UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

ARGON2 E LUKS: ANALISI CRITICA DI UNA IMPLEMENTAZIONE BASATA SU FUNZIONI CRITTOGRAFICHE MEMORY-HARD

Relatore: Prof. Andrea Visconti

Correlatore: Dr.ssa Michela Ceria

Tesi di:
Gabriele Quagliarella
Matricola: 870773

Anno Accademico 2018-2019

Ai miei genitori, a mio fratello e a tutti i miei familiari,
che hanno reso possibile la realizzazione di questo importante traguardo.
Alla mia fidanzata Giulia, la cui forza e amore
mi hanno aiutato a superare qualsiasi ostacolo sul mio cammino.
A me medesimo, per ricordarmi di non mollare mai,
poichè è solo nelle difficoltà che si può scoprire la propria vera forza...

Preface

The disk encryption is the main protection that users might adopt in order to prevent that an attacker, who has physical access to the disk, would be able to steal private information stored on it.

The research presented here is about the security evaluation of the new cryptographic algorithm *Argon2* and its integration into the de-facto standard Linux encryption tool: *cryptsetup*.

Since the currently most used key derivation function (i.e. *PBKDF2*) is affected by some security weaknesses [19], the international cryptographic community conducted the Password Hashing Competition (PHC) in order to identify a new efficient and more secure password-hashing schemes. *Argon2* has been selected as the winner of the competition.

The goal of this research is about finding some condition on the operating system which might affect the security of the encrypted disk when *Argon2* is chosen. To do that, most of the efforts have been pointed to the actual implementation of its benchmark function, necessary to set the initial values of the algorithm.

Before showing the results of this work [4.4], the internals of *cryptsetup* will be described into Chapter 2, the design of the new *Argon2* algorithm is shown in Chapter 3 and the weaknesses of the actual key derivation function will be presented in Section 2.3.

At the end, a statistical-based solution will be proposed in Chapter 5 in order to prevent that a malicious process would be able to lower the security of the disk encryption when *Argon2* is used as the key derivation function.

Contents

ii

Preface **iv**

1 Introduction **1**
- 1.1 Disk-encryption 1

2 Cryptsetup **3**
- 2.1 dm-crypt 3
- 2.2 LUKS 4
  - 2.2.1 Motivation 4
  - 2.2.2 Design 5
  - 2.2.3 Master key recovery 6
- 2.3 Weaknesses of PBKDF2 7

3 Argon2 **10**
- 3.1 Motivation 10
- 3.2 Features 11
- 3.3 Input parameters 12
- 3.4 Design 12
- 3.5 Compression function G 14
- 3.6 Variable-length hash function 16

4 Testing Argon2's benchmark function **17**
- 4.1 Threat model 17
- 4.2 Analysis of the benchmark function 19
- 4.3 Attacking the CPU 27
- 4.4 Results 32

5 Considerations and proposed solution **36**

Chapter 1

Introduction

1.1 Disk-encryption

In our ages, personal information became a sensitive resource that - if non-properly protected - can be used by attackers to cause damage to assets and also people themselves. Slowly, due to the low cost of the actual USB and in general of physical storage devices, more and more resources have been converted to the digital world: private documents, photo, medical insurance documents, and so on.

Even if there are more then just one solution in order to secure private data, it is very common that these mechanisms are not enabled by default or are intentionally disabled to improve the performance of the disk on which the data are saved.

This condition exposes users to a critical security issue: without protection, all the data saved into the hard drive could be easily read by an external attacker which has physical access to the device.

The disk encryption is the process which enables to protect all the information saved on a physical hard drive just by encrypting them with a symmetric key provided by the user [10].

All disk encryption methods operate in such a way that even though the disk actually holds encrypted data, the operating system and applications "see" it as the corresponding normal readable data as long as the cryptographic container (i.e. the logical part of the disk that holds the encrypted data) has been "unlocked" and mounted.

This process usually takes place during the bootstrapping phase in which, if the key supplied by the user is correct, the operating system is able to retrieve the correct master key necessary to decrypt each block of the hard drive. Otherwise, if an unauthorized person would insert the wrong password, then he will only find garbled random-looking data instead of the actual files [7]. More details about this process will be described in Section 2.2.

Since 15 years ago, most of the operating systems have started to integrate disk encryption in their code: Apple introduced into Mac OS X Tiger the first version of *FileVault*, while Microsoft deployed *Bitlocker* with Windows Vista.

In the open-source world, in particular in the Linux environment, there are a lot of alternative software that implement it. Among all of them, the solution that has become the de-facto standard on the Linux kernel is *dm-crypt* 2.1 [15].

