[60]$, and $W[62]$ are replaced with those presented in this Section. Therefore, we further increase the total number of useless XORs from 61 to 66 (out of 192).

Notice that although Equation 15 increases the total number of XOR operations, it requires to store (in the worst case) sixteen 32-bit words. This value can be largely reduced in specific cases. For example, when we compute HMAC-SHA-1, it requires to store only five words — i.e., $W[0], \ldots, W[4]$ (see Equation 17). Therefore, this approach can be considered a valid alternative method for computing the SHA-1 message scheduling function in systems with tight memory constraints.

5 Discussion and concluding remarks

In this paper, we show how performance improvements of SHA-1 and HMAC-SHA-1 might be exploited to speed up a PBKDF2-HMAC-SHA-1 implementation. In particular, we investigated the possibility to avoid part of the CPU-intensive operations involved in the PBKDF2 computation by executing a set of equivalent, but less onerous, instructions. Since the computation of first message block of HMAC-SHA-1 does not provide any contribution in slowing attackers down, we focused on the second message block, the only one able to slow them down. Exploiting and extending Intel optimizations and the Boyar-Peralta heuristic, we provide a new SHA-1 message scheduling function and showed that 66 (out of 192) XOR operations can be easily omitted. Since PBKDF2 is used in

real-world applications to benchmark the userâĂŹs system and provide an appropriate level of security (e.g. LUKS), users can benefit from such performance improvements. Therefore, it is important that these findings are implemented in crypto libraries.

Moreover, there is a second way to gauge how interesting these results are. The SHA-1 message scheduling function presented in Equation 14 (with or without the improvements of $W[29]$, $W[30]$, $W[31]$, $W[60]$, and $W[62]$ described in Section 4.3) is efficient from the standpoint of execution time, but it is not from the memory usage. In the worst case scenario it requires to store eighty 32-bit words. Adopting the idea of the circular queue [32] mentioned in Section 2, Equations 14 requires to store sixty-four words. The latter are far from the sixteen used in Equation 3, hence they cannot be used if the memory is limited. For this reason, we suggest to adopt the unfolded SHA-1 message scheduling function described in Equation 17 for computing HMAC-SHA-1 under tight memory constraints — e.g. embedded systems. This approach requires to store only five words — i.e., $W[0], \ldots, W[4]$ — compared with sixteen (at least) of the other equivalent functions described in the literature. The only drawback of this second approach is the lengthening of the execution time of a single HMAC-SHA-1 due to the increased number of XOR operations in Equation 17.

It might seem counterintuitive, but also attackers might exploit this approach to speed up a GPU-based brute force attack against PBKDF2. Indeed, the lengthening of the execution time of a single HMAC-SHA-1 and the overall performance of an attack are not actually at odds. Since each GPU thread has only a small amount of fast memory available and, cache misses drastically affect the performance due to data swapping and loading, the efficiency of a GPU-based attack strongly depends on the compact design of the HMAC-SHA-1 processing function. Therefore, attackers might adopt Equation 17 to generate a compact HMAC-SHA-1 function, that means saving fast GPU memory. In this way, the drawbacks of the lengthening of the execution time of a single HMAC-SHA-1 will be offset by the reduced number of cache misses which negatively affect the performance and the overall execution time of a brute force attack might be improved.

References

1. Password hashing competition (2015), `https://password-hashing.net/`, [Online, accessed 10-October-2016]
2. AgileBits: How PBKDF2 strengthens your Master Password ([Online, accessed 30-January-2017]), https://support.1password.com/pbkdf2/
3. Apple Inc.: Best Practices for Deploying FileVault 2 ([Online, accessed 21-October-2016]), `http://training.apple.com/pdf/WP_FileVault2.pdf`
4. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Proceedings of Advances in Cryptology—CRYPTOâĂŹ96. pp. 1–15. Springer (1996)
5. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions—the HMAC construction. RSA Laboratories CryptoBytes 2(1), 12–15 (1996)

6. Biryukov, A., Dinu, D., , Khovratovich, D.: Argon2 (version 1.2). University of Luxembourg, Luxembourg (July 2015), https://password-hashing.net/submissions/specs/Argon-v3.pdf
7. Bonneau, J.: Guessing human-chosen secrets. Tech. rep., University of Cambridge, Computer Laboratory (2012)
8. Bosselaers, A., Govaerts, R., Vandewalle, J.: Sha: a design for parallel architectures? In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 348–362. Springer (1997)
9. Bossi, S., Visconti, A.: What users should know about full disk encryption based on luks. In: International Conference on Cryptology and Network Security. pp. 225–237. Springer (2015)
10. Boyar, J., Matthews, P., Peralta, R.: On the shortest linear straight-line program for computing linear forms. Lecture Notes in Computer Science 5162, 168–179 (2008)
11. Boyar, J., Matthews, P., Peralta, R.: Logic minimization techniques with applications to cryptology. Journal of Cryptology pp. 1–33 (2012), http://dx.doi.org/10.1007/s00145-012-9124-7
12. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Festa, P. (ed.) Experimental Algorithms, Lecture Notes in Computer Science, vol. 6049, pp. 178–189. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13193-6_16
13. Brož, M.: Cryptsetup 1.7.5 Release Notes (2017), https://www.kernel.org/pub/linux/utils/cryptsetup/v1.7/v1.7.5-ReleaseNotes
14. Burr, W.E., Dodson, D.F., Newton, E.M., Perlner, R.A., Polk, W.T., Gupta, S., Nabbus, E.A.: SP 800-63-2. Electronic Authentication Guideline (August 2013), http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf
15. Choudary, O., Grobert, F., Metz, J.: Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. Cryptology ePrint Archive, Report 2012/374 (2012), https://eprint.iacr.org/2012/374.pdf
16. Codebook: Your secrets are safe with Codebook ([Online, accessed 11-July-2017]), https://www.zetetic.net/codebook/
17. Eastlake 3rd, D., Jones, P.: Us secure hash algorithm 1 (sha1). Tech. rep. (2001)
18. Forler, C., Lucks, S., Wenzel, J.: Catena : A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525 (2013)
19. Fruhwirth, C.: New methods in Hard Disk Encryption (2005), http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf
20. Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.2.2 (2016), https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf
21. Gaudet, D.: Cryptographic software ([Online, accessed 27-November-2016]), http://arctic.org/~dean/crypto/
22. Goldberg, Jeffrey: Bcrypt is great, but is password cracking âĂIJinfeasibleâĂİ? ([Online, accessed 11-July-2017]), https://blog.agilebits.com/2015/03/30/bcrypt-is-great-but-is-password-cracking-infeasible/
23. Goldberg, Jeffrey: Defending against crackers: Peanut Butter Keeps Dogs Friendly, Too. ([Online, accessed 11-July-2017]), https://blog.agilebits.com/2011/05/05/defending-against-crackers-peanut-butter-keeps-dogs-friendly-too/
24. GRUB: GNU GRUB Manual 2.00 ([Online, accessed 4-Spetember-2016]), http://www.gnu.org/software/grub/manual/grub.html

25. IEEE 802.11 WG: Part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Std 802.11 i-2004 (2004)
26. Intel: Improving the Performance of the Secure Hash Algorithm (SHA-1) ([Online, accessed 28-December-2016]), https://software.intel.com/en-us/articles/improving-the-performance-of-the-secure-hash-algorithm-1/
27. Kaliski, B.: PKCS#5: Password-Based Cryptography Specification Version 2.0. RFC 2898, RFC Editor (September 2000), https://www.rfc-editor.org/rfc/rfc2898.txt
28. Keeper: Keeper's Best-In-Class Security ([Online, accessed 30-January-2017]), https://keepersecurity.com/security.html
29. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor (February 1997), https://www.rfc-editor.org/rfc/rfc2104.txt
30. LassPass: Password Iterations (PBKDF2) ([Online, accessed 30-January-2017]), https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2/
31. NIST: FIPS PUB 198-1. The Keyed-Hash Message Authentication Code (HMAC) (July 2008), http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf
32. NIST: FIPS PUB 180-4. Secure Hash Standard (SHS) (2012), http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
33. Paar, C.: Optimized arithmetic for reed-solomon encoders. In: IEEE International Symposium on Information Theory (1997)
34. Peslyak, A.: yescrypt – password hashing scalable beyond bcrypt and scrypt. Openwall, Inc. (May 2014), http://www.openwall.com/presentations/PHDays2014-Yescrypt/, [Online, accessed 10-October-2016]
35. Pornin, T.: The MAKWA Password Hashing Function (April 2015), http://www.bolet.org/makwa/makwa-spec-20150422.pdf, [Online, accessed 10-October-2016]
36. RARLAB: RAR 5.0 archive format ([Online, accessed 11-July-2017]), http://www.rarlab.com/rarnew.htm
37. RSA Laboratories: PKCS #5 V2.1: Password Based Cryptography Standard (2012)
38. Ruddick, A., Yan, J.: Acceleration attacks on pbkdf2: or, what is inside the black-box of oclhashcat? In: 10th USENIX Workshop on Offensive Technologies (2016)
39. Shannon, C.E.: Prediction and entropy of printed english. Bell system technical journal 30(1), 50–64 (1951)
40. 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)
41. Steube, J.: Optimising Computation of Hash-Algorithms as an Attacker ([Online, accessed 4-October-2016]), https://hashcat.net/events/p13/js-ocohaaaa.pdf
42. Turan, M.S., Barker, E.B., Burr, W.E., Chen, L.: SP 800-132. Recommendation for Password-Based Key Derivation. Part 1: Storage Applications (December 2010), http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
43. 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)
44. Visconti, A., Schiavo, C., Peralta, R.: Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF(2). Cryptology ePrint Archive, Report 2017/194 (2017), https://eprint.iacr.org/2017/194.pdf