Examining PBKDF2 security margin — case study of LUKS *

Andrea Visconti[1], Ondrej Mosnáček[2], Milan Brož[2], and Vashek Matyáš[2]

[1] Department of Computer Science, Università degli Studi di Milano
andrea.visconti@unimi.it,
WWW home page: http://www.di.unimi.it/visconti
[2] Faculty of Informatics, Masaryk University,
xmosnac@fi.muni.cz, xbroz@fi.muni.cz, matyas@fi.muni.cz

Abstract. Passwords are widely used to protect our sensitive information or to gain access to specific resources. They should be changed frequently and be strong enough to prevent well-known attacks. Unfortunately, user-chosen passwords are usually short and lack sufficient entropy. A possible solution to these problems is to adopt a Key Derivation Function (KDF) that allows legitimate users to spend a moderate amount of time on key derivation, while imposing CPU/memory-intensive operations on the attacker side. In this paper, we focus on long-term passwords secured by the Password-Based Key Derivation Function 2 (PBKDF2) and present the case study of Linux Unified Key Setup (LUKS), a disk-encryption specification commonly implemented in Linux based operating systems. In particular, we describe how LUKS protects long-term keys by means of iteration counts defined at runtime, and analyze how external factors may affect the iteration counts computation. In doing so, we provide means of evaluating the iteration count values defined at run-time and experimentally show to what level PBKDF2 is still capable of providing sufficient security margin for a LUKS implementation.

Keywords: Password-Based Key Derivation Function 2 (PBKDF2), Iteration counts, GPU, Linux Unified Key Setup (LUKS)

1 Introduction

User-chosen passwords are widely used to protect personal data or to gain access to specific resources. Therefore, passwords should be strong enough to prevent

---

[*] This paper extends and improves the previous work of the first author "What users should know about Full Disk Encryption based on LUKS" presented at the 14th International Conference on Cryptology and Network Security (CANS 2015). New experiments were conducted for this article (on 32-bit and 64-bit architectures), using different library versions and different cryptographic backends. In addition, a costing model is defined and used to estimate the cost of attacks on LUKS partition passwords created using Cryptsetup.

well-known attacks such as dictionary and brute-force attacks. User-chosen passwords are usually short and lack enough entropy [43], [23] and cannot be directly used as a key for secure cryptographic systems. A solution to this problem is described in [42]. By applying a Key Derivation Function (KDF) to a user-chosen password, we allow legitimate users to spend a moderate amount of time on key derivation, while we impose CPU-intensive operations on the attacker side.

The developments of the Graphic Processing Unit (GPU) architecture have changed this perspective. As opposed to CPUs, the GPU architecture consists of a large number of small processors (so-called *streaming multiprocessors*), which are capable of running a group of threads that execute the same program (and always the same instruction) over different data [41]. In principle, this architecture is convenient for password cracking, because it involves computing the same function (a password-hashing function or a PBKDF) over a large number of passwords. However, since each GPU thread has only a small amount of fast memory available, the potential efficiency of the GPU implementation strongly depends on the design of the password processing function.

The first standardized PBKDF is *PBKDF2* [30, 45], a CPU-intensive key strengthening algorithms. The design of PBKDF2 includes a time-based security parameter (the iteration count), but it has no parameter to increase the necessary memory usage of the function. PBKDF2 is defined generically in the sense that it does not strictly specify the *pseudorandom function* (PRF) to be used as its core, but in practice HMAC [31] is almost solely used (usually with SHA-1, often also with SHA-256 or SHA-512). This means that PBKDF2 can be often implemented very efficiently on GPUs, thereby providing an attacker a huge potential speedup compared to the defender (who almost always runs PBKDF2 on a CPU).

As a reaction to the weakness of PBKDF2 against GPU-based attacks (and some other flaws), several memory-intensive key strengthening algorithms have been developed.

Bcrypt, for example, was introduced in 1999 [40] and is used for password hashing in the OpenBSD operating system and PHP [15]. It employs some protections against GPU/ASIC/FPGA attacks, but the memory usage is still fixed, so it is not completely safe for the future [38].

Scrypt was introduced in 2009 [35] and allows tuning both time-based and memory-based security parameters. However, it has been criticised for allowing a so-called *time-memory tradeoff*, which makes it possible to achieve constant memory usage by performing more computation [36]. It has also been recently (August 2016) standardized as RFC 7914 [34].

In 2013, an open competition called *Password Hashing Competition* was announced. The aim of the competition was to “identify new password hashing schemes in order to improve on the state-of-the-art” [9]. In July 2015, the competition selected *Argon2* as the winning algorithm [20] and gives special recognition to Catena [25], Lyra2 [44], yescrypt [37], and Makwa [39].

1.1 Motivations and contributions

Although Argon2 is expected to become a new de-facto standard for password-based key derivation and password hashing, PBKDF2 is still widely implemented to derive keys in many security-related systems such as WPA/WPA2 encryption process [29], Linux Unified Key Setup (LUKS) [21, 27], VeraCrypt [16], EncFS [2], FileVault Mac OS X [17, 24], GRUB2 [3], Winrar [4], and many others.

In PKCS#5 [42] a number of recommendations for the implementation of PBKDF2 have been described. More precisely, PBKDF2 uses salt and iteration count to slow down the attackers as much as possible. The salt, randomly selected, is used to generate a large set of keys corresponding to a given password, while the iteration count specifies the number of times the underlying pseudorandom function is called to generate a block of keying material. Both salt and iteration count do not need to be kept secret. NIST suggests to select the iteration count as large as possible, as long as the time required to generate the key is acceptable for the user [45]. More precisely, SP 800-132 specifies that 10,000,000 may be an appropriate iteration count value for very critical keys on very powerful system, and 1,000 is a minimum recommended value.

In real-world applications, mainly two different approaches have been adopted in providing the iteration count:

1. define a value a priori — e.g., WPA/WPA2 encryption process [29], VeraCrypt [10];
2. define a value at runtime — e.g., LUKS disk encryption specification [21,27].

The first approach is widely used, yet it does not take into account hardware specifications of the devices on which PBKDF2 should run. This means that old devices and more powerful ones use the same iteration count value that is (usually) provided adopting a conservative approach — i.e., it favors performance at the expense of security. On the contrary, when applications care about security, the iteration count value is increased considerably — e.g., VeraCrypt [10] — impacting system performance and usability.

The second approach tries to resolve the cons previously described. It executes a runtime test on a device gathering information about the environment and, on the bases of hardware and software characteristics collected, provides an appropriate iteration count value — the details of the runtime test can be found in Section 4.1 or in LUKS disk encryption specification [26]. Unfortunately, this approach does not solve all problems. Indeed, the runtime testing may be negatively affected by several external factors such as the performance of different cryptographic backends, e.g., OpenSSL, Libgcrypt, the version of such cryptographic libraries, the particular architecture on which the code is running on, e.g., 32-bit/64-bit architecture, and so on.

In addition, we do not forget that (a) legitimate users and attackers run their algorithms on different hardware — i.e., regular users run their code on CPUs, while attackers may also run it on specialized hardware (ASIC/FPGA) or GPUs — and (b) some applications require long-term keys and therefore attackers can easily run their code off-line for a long time.

In such a scenario, we focus on the iteration count defined at runtime, presenting the case study of the Linux Unified Key Setup (LUKS), a disk-encryption specification commonly implemented in Linux based operating systems. In particular, we describe how LUKS protects long-term keys by means of an iteration count defined at runtime, and analyze how external factors may affect the iteration counts computation. In doing so, we provide means of evaluating the iteration count values defined at run-time and experimentally show to what level PBKDF2 is still capable of providing sufficient security margin for a LUKS implementation.

1.2 Organization of the paper

The remainder of the paper is organized as follows. In Section 2, we briefly introduce the Password-Based Key Derivation Function version 2. In Section 3, we introduce a method of estimating costs and duration of cracking a password protected by PBKDF2. In Section 4, we describe the Linux Unified Key Setup, a disk encryption specification based on PBKDF2, which computes the iteration count values by executing a runtime test. We experimentally show how external factors may affect the iteration count computation. In Section 5, we provide means of evaluating the iteration count values defined at run-time and experimentally show to what level PBKDF2 is still capable of providing sufficient security margin for a LUKS implementation. Finally, discussion and conclusions are drawn in Section 6.

2 Password-Based Key Derivation Function 2

PBKDF2 is a Password-Based Key Derivation Function described in PKCS #5 [42], [45]. 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 [46, 47], 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 [33], cipher, or HMAC [18], [19], and [31]. In this paper, we will refer to HMAC-SHA-1, HMAC-SHA-256, HMAC-SHA-512, and HMAC-RIPEMD-160.

3 Estimating the cost of attacking PBKDF2

Usually, the computational cost of cracking a password protected by PBKDF2 is estimated on the number of passwords per second that can be processed. In this section, we introduce a different approach of estimating such a cost. We will focus not only on the time required to recover a password, but also on the amount of computing resources needed to do so. In particular, the method suggested is based on two main sources of the cost: the cost of the power consumed by the hardware doing the cracking and the cost of the hardware itself. The cost is influenced by several factors – the PBKDF2 iteration count, the PRF used for PBKDF2, the size of the derived key/hash, the strength of the password and the performance characteristics of the hardware used for cracking. Since there might be many other contributing factors adding to the cost of an actual attack, our proposed estimations can only be seen as a lower bound for the actual cost.

3.1 Attack scenario

The method we introduce assumes either brute-force or dictionary offline attack – i.e., that the attacker has access to the PBKDF2 hash of the searched password (along with the associated salt and iteration count), or some other information that allows her[3] to verify that the output of PBKDF2 matches the expected one (e.g., she knows a plaintext-ciphertext pair encrypted with the output used as the key). The method further assumes that the time and computation power needed for verifying the PBKDF2 output of a given password candidate is negligible.

---

[3] The sex of the attacker was set by a random coin toss.

3.2 Method of calculation

The estimation uses the following input variables:

- I [iter $\cdot$ pw^{-1}] – the number of PBKDF2 iterations,
- $B = \left\lceil \frac{\text{derived key size}}{\text{PRF output size}} \right\rceil$ – the number of PBKDF2 output blocks,
- S [pw] – the password search space,
- T [s] – the maximum acceptable (average) duration of the attack,
- E [\$ $\cdot$ kWh^{-1}] – the price of electricity available to the attacker[4],
- V [iter $\cdot$ s^{-1} $\cdot$ dev^{-1}] – the attacker's device's PBKDF2 computation speed (in terms of PBKDF2 iterations per second),
- P [W] – the attacker's device's power draw while in full operation,
- D [\$] – the purchase price of a single device,
- L [s] – the average lifetime of a single device.

The estimated number of devices needed for the attack and its cost are calculated using the following formulas:

$$\text{number of devices needed [dev]} = \frac{IBS}{2VT}, \text{ and} \tag{4}$$

$$\text{average attack cost [\$]} = \frac{IBS}{2V}\left(PE + \frac{D}{L}\right). \tag{5}$$

3.3 Parameters for calculation

Password search space: This parameter (S) should express the assumed size of the password search space. This number depends on the way the password was selected and may vary for different scenarios. For example if the password is a random string of 8 characters, e.g., A-Z, a-z, and 0-9, the search space is $(26 + 26 + 10)^8 \approx 2^{48}$.

Maximum acceptable attack duration: When cracking a password, it is usually required that the attack succeeds within a reasonable amount of time (e.g., 1 month/1 year/5 years/...). Since a brute-force or dictionary attack can be trivially parallelized, it is sufficient to increase the number of devices that perform the cracking. Such an optimization does not (in theory) increase the total cost of computation, but there is certain practical limit on how many devices the attacker can use in parallel.

The value of this parameter (T) should reflect the upper bound on the attack duration and allows us to calculate the number of devices that would be needed to achieve such duration.

Price of electricity: Since we assume that the attack cost is mainly defined by the cost of consumed electricity, it is necessary to specify minimum expected electricity price for the potential attacker. The price of electricity varies a lot depending on the type of source and geographic location [1]. We suggest a conservative value of $E = \$0.05\ \text{kWh}^{-1}$ to be used for the calculations.

---

[4] \$ = US dollar.

Device-specific constants: When estimating the general cost of an attack one needs to consider what kind of hardware the attacker will use for the attack. In general, it is best to assume the worst case – that the attacker will use the most cost-efficient solution available. In the case of PBKDF2, this is especially important, since computing PBKDF2 is significantly faster and cheaper on highly parallel hardware (such as GPUs or FPGAs) than on a regular CPU (see section 1). Therefore, it is important to base the cost estimation on the most efficient hardware, for which we have the PBKDF2 efficiency characteristics available.

PBKDF2 computation speed: This constant (V) is the experimentally measured speed of PBKDF2 computation on the device. It is expressed in terms of iterations per second per PRF output block.

Device power draw: This constant (P) represents the power draw of the device during the computation. The value can be measured or obtained from the device's data sheet.

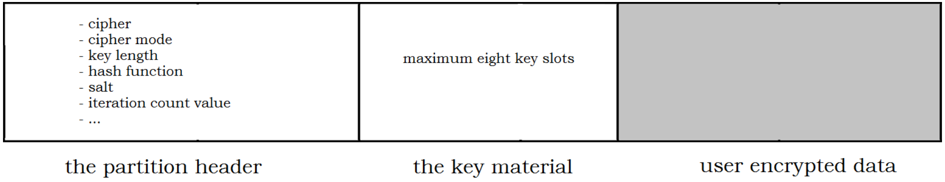
4 How to protect long-term keys: a case study of LUKS

4.1 A brief introduction to LUKS

The Linux Unified Key Setup (LUKS) is a disk-encryption specification commonly implemented in Linux based operating systems. It is a platform-independent standard on-disk format developed by Clemens Fruhwirth in 2004 [26, 27]. A LUKS partition (see Figure 1) includes:

1. the partition header,
2. the key material (i.e., a number of key slots), and
3. the user encrypted data.

Fig. 1. A LUKS partition.

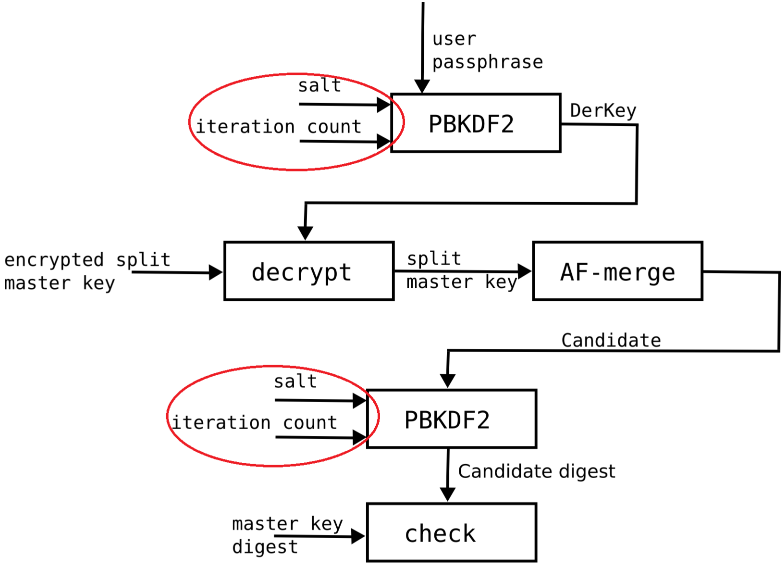


Firstly, the partition header contains information about cipher, cipher mode, key length, hash function, master key checksum, salt, iteration counts, etc. [26]. Secondly, a number of key slots (maximum eight) are used to store the encrypted master key. More precisely, LUKS is based on a two-level key hierarchy

[30]. A strong master key generated by the system is used to encrypt/decrypt the whole hard disk. This key is encrypted with a secret user key. The master key is unique, but a number of encrypted keys are stored, one for each user who has access to the device. LUKS protects the keys stored on the hard disk using (1) a key derivation function (i.e., PBKDF2), and (2) an anti-forensic splitter to solve the remanence issues in magnetic storage devices [28]. In particular, the anti-forensic splitter inflates and splits the master key, then a hash function is used as diffusion element. In order to recover the master key, we need a valid LUKS partition header and a secret user key. The user key unlocks a specific user key slot. Then, PBKDF2, the anti-forensic splitter, and a cipher are used to compute the master key as shown in Figure 2.

Finally, user encrypted data are stored on the device.

Fig. 2. LUKS: the master key recovery process.



Among the various parameters stored in a LUKS partition header, the two most important are salt and iteration counts (see Figure 2), because they are used by PBKDF2 to slow down brute force attacks. The salt is fetched from a random source [27], while the iteration counts are computed by making a run-time test when the encrypted partition is generated. More precisely, the run-time test computes the iteration count value by running and evaluating a PBKDF2 implementation over one- or two-second window size — respectively before and after release 1.7.0 — with a predefined password, a specific salt and a fixed key size. However, users can easily adjust this value as desired, defining a time in seconds to benchmark the iteration counts. In accordance with NIST specifications [45], the iteration count computed cannot be smaller than 1,000.

Our analysis will focus on iteration count values computed in this way. In particular, we try to understand how this parameter computed at runtime may be affected by external factors.

4.2 LUKS: Iteration count values computed at runtime

In [46,47] and [21] Visconti et al. showed how some weaknesses of PBKDF2 and Cryptsetup affected the runtime testing used by LUKS to compute the iteration count values. These weaknesses have been patched in Libgcrypt 1.7.0 [48] and Cryptsetup 1.7.0 [22], respectively. However, the runtime testing may be also affected by regular processes that run on Linux distributions, the execution speed of cryptographic algorithms implemented in different cryptographic backends and also the choice of a specific architecture — i.e., 32-bit or 64-bit OS. The influence of these external factors cannot be understood as a “weakness” and cannot be patched. Therefore, we tried to identify which factors negatively affect the iteration count computation.

In doing so, we experimentally collected several partition headers using two laptops equipped with different hardware configurations: (1) a machine with an Intel Core i5 5300U processor (with maximum frequency of 2.9 GHz, 2 cores and 4 threads) and 16 GB of RAM; (2) a machine with an Intel Core i7 4500U processor (with maximum frequency of 3.0 GHz, 2 cores and 4 threads) and 16 GB of RAM. In our testing activities, we used the following Linux distributions:

- Arch Nov 26, 2015,
- Debian (XFCE) 8.2.0,
- Fedora (GNOME) 23,
- Kali (GNOME) 2.0,
- Lubuntu (LXDE) 15.10,
- Mint (Cinnamon) 17.2,
- Ubuntu (Unity) 15.10.

All tests have been repeated for 32-bit and 64-bit architecture, using “Live CDs”. To implement a uniform method of data collection, a common configuration for all distributions has been adopted. In particular, we installed the following libraries[5]: Libgcrypt 1.6.5, OpenSSL 1.0.2g, Cryptsetup 1.6.8 and Cryptsetup 1.7.0.

For each distribution (seven), each architecture (two) and each library (four) listed above, we executed 100 runs for a total of $7 \times 2 \times 4 \times 100 = 5600$ iteration counts collected. Average values are reported in Tables 1-2 and 3-4. They refer to Intel Core i5 5300U and Intel Core i7 4500U processor, respectively.

---

[5] Note that Libgcrypt 1.6.5 and OpenSSL 1.0.2g were the latest stable version available at the time of testing.

Table 1. Average iteration counts involved in the key derivation process (Libgcrypt 1.6.5, Live CD, Intel Core i5 5300U processor).

		Cryptsetup version 1.6.8		Cryptsetup version 1.7.0	
		32-bit OS	64-bit OS	32-bit OS	64-bit OS
Arch	SHA-1	381,013	462,152	753,287	941,841
	SHA-256	183,376	320,395	808,518	1,284,213
	SHA-512	64,834	236,345	251,298	1,073,867
	RIPEMD-160	272,278	295,299	531,893	592,677
Debian	SHA-1	367,932	520,857	731,641	1,044,445
	SHA-256	191,432	336,983	798,859	1,342,441
	SHA-512	61,667	265,623	243,276	1,092,938
	RIPEMD-160	254,239	303,677	504,007	606,515
Fedora	SHA-1	387,923	513,939	771,184	1,026,156
	SHA-256	205,202	334,415	821,523	1,323,591
	SHA-512	63,331	268,130	253,724	1,089,109
	RIPEMD-160	274,983	302,392	552,623	605,248
Kali	SHA-1	387,236	514,248	772,334	1,032,963
	SHA-256	203,523	335,432	882,872	1,307,619
	SHA-512	56,562	273,236	241,608	1,072,372
	RIPEMD-160	252,314	301,133	535,232	595,942
Lubuntu	SHA-1	385,344	514,234	765,234	1,015,323
	SHA-256	205,652	331,223	818,561	1,293,967
	SHA-512	65,259	260,967	261,396	1,098,561
	RIPEMD-160	267,354	299,763	535,105	601,943
Mint	SHA-1	279,701	507,137	541,517	1,010,782
	SHA-256	196,339	338,481	788,208	1,350,419
	SHA-512	59,826	248,630	239,419	995,286
	RIPEMD-160	258,418	286,416	510,936	562,741
Ubuntu	SHA-1	385,432	508,033	771,355	1,015,164
	SHA-256	202,344	334,979	863,371	1,211,739
	SHA-512	65,672	270,253	275,296	1,197,264
	RIPEMD-160	260,230	300,666	531,976	601,356

Table 2. Average iteration counts involved in the key derivation process (OpenSSL 1.0.2g, Live CD, Intel Core i5 5300U processor).

		Cryptsetup version 1.6.8		Cryptsetup version 1.7.0	
		32-bit OS	64-bit OS	32-bit OS	64-bit OS
Arch	SHA-1	629,231	825,123	1,312,223	1,635,781
	SHA-256	468,335	616,545	2,007,356	2,455,634
	SHA-512	294,455	443,723	1,177,231	1,787,083
	RIPEMD-160	389,345	504,433	792,845	941,175
Debian	SHA-1	570,418	782,920	1,131,172	1,561,416
	SHA-256	364,159	481,299	1,444,241	1,891,303
	SHA-512	233,972	377,063	929,403	1,479,630
	RIPEMD-160	361,662	491,514	720,363	983,532
Fedora	SHA-1	537,051	803,023	1,064,544	1,577,923
	SHA-256	423,119	593,211	1,667,642	2,362,423
	SHA-512	261,398	439,068	1,049,324	1,753,742
	RIPEMD-160	345,237	496,023	686,424	975,962
Kali	SHA-1	577,389	775,991	1,099,834	1,550,711
	SHA-256	368,283	459,746	1,459,604	1,881,460
	SHA-512	232,229	375,621	919,793	1,479,147
	RIPEMD-160	365,082	491,319	725,479	786,642
Lubuntu	SHA-1	644,752	820,537	1,255,597	1,621,947
	SHA-256	493,033	619,805	1,923,145	2,427,355
	SHA-512	289,278	435,479	1,146,912	1,718,007
	RIPEMD-160	385,815	487,371	771,541	1,007,931
Mint	SHA-1	627,625	788,036	1,224,068	1,564,635
	SHA-256	391,481	482,962	1,532,511	1,898,248
	SHA-512	243,739	385,121	961,830	1,540,997
	RIPEMD-160	384,194	499,491	758,629	995,816
Ubuntu	SHA-1	645,242	824,821	1,293,239	1,610,281
	SHA-256	496,901	604,167	1,939,750	2,427,615
	SHA-512	286,276	439,493	1,154,148	1,740,286
	RIPEMD-160	391,360	504,617	781,673	1,008,184

Table 3. Average iteration counts involved in the key derivation process (Libgcrypt 1.6.5, Live CD, Intel Core i7 4500U processor).

		Cryptsetup version 1.6.8		Cryptsetup version 1.7.0	
		32-bit OS	64-bit OS	32-bit OS	64-bit OS
Arch	sha1	406,348	495,164	764,171	990,328
	sha256	193,938	329,896	870,472	1,051,325
	sha512	63,240	259,371	247,582	1,048,105
	ripemd	241,508	304,761	573,347	609,523
Debian	sha1	359,549	516,128	715,082	1,036,435
	sha256	192,770	336,841	766,468	1,354,496
	sha512	58,823	265,164	233,576	1,053,436
	ripemd	248,061	309,178	496,123	615,382
Fedora	sha1	337,285	518,217	677,248	1,027,080
	sha256	203,173	345,012	813,990	1,381,916
	sha512	62,135	260,958	233,150	1,057,849
	ripemd	241,965	308,062	484,487	613,908
Kali	sha1	423,340	581,718	652,121	1,152,271
	sha256	230,216	304,262	916,570	1,220,263
	sha512	62,015	251,418	248,306	1,067,746
	ripemd	215,060	421,052	646,927	836,201
Lubuntu	sha1	381,094	501,693	732,663	999,307
	sha256	211,327	321,316	829,288	1,309,996
	sha512	64,831	260,017	255,528	1,061,365
	ripemd	279,481	306,241	551,369	608,947
Mint	sha1	362,468	488,548	735,630	977,097
	sha256	203,821	329,896	805,031	1,319,585
	sha512	57,552	264,461	254,978	1,049,179
	ripemd	283,185	300,468	568,876	595,347
Ubuntu	sha1	367,815	484,847	670,156	977,097
	sha256	193,938	326,530	815,285	1,312,620
	sha512	63,744	262,294	254,978	1,057,849
	ripemd	228,462	300,468	474,561	598,128

Table 4. Average iteration counts involved in the key derivation process (OpenSSL 1.0.2g, CD, Intel Core i7 4500U processor).

		Cryptsetup version 1.6.8		Cryptsetup version 1.7.0	
		32-bit OS	64-bit OS	32-bit OS	64-bit OS
		32 bits	64 bits	32 bits	64 bits
Arch Linux	sha1	619,854	839,343	1,238,208	1,686,984
	sha256	483,018	635,235	1,921,189	2,528,394
	sha512	281,318	431,705	1,121,576	1,723,904
	ripemd	389,649	503,936	775,753	1,034,341
Debian	sha1	528,924	790,123	1,044,896	1,438,201
	sha256	342,245	467,158	1,361,701	1,836,294
	sha512	217,686	383,233	864,863	1,514,791
	ripemd	338,624	507,935	670,196	1,007,873
Fedora	sha1	493,255	825,805	1,028,111	1,627,980
	sha256	414,239	615,383	1,317,868	2,314,123
	sha512	257,544	422,441	901,937	1,659,642
	ripemd	345,478	513,540	642,408	1,009,861
Kali	sha1	656,410	825,605	1,168,949	1,610,062
	sha256	492,307	624,369	1,939,392	2,426,539
	sha512	283,185	427,652	1,163,634	1,651,611
	ripemd	395,061	507,935	785,275	1,007,873
Lubuntu	sha1	671,207	841,074	1,300,998	1,665,598
	sha256	511,963	631,346	1,992,948	2,429,309
	sha512	299,395	427,662	1,115,948	1,699,531
	ripemd	401,935	521,856	769,386	953,529
Mint	sha1	589,152	812,938	976,309	1,651,954
	sha256	360,529	600,389	1,297,898	2,302,316
	sha512	217,637	449,283	843,759	1,699,197
	ripemd	313,863	521,538	611,966	1,047,714
Ubuntu	sha1	670,156	847,681	1,312,820	1,684,208
	sha256	507,935	624,389	1,984,496	2,426,539
	sha512	296,295	438,355	1,158,369	1,729,728
	ripemd	390,243	522,438	805,031	1,028,111

To understand if our testing configuration — i.e., “Live CDs” — may have affected the runtime testing, we also installed some of the Linux distributions on the first laptop (Intel Core i5 5300U processor). Then we collected 1200 [6] iteration count values with the distributions installed (see Tables 5 and 6) and compared these values with those shown in Tables 1 and 2. Note that the difference between the configurations called “OSs installed” and “Live CDs” is negligible.

---

[6] For each ditribution (three), each architecture (one) and each library (four), we executed 100 runs for a total of $3 \times 1 \times 4 \times 100 = 1200$ iteration counts collected. Average values are reported in Tables 5 and 6

Table 5. Average iteration counts involved in the key derivation process (Libgcrypt 1.6.5, OSs installed).

		Cryptsetup 1.6.8	Cryptsetup 1.7.0
		64-bit OS	
Arch	SHA-1	480,019	959,102
	SHA-256	324,114	1,290,239
	SHA-512	266,921	1,069,921
	RIPEMD-160	294,801	585,291
Debian	SHA-1	518,882	1,032,923
	SHA-256	335,012	1,301,522
	SHA-512	272,611	1,079,185
	RIPEMD-160	303,092	603,801
Fedora	SHA-1	513,721	1,020,105
	SHA-256	337,001	1,320,007
	SHA-512	268,092	1,055,801
	RIPEMD-160	302,102	599,987

Table 6. Average iteration counts involved in the key derivation process (OpenSSL 1.0.2g, OSs installed).

		Cryptsetup 1.6.8	Cryptsetup 1.7.0
		64-bit OS	
Arch	SHA-1	820,723	1,537,322
	SHA-256	613,023	2,467,901
	SHA-512	438,491	1,765,118
	RIPEMD-160	507,002	976,992
Debian	SHA-1	768,173	1,535,254
	SHA-256	487,468	1,877,618
	SHA-512	370,287	1,459,036
	RIPEMD-160	491,227	976,604
Fedora	SHA-1	701,334	1,568,833
	SHA-256	605,046	2,340,005
	SHA-512	434,133	1,726,227
	RIPEMD-160	484,943	984,109

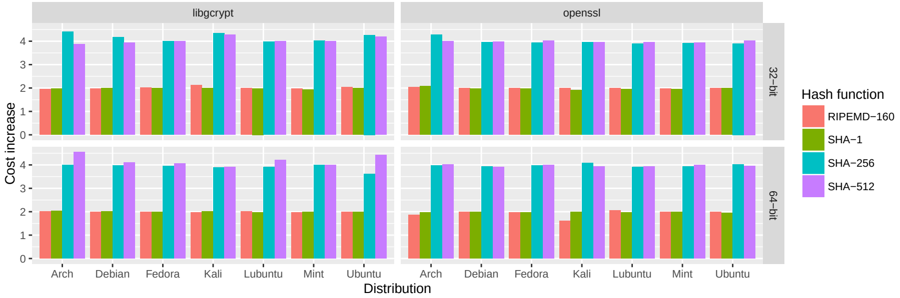
4.3 How external factors may affect the iteration count values

Tables 1 and 2 can help us to identify which factors negatively affect the iteration count computation. In particular, we look closely at library versions, architectures (32/64-bit OS), backends and distributions.

Library versions: The first external factor that we analyzed is the version of the libraries installed. Such a factor may affect the iteration count computation and consequently the attack cost. As an example, we show the differences from

Cryptsetup version 1.7.0 to 1.6.8 (see Figure 3). The value indicated in the graph has been obtained by dividing the iteration count for version 1.7.0 by the corresponding iteration count for version 1.6.8. These ratios are compared across different configurations. Note that the ratio between two costs is equivalent to the ratio between the corresponding iteration counts, as long as the hash functions and GPU devices used in Equation 5 are the same.

Fig. 3. Difference in attack cost in Cryptsetup 1.7.0 vs 1.6.8.



Disregarding some minor measurement variations, the attack cost is increased by a factor of 2 for SHA-1 and RIPEMD-160 and by a factor of 4 for SHA-256 and SHA-512. This increase is due to two independent bugs in the iteration count calculation that have been fixed in Cryptsetup version 1.7.0.

Interestingly, in Live CD distributions we found old library versions (see Table 7) such as Cryptsetup 1.6.1 (March 2013), Libgcrypt 1.6.1 (January 2014) and OpenSSL 1.0.1f (January 2014), while the latest version available at the time of writing were Cryptsetup 1.7.2 (June 2016), Libgcrypt 1.7.0 (April 2016) and OpenSSL 1.0.2h (May 2016).

Table 7. Library versions available in Live CDs (December 2015).

	Cryptsetup	Libgcrypt	OpenSSL
Arch	1.7.0	1.6.4	1.0.2d
Debian	1.6.6	1.6.3	1.0.1k
Fedora	1.6.8	1.6.4	1.0.1k
Kali	1.6.6	1.6.3	1.0.1k
Lubuntu	1.6.6	1.6.3	1.0.2d
Mint	1.6.1	1.6.1	1.0.1f
Ubuntu	1.6.6	1.6.3	1.0.2d

Unfortunately, also updated library versions available in official repositories are far from being up-to-date (see Table 8). Most of them are affected by the weaknesses described in [46],[21]. Only Arch supports libraries that are not.

Table 8. Updated library versions available in official repositories (May 2016).

	Cryptsetup	Libgcrypt	OpenSSL
Arch	1.7.1	1.7.0	1.0.2h
Debian	1.6.6	1.6.3	1.0.1k
Fedora	1.6.8	1.6.4	1.0.2h
Kali	1.6.6	1.6.3	1.0.1k
Lubuntu	1.6.6	1.6.3	1.0.2d
Mint	1.6.1	1.6.1	1.0.1f
Ubuntu	1.6.6	1.6.3	1.0.2d

Architectures (32/64-bit OS): A second external factor that may affect the iteration count computation is the architecture on which our code is running. Experimental results reported in Tables 1 and 2 show an important gap between the iteration count values computed using several configurations. As an example, we list the following:

- "Table 1, Ububtu, SHA-512, 32-bit OS, Cryptsetup 1.7.0" versus "Table 1, Ububtu, SHA-512, 64-bit OS, Cryptsetup 1.7.0". The iteration counts computed are $275,296$ (32-bit OS) and $1,197,264$ (64-bit OS). They quadrupled.
- "Table 1, Kali, SHA-512, Cryptsetup 1.6.8" ($56,562$ vs $273,236$, 32-bit and 64-bit OS respectively). Again, they quadrupled;
- "Table 1, Fedora, SHA-512, Cryptsetup 1.7.0" ($253,724$ vs $1,089,109$, 32-bit and 64-bit OS respectively). Again;
- "Table 2, Fedora, SHA-512, Cryptsetup 1.6.8" ($261,398$ vs $439,068$, 32-bit and 64-bit OS);
- "Table 2, Mint, SHA-512, Cryptsetup 1.7.0" ($961,830$ vs $1,540,997$, 32-bit and 64-bit OS);
- ...

This means that, from security perspective, it is better for the user to run the code on a 64-bit OS rather than a 32-bit OS, since we can get higher iteration counts (and thus higher costs for the attacker) for the same level of user inconvenience (i.e., time for unlocking the disk).

Backends: The third factor that we analyzed is the backend installed on the system, i.e., Libgcrypt vs. OpenSSL. Interesting results can be observed when analyzing the following configurations:

- "Table 1, Kali, SHA-512, 32-bit OS, Cryptsetup 1.6.8" versus "Table 2, Kali, SHA-512, 32-bit OS, Cryptsetup 1.6.8". The iteration counts collected are $56,562$ (Libgcrypt) and $232,229$ (OpenSSL). They quadrupled.
- "Lubuntu, SHA-512, 32-bit OS, Cryptsetup 1.7.0" ($261,396$ vs $1,146,912$, Table 1 and Table 2 respectively). Again, they quadrupled;
- "Fedora, SHA-512, 32-bit OS, Cryptsetup 1.6.8" ($63,331$ vs $261,398$, Table 1 and Table 2 respectively). Again;

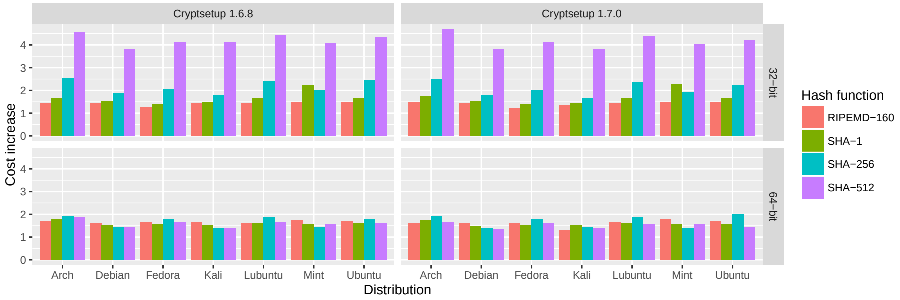
- “Arch, SHA-256, 64-bit OS, Cryptsetup 1.7.0” (1, 284, 213 vs 2, 455, 634, Table 1 and Table 2 respectively). In this case, the iteration count value is doubled.
- . . .

More in general, on 64-bit systems, Cryptsetup with OpenSSL used as the crypto backend produced between 1.5 and 2 times higher iteration counts as opposed to Cryptsetup with Libgcrypt. On 32-bit systems the following differences have been observed:

- for RIPEMD-160: 1.2-1.5 higher with OpenSSL vs. Libgrypt,
- for SHA-1: 1.4-1.7 higher with OpenSSL vs. Libgrypt (and 2.25 higher on Linux Mint),
- for SHA-256: 1.8-2.5 higher with OpenSSL vs. Libgrypt,
- for SHA-512: 3.8-4.5 higher with OpenSSL vs. Libgrypt.

Note that part of this increase is due to a performance bug [46] in PBKDF2 provided by Libgcrypt and fixed in Libgcrypt version 1.7.0 [48]. Figure 4 helps us to visualize these differences. The ratios in this graph have been obtained analogically to that of Figure 3 — i.e., by dividing the iteration count computed with OpenSSL by the corresponding Libgcrypt. Then, such ratios are compared across a number of configurations.

Fig. 4. Difference in attack cost in Cryptsetup backend OpenSSL vs Libgcrypt



The graph suggests that the choice of cryptographic library to be used as the backend for Cryptsetup has a significant impact on the resulting iteration counts and, consequently, the attack cost.

Linux Distributions: The last factor which may affect the iteration count computation is the distribution installed. In particular, we analyzed data gathered with a number of Linux desktop distributions and we found the following result:

- In “Table 1, SHA-1, 32-bit OS, Cryptsetup 1.6.8”, the iteration counts gathered are 279, 701 (Mint) and 387, 923 (Fedora);

- "Table 2, SHA-256, 32-bit OS, Cryptsetup 1.7.0" ($1,444,241$ vs $2,007,356$, Debian and Arch respectively);
- "Table 2, SHA-256, 32-bit OS, Cryptsetup 1.6.8" ($364,159$ vs $496,901$, Debian and Ubuntu respectively).

In cases listed above, the iteration count values are increased about 36-40 percent. Interestingly, the iteration count on the 32-bit Linux Mint with the Libgcrypt crypto backend is unusually low compared to other distributions. We were not able to identify the reason behind this deviation.

5 LUKS: Examining PBKDF2 security margin

In this Section we evaluate the estimated cost of attack on the password of a LUKS partition created using Cryptsetup. We assumed an optimized version of the attack where only key derivation PBKDF2 computation is needed. In general case the attack computation would have to include also keyslot decryption and master key checksum computation, which would increase the computation time by about 20-25% (see [21, section 4.1]).

To model an attacker's hardware, we used PBKDF2 benchmarks produced by an open-source tool [32] using the experimentally collected iteration counts listed in Section 4.2. To estimate the cost of cracking a password protected by PBKDF2, we chose specific system parameters — e.g., the derived key length, the electricity price, the purchase price of a single device, and so on — and imposed limitations to some of them — e.g., a maximum acceptable attack duration. Finally, we ran our code using different GPUs such as NVIDIA GeForce 465, NVIDIA Tesla M2090, NVIDIA Tesla K20, and NVIDIA Tesla K20X. More details about system parameters and GPUs can be found in Tables 9, 10, and 11.

Table 9. Attack estimation parameters.

Parameter name	Value	Comment
Derived key size	256 bits	
Password search space (S)	2^{48}	(8-character alphanumeric)
Max. attack duration (T)	5 years	
Price of electricity (E)	\$0.05 kWh^{-1}	[1]

5.1 Attack feasibility evaluation

We select two configurations of the user's system (see Table 12) between those listed in Tables 1 and 2. More precisely, we chose the configurations that produce the worst and best iteration count value.

Using Eq. (4) and multiplying this result by the cost of the respective device, we get an idea of the cost of the whole equipment needed to perform an attack

Table 10. Attack estimation parameters for GPU devices.

Device	Parameter name	Value	Comment
NVIDIA GeForce GTX 465	Power draw (P)	200 W	[11]
	Purchase price (D)	\$215	[11] (MSRP)
	Lifetime (L)	5 years	(conservative estimate)
NVIDIA Tesla M2090	Power draw (P)	250 W	[14]
	Purchase price (D)	\$2500	[14] (MSRP)
	Lifetime (L)	10 years	(conservative est., based on [6])
NVIDIA Tesla K20	Power draw (P)	225 W	[12]
	Purchase price (D)	\$3150	[12] (MSRP)
	Lifetime (L)	15 years	[6]
NVIDIA Tesla K20X	Power draw (P)	235 W	[13]
	Purchase price (D)	\$7700	[13] (MSRP)
	Lifetime (L)	15 years	[7]

Table 11. Measured PBKDF2 computation speeds for GPU devices

Device	Hash function	PBKDF2 computation speed (V)
NVIDIA GeForce GTX 465	SHA-1	178437745.7
	RIPEMD-160	138319407.9
	SHA-256	75482616.7
	SHA-512	24101794.6
NVIDIA Tesla M2090	SHA-1	291032649.1
	RIPEMD-160	219635621.6
	SHA-256	120639441.8
	SHA-512	37930735.9
NVIDIA Tesla K20	SHA-1	676654200.7
	RIPEMD-160	524113035.1
	SHA-256	178302193.3
	SHA-512	33038175.2
NVIDIA Tesla K20X	SHA-1	721176830.2
	RIPEMD-160	540764271.2
	SHA-256	191052691.1
	SHA-512	36163354.4

Table 12. Attack estimation user configurations.

Configuration	Distribution	Architecture	Backend	Cryptsetup version
“worst-case”	Mint	32-bit	Libgcrypt	1.6.8
“best-case”	Arch	64-bit	OpenSSL	1.7.0

in 5 years. Note that the actual cost might be higher or lower than this value – it has to include the cost of consumed electricity and, on the other hand, can be lower by the residual price of the hardware after performing the attack (we assume that the hardware can be resold or used for a different purpose after the attack). For simplicity, our model assumes that the price of the device decreases linearly with the usage time.

Fig. 5. The attack's initial hardware cost under "worst-case" configuration.

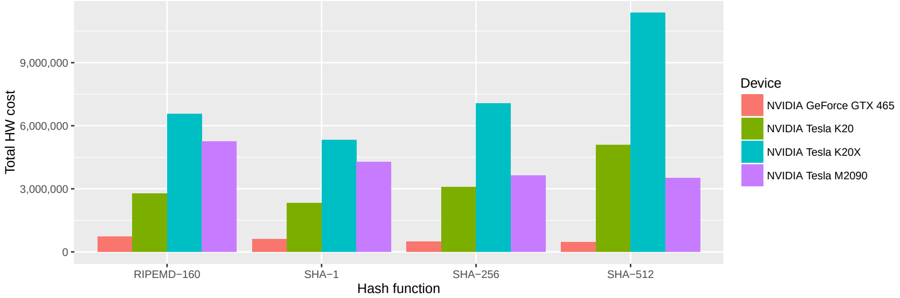
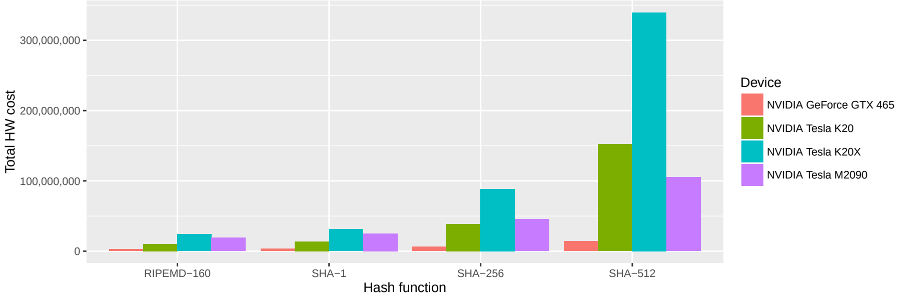


Fig. 6. The attack's initial hardware cost under "best-case" configuration.



Figures 5 and 6 show the hardware costs calculated for the "worst-case" and "best-case" configurations, respectively. Notice that for the cheap NVIDIA GeForce GTX 465 GPU, the initial costs are an order of magnitude lower ($500-700K for "worst-case") than for the other GPU devices ($2-11M for "worst-case").

The overall values of the costs suggest that an attack on LUKS passwords would require a very large amount of resources and would be feasible only for very wealthy attackers. Also, even in the weakest configuration, the attack would require thousands of GPU devices in order to be successful after 5 years on average. Such amount of devices might pose serious challenges and incur other costs (e.g., for the host computers, cooling, or physical storage), which we do not take into account.

5.2 Attack cost comparison

Figures 7 and 8 show the attack costs calculated for the "worst-case" and "best-case" configurations, respectively. Taking the minimum over all the GPU devices, we get a minimum cost of $1-1.5M for the "worst-case" configuration (for all hash functions). For the "best case" configuration, we get $5-7M with SHA-1 and RIPEMD-160 hash functions, $19M with SHA-256 and $43M with SHA-512.

Fig. 7. The attack cost under "worst-case" configuration.

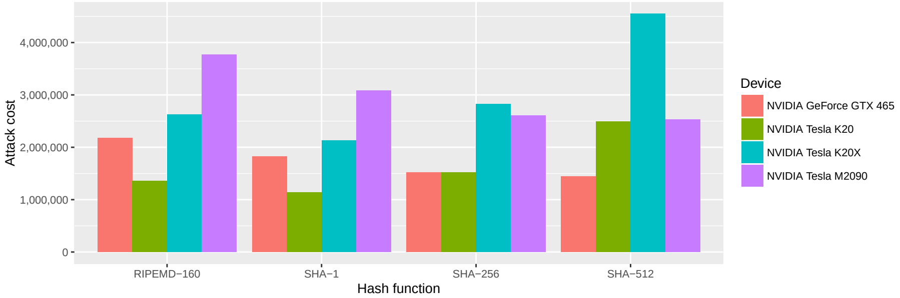
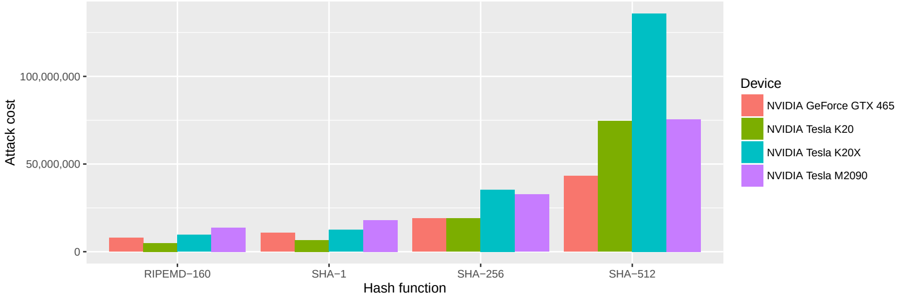


Fig. 8. The attack cost under "best-case" configuration.

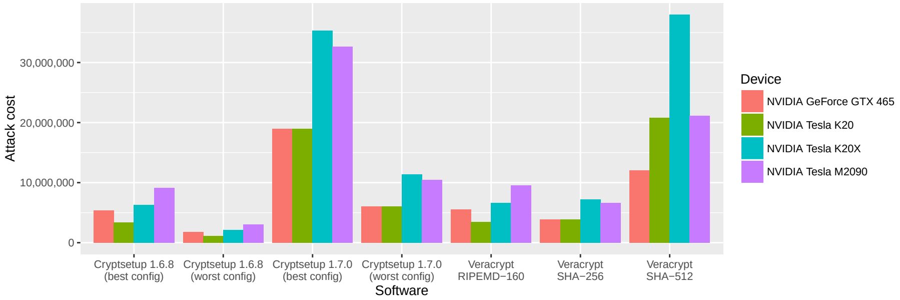


5.3 Cryptsetup vs. VeraCrypt

Finally, we compared the cost of brute-force attacks on a LUKS partition created by Cryptsetup versions 1.6.8 and 1.7.0 to the cost of attacking a VeraCrypt partition.

VeraCrypt is the successor of a popular disk encryption software for the Windows operating system, TrueCrypt, which has been discontinued on 28 May 2014 [5,16]. Both TrueCrypt and VeraCrypt use PBKDF2 as the key derivation function. One of the most criticised problems of the original TrueCrypt was that the PBKDF2 iteration count was fixed and extremely low (1,000-2,000 iterations). In VeraCrypt, the iteration count has been raised to 327,661-655,331 iterations (depending on the hash function), but is still fixed [10]. Since version 1.12, the user can specify an optional "PIM" value to personalize the iteration count. However, since PIM has to be remembered by the user together with the password, this feature is unlikely to be used by the majority of users [8].

Fig. 9. Attack cost of Cryptsetup vs VeraCrypt.



As seen in Figure 9, the estimated attack cost is comparable between VeraCrypt and Cryptsetup 1.7.0, while Cryptsetup 1.6.8 is falling behind both.

Note that due to the fact that the iteration count is set to a fixed value in VeraCrypt, the disk unlocking time may be very high on legacy hardware (especially if the SHA-512 hash function is used). Since Cryptsetup picks the iteration count dynamically, it avoids this problem (assuming the encrypted disk is always used on the same/similar HW-SW configuration), although the attack cost may be reduced if the PBKDF2 computation is slow.

6 Discussion and conclusions

Although Argon2 is expected to supersede PBKDF2 in the next few years, currently PBKDF2 is still widely implemented to derive keys in many security-related systems. In this paper, we addressed the problem of long-term passwords secured by PBKDF2, presenting the case study of LUKS. We (a) analyzed how external factors may affect the iteration counts computation used by PBKDF2 to slow down attackers, (b) provided means of evaluating the iteration count values defined by real-world applications, and finally (c) showed that PBKDF2 is — and will still remain for years to come — capable to provide enough security margin for applications that require long-term keys.

More precisely, our testing activities identified four external factors that cannot be defined "weaknesses" (and cannot be patched) but which negatively affect the iteration count computation, improving the speed of a brute-force attack:

1. *Library versions*: Experimental results show that a security or performance bug coded into lines of an older version of the library can considerably affect the iteration count computation and, consequently, the attack cost. In our testing activities, we found old library versions in Live CD distributions, but also those available in official repositories are far from being up-to-date. This means that the majority of Linux distributions tested are affected by well-know LUKS weaknesses described in literature.
2. *Architectures (32/64-bit OS)*: We show an important gap between the iteration count values computed using 32-bit or 64-bit configurations. Experimental results suggest that, in several 32-bit Linux distributions, iteration counts can be reduced to a quarter when compared with the same 64-bit distros and, consequently, the attack cost too. Hence, from the user's point of view, is more convenient use a 64-bit OS rather than a 32-bit OS.
3. *Backends*: The choice of cryptographic library to be used as the backend for Cryptsetup has a significant impact on the resulting iteration counts. Because the execution speed of cryptographic algorithms is due to independent optimizations coded in Libgcrypt and OpenSSL, our testing activities show that, on average, OpenSSL performs better than Libgcrypt.
4. *Linux distributions*: We analyzed the effects that a number of Linux distributions have on the iteration count computations. In some cases, the choice of a particular Linux distribution can help user to increase the iteration count values, and hence the attack cost, by about 36-40 percent. Moreover, we noted that the 32-bit Linux Mint with the Libgcrypt crypto backend performs below average, but we were not able to identify the reason behind this deviation.

Once external factors have been identified, we estimated the attack costs on the basis of the power consumed by the hardware doing the cracking and the cost of the hardware itself. Since there might be many other contributing factors — e.g., physical storage, cooling, or host computers — our estimations can only be seen as a lower bound for the actual cost.

Our results suggest that even in the weakest configuration, an attack on LUKS passwords would require a very large amount of resources — i.e., thousands of GPUs — in order to be successful in a reasonable amount of time (five years on average). In the worst-case scenario, which is based on the configurations that produce the worst iteration count value, the estimated cost is between \$1M and \$1.5M. In the best case scenario, which is based on the configurations that produce the best iteration count value, we get a minimum cost of \$5-7M with SHA-1 and RIPEMD-160 and a maximum cost of \$43M with SHA-512. Interestingly, the cost for SHA-512 is so high due to 64-bit arithmetic operations being less efficient on the GPU (as opposed to CPU).

7 Acknowledgment

Andrea Visconti was partially supported by a grant from Università degli Studi di Milano, “Piano di Sostegno alla Ricerca UNIMI 2015-2017”.

Ondrej Mosnáček and Vashek Matyáš were partially supported by the ITI centre of Czech Science Foundation, GBP202/12/G061. Ondrej Mosnáček and Milan Brož were also partially supported by Red Hat Czech. Computational resources were provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the programme “Projects of Large Research, Development, and Innovations Infrastructures”.

References

1. Electricity prices by type of user. Eurostat, `http://ec.europa.eu/eurostat/tgm/refreshTableAction.do?tab=table&plugin=1&pcode=ten00117&language=en`, [Online, accessed 10-October-2016]
2. EncFS Encrypted Filesystem, `https://sites.google.com/a/arg0.net/www/encfs`, [Online, accessed 10-October-2016]
3. GNU GRUB Manual, Version: 2.00, `http://www.gnu.org/software/grub/manual/grub.html`, [Online, accessed 10-October-2016]
4. RAR Archive Format, Version: 5.0, `http://www.rarlab.com/technote.htm`, [Online, accessed 10-October-2016]
5. TrueCrypt, `http://truecrypt.sourceforge.net/`, [Online, accessed 10-October-2016]
6. Tesla K20 GPU Accelerator Board Specification. NVIDIA Corporation (July 2013), `http://www.nvidia.com/content/PDF/kepler/tesla-k20-passive-bd-06455-001-v07.pdf`, [Online, accessed 10-October-2016]
7. Tesla K20X GPU Accelerator Board Specification. NVIDIA Corporation (July 2013), `http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf`, [Online, accessed 10-October-2016]
8. Veracrypt – PIM. IDRIX (February 2013), `https://veracrypt.codeplex.com/wikipage?title=Personal%20Iterations%20Multiplier%20%28PIM%29`, [Online, accessed 10-October-2016]
9. Password hashing competition (2015), `https://password-hashing.net/`, [Online, accessed 10-October-2016]
10. Veracrypt – Header Key Derivation, Salt, and Iteration Count. IDRIX (July 2015), `https://veracrypt.codeplex.com/wikipage?title=Header%20Key%20Derivation`, [Online, accessed 10-October-2016]
11. ASUS GeForce GTX 465 PCIe 2.0 Graphics Card 1GB GDDR5 Specifications. CBS Interactive Inc. (June 2016), `http://www.cnet.com/products/asus-geforce-gtx-465-pcie-2-0-graphics-card-1gb-gddr5/specs/`, [Online, accessed 10-October-2016]
12. NVIDIA Tesla K20 GPU computing processor - Tesla K20 - 5 GB Series Specifications. CBS Interactive Inc. (June 2016), `http://www.cnet.com/products/nvidia-tesla-k20-gpu-computing-processor-tesla-k20-5-gb-series/specs/`, [Online, accessed 10-October-2016]

13. NVIDIA Tesla K20X GPU computing processor - Tesla K20X - 6 GB Series Specifications. CBS Interactive Inc. (June 2016), `http://www.cnet.com/products/nvidia-tesla-k20x-gpu-computing-processor-tesla-k20x-6-gb-series/specs/`, [Online, accessed 10-October-2016]
14. NVIDIA Tesla M2090 GPU computing processor - Tesla M2090 - 6 GB Specifications. CBS Interactive Inc. (June 2016), `http://www.cnet.com/products/nvidia-tesla-m2090-gpu-computing-processor-tesla-m2090-6-gb-teslam2090/specs/`, [Online, accessed 10-October-2016]
15. PHP: password_hash – Manual. The PHP Group (2016), `https://php.net/manual/en/function.password-hash.php`, [Online, accessed 10-October-2016]
16. VeraCrypt – Home. IDRIX (April 2016), `https://veracrypt.codeplex.com/`, [Online, accessed 10-October-2016]
17. Apple Inc.: Best Practices for Deploying FileVault 2. Tech. rep. (2012), `http://training.apple.com/pdf/WP_FileVault2.pdf`
18. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Proceedings of Advances in Cryptology—CRYPTO'96. pp. 1–15. Springer (1996)
19. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions—the HMAC construction. RSA Laboratories CryptoBytes 2(1), 12–15 (1996)
20. 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`
21. 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)
22. Brož, M.: Cryptsetup 1.7.0 Release Notes (2016), `https://www.kernel.org/pub/linux/utils/cryptsetup/v1.7/v1.7.0-ReleaseNotes`, [Online, accessed 10-October-2016]
23. 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. Tech. rep., The U.S. National Institute of Standards and Technology (August 2013), `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf`
24. 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`
25. Forler, C., Lucks, S., Wenzel, J.: Catena : A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Report 2013/525 (2013)
26. Fruhwirth, C.: New methods in hard disk encryption (2005), `http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`
27. Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.2.2 (2016), `https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf`
28. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA. vol. 14 (1996)
29. IEEE 802.11 WG: Part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Std 802.11 i-2004 (2004)

30. 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`
31. 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`
32. Mosnáček, O.: PBKDF2-GPU – A C++11 library for brute-forcing PBKDF2 using GPU and a tool for brute-forcing LUKS passwords. (2015), `https://github.com/WOnder93/pbkdf2-gpu`
33. NIST: FIPS PUB 180-4: Secure Hash Standard (Mar 2012), `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`
34. Percival, C., Josefsson, S.: The scrypt Password-Based Key Derivation Function. RFC 7914, RFC Editor (August 2016), `https://www.rfc-editor.org/rfc/rfc7914.txt`
35. Percival, C.: Stronger key derivation via sequential memory-hard functions (May 2009), `https://www.tarsnap.com/scrypt/scrypt.pdf`, [Online, accessed 10-October-2016]
36. Percival, C.: Re: scrypt time-memory tradeoff (June 2011), `http://mail.tarsnap.com/scrypt/msg00029.html`, [Online, accessed 10-October-2016]
37. 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]
38. Peslyak, A., Marechal, S.: Password security: past, present, future (presentation). Openwall, Inc. (December 2012), `http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/`, [Online, accessed 10-October-2016]
39. Pornin, T.: The MAKWA Password Hashing Function (April 2015), `http://www.bolet.org/makwa/makwa-spec-20150422.pdf`, [Online, accessed 10-October-2016]
40. Provos, N., Mazieres, D.: A future-adaptable password scheme. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, FREENIX Track. pp. 81–91 (1999)
41. Rege, A.: An Introduction to Modern GPU Architecture. NVIDIA Corporation (2016), `ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`, [Online, accessed 10-October-2016]
42. RSA Laboratories: PKCS #5 V2.1: Password Based Cryptography Standard (2012)
43. Shannon, C.E.: Prediction and entropy of printed english. Bell system technical journal 30(1), 50–64 (1951)
44. 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)
45. Turan, M.S., Barker, E.B., Burr, W.E., Chen, L.: Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications. Tech. rep., The U.S. National Institute of Standards and Technology (December 2010), `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf`
46. 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)
47. Visconti, A., Gorla, F.: Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2. Cryptology ePrint Archive, Report 2018/097 (2018), `https://eprint.iacr.org/2018/097.pdf`

48. Yutaka, N.: Libgcrypt 1.7.0 Keep contexts for HMAC in GcryDigestEntry (2015), `http://git.gnupg.org/cgi-bin/gitweb.cgi?p=libgcrypt.git;a=commit;h=f7505b550dd591e33d3a3fab9277c43c460f1bad`, [Online, accessed 10-October-2016]