Even if the disk encryption process in a Linux-like operating system has become easier to handle thanks to software like *cryptsetup*, this procedure involves an advanced experience in order to create and then manage the encrypted partition: in addition to the steps necessary to prepare and format the disk, the user has to define the initial parameters that will be used by the cryptographic algorithms involved into the disk encryption process.

In order to simplify this last step, the developers of *cryptsetup* created a list of benchmark functions that are used when the user does not specify all the required parameters: running dynamic tests, each of these benchmark functions has the goal to find sub-optimal values for a specific cryptographic algorithm so that the ciphertext produced is the strongest possible and the encryption/decryption phase does not take more then 2 seconds.

As discovered in other reseaches [4], some external factors could influence the parameters used by the encryption process.

For this reason, the goal of this research was to evaluate the security of the benchmark function of the state-of-the-art key derivation function: *Argon2*. In particular, the target was to find some states of the operating system, called *corner cases*, which could effect the benchmark and lead it to to find lower values than what it might do in a normal condition.

More details about the findings and the results are found into Chapter 4.

Chapter 2

Cryptsetup

2.1 dm-crypt

dm-crypt is a transparent disk encryption subsystem since Linux kernel version 2.6. It is the most used block device encryption solution, which means that it operates below the file-system layer and it make sure that everything written to a certain block device is encrypted. *dm-crypt* is part of the device mapper infrastructure, which is a framework provided by the Linux kernel for mapping physical block devices onto higher-level virtual block devices. In particular, using the kernel's Crypto API, dm-crypt provides a transparent encryption of block devices which gives to the user the flexibility to encrypt any disk-backed file systems supported by the operating system.

This means that the user can basically specify only the symmetric cipher, the encryption mode and the passphrase in order to create a new block device in the */dev* directory on which each writes will be transparently encrypted and each reads decrypted.

Usually, *dm-crypt* is only used for low-level access to volumes and is not so easy to use. For this reason, the interface used to create and manage a dm-crypt device is the *cryptsetup* [5] tool which easily allows to create and handle several types of dm-crypt block-device format like LUKS, plain, loopAES and Truecrypt.

The mechanisms used by *dm-crypt* in order to manage the encrypted device require multiple cryptographic algorithms and each of them has a variable length of arguments that has to be tuned.

For this reason, *cryptsetup* implements a suite of benchmark functions that has the goal to give to the user a concrete means for selecting the best combination of algorithms and the corresponding parameters that maximizes the efficiency on the system.

This function is also used when the user does not provide one or more cryptographic parameter necessary to initialize the disk creation process. In this situation, *cryptsetup* can not use some fixed values because they would not be guarantee both security and efficiency.

It is here that the benchmark function plays a central role: after a variable number of iterations, the function would find sub-optimal values for each parameter of the algorithms involved into the encryption process. These parameters are selected so that the cipher-text produced is the strongest possible and the computation necessary to unlock the disk is acceptable in terms of time.

As can be well imagined, the realization of the benchmark function is an hard problem. Each cryptographic algorithm needs its own function and usually the creators are not the same that produce the benchmark. The security effects related to this condition will be discussed into the Chapter 4.

2.2 LUKS

2.2.1 Motivation

The password management is one of the main problems that arise in the device encryption process when a full-disk encryption it is chosen (e.g *FDE*) solution. Indeed, since the master-key used to encrypt the device is encrypted, as well as all the other sectors of the disk, it is necessary to adopt a solution that has the following requirements:

- It must handle multiple users each of one having a different password.
- Changing the password of an user does not imply the re-encryption of the whole disk.
- Given each user's password, it is possible to retrieve the master-key necessary to unlock the disk.
- Usually passwords chosen by users are not complex and involve a small range of characters. This condition advantages an external attacker that gains physical access to the disk: in this situation, he would be able to perform a brute-force attack against the stored password, retrieve the master key and unlock the disk.

As said in the previous section, cryptsetup can handle more then one block-device format. Among these, LUKS format solves all the described problems and gives a platform-independent standard on-disk format.

It does not only facilitates compatibility and interoperability among different programs, but also assures that they all implement password management in a secure and documented manner.

2.2.2 Design

The Linux Unified Key Setup (LUKS) [8] [9] is a disk-encryption specification commonly implemented in Linux based operating systems.

Because of its versatility, since its creation in 2004 by Clemens Fruhwirth, it has become one of the most used formats in the Linux open-source environment.

In order to solve some of the problems described before, like the re-encryption of the disk after that an user changes his password, LUKS adopts a two level key hierarchy. In concrete, the key used to really encrypt each block of the disk, which takes the name of *master-key*, is not defined by the user, but instead, it is generated by a PRNG function.

This *master-key* is then encrypted using a derived key from the passphrase of the user. The method used to compute the derived key usually involves a *key derivation function* (e.g KDF). This function has the property to take a source of initial keying material and derive from it one or more pseudo-random keys.

Introducing CPU-intensive and cycle-free operations, this approach based on a KDF not only slows down a brute force attack as much as possible, but also allows to increase the size of a cryptographic key.

By default, the reference implementation of LUKS format uses *PBKDF2* algorithm as its key derivation function. More details about this KDF will be described in Section 2.3.

In order to handle multiple users and passphrases that can be individually revoked or changed independently, LUKS has to introduce and additional layer which stores all of the needed setup information for dm-crypt on the disk itself: a standardized header at the start of the device and a key-slot area directly behind it.

The partition header contains the following information:

- Encryption algorithm
- Key length
- Hash function
- Master key checksum
- Salt
- Iteration count

Instead, since the same master key can be encrypted in multiple cipher-text, LUKS sets up to eighth the number of key-slots to use in order to save the key material of each user.

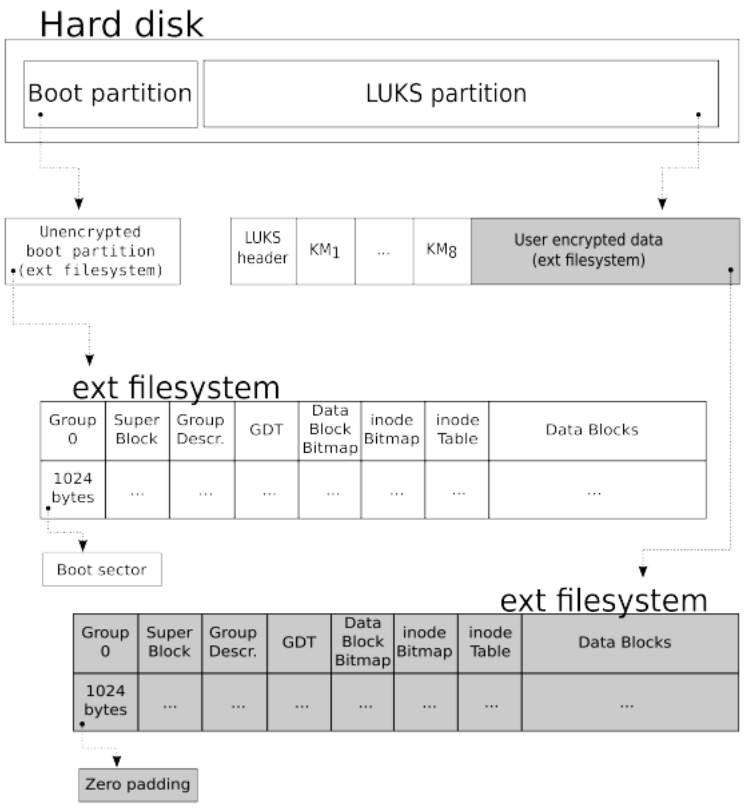

Figure 1: LUKS partition

2.2.3 Master key recovery

During the boot of the operating system the master key recovery process takes place. This phase involves that the user inserts his credentials and the operating system chooses the right key-slot necessary to validate them.

In particular, the steps performed in order to decrypt the *master-key* and then unlock the disk can be summarized into the following steps:

1. Read the user password p
2. Read salt s from the selected key slot
3. Read first iteration count c from active key slot
4. Use PBKDF2 to compute derived key $DerKey$
5. Read the start sector of key material from active key slot

6. Read the split master key from key material
7. Decrypt the split master key using derived key *DerKey*
8. Merge the split encrypted master key and obtain a candidate master key
9. Read the second iteration count for computing the master key digest
10. Use PBKDF2 to compute the candidate master key digest
11. Compare such digest with those stored in the partition header
12. If equal, the recovery is successful. Otherwise, the candidate is not the correct master key and the process ends.

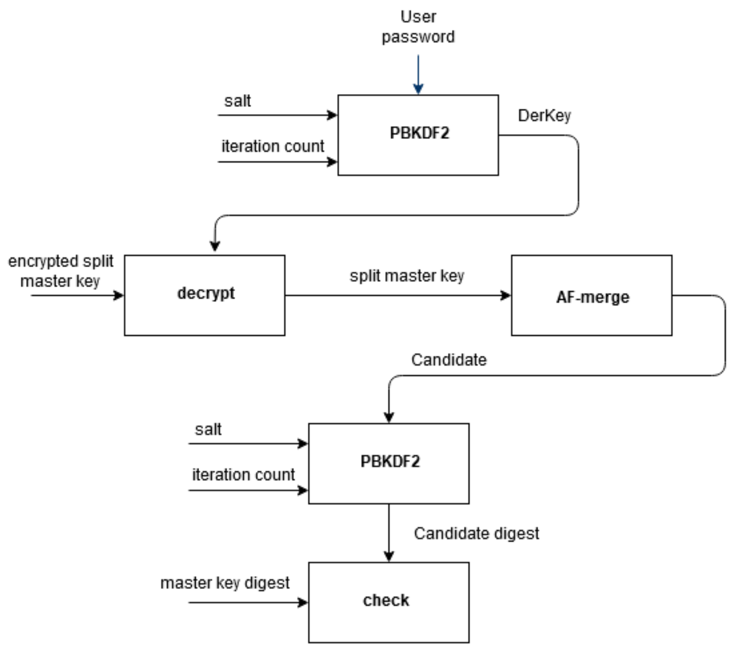

Figure 2: Master key process

2.3 Weaknesses of PBKDF2

PBKDF2 is the default key derivation function used into the LUKS format in order to protect the master key from an external attacker. Thanks to the usage of an pseudo-random function (*PRF*), PBKDF2 introduces CPU-intensive operations which provide a better resistance against brute force attacks.

It is important to notice that each of these CPU-intensive operations has to be cycle-free in order to avoid that the attacker would be able to get the derived key by executing a set of functionally-equivalent instructions.

In concrete, PBKDF2 takes the following parameters in order to compute the resulting derived key *DerKey*:

- PRF: pseudo-random function
- p: the user password
- s: a random salt
- c: an iteration count
- keyLength: length of the derived key

Furthermore, the design of PBKDF2 can be described as follows:

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

$$DerKey = T_1 \| T_2 \| ... \| T_n \tag{2}$$

where each T_n value is calculated as follows:

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

The *Function* is defined as:

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

and each U_c block is computed as:

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

Even if *PBKDF2* provides a better security strength then other cryptographic solutions like a simple hash function, it has some criticality that could be exploited by an attacker.

In particular, the first problem arises from the choice of the *pseudo-random function* (PRF). Even if the algorithm does not define any restrictions, in practice HMAC-SHA1 is the most common solution. As pointed out in several researches [19] [18], if HMAC-SHA-1 is computed in a standard mode without following the performance improvements described in the implementation note of RFC 2104 [11], an attacker is able avoid 50% of PBKDF2's CPU intensive operations, by replacing them with precomputed values [17].

Furthermore, the actual implementation of PBKDF2 has no parameter to increase the necessary memory usage. This means that an attacker has no need to have a large amount of memory in order to break it.

For the reasons exposed above, PBKDF2 can no longer provide the best available protection as a key derivation function.

Chapter 3

Argon2

3.1 Motivation

Until a few years ago, the hash functions were considered one of the most secure and at the same time easy-to-use cryptographic instrument. Message integrity, digital signature and proof-of-work are just a few example of common applications in which the hash functions are applied.

However, the latest researches in the cryptography field have proved that this fact is no more valid [16] [20]. Thanks to the increase of the speed of modern computational units, the cost of a password attack has been greatly reduced. At the same time, the event of new architectures like FPGA, ASIC module and GPU decreases the amount of time needed to compute hash functions, enabling an attacker to easily break low-entropy passwords in a small amount of time.

The first step adopted in order to reduce this computational advantage has been done by designing memory-hard functions. These new algorithms base their strength on the fact that the devices created in order to break the actual hash functions are great when the computation is almost memory-less, but they experience difficulties when operating on a large amount of memory [14] [1].

Since the start of the hash functions era, the design of a memory-hard function has been always a tough problem. The main reason is that many cryptographic problems that seemingly require large memory, actually allow for a time-memory trade-off [3]. This situation occurs for example when an algorithm trades increased space usage with decreased time. A concrete example of this problem is the usage of "Rainbow tables": using partially precomputed values in the hash space of a cryptographic hash function, the attacker is able to trade space, required to store the content of the table, in order to speed up the computational process.

For all of these reasons, the NIST institute organized in 2013 a "Password Hasing Competition" with the goal to select a new hash scheme which is secure against cracking-optimized systems, side-channel attacks and parallels attack. Among all the proposals, *Argon2* has been selected as the winner of the competition.

3.2 Features

As mentioned in the previous section, the *Argon2* [2] algorithm summarizes the state of the art in the design of memory-hard functions. It can be used to hash passwords for credential storage, key derivation, or other applications. Argon2 has three variants:

- **Argon2d**: it is the fastest version. It uses data-depending memory access so that it is highly resistant against GPU cracking attacks and it is suitable for applications with no threats from side-channel timing attacks (e.g. cryptocurrencies).
- **Argon2i**: unlike the previous one, Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. This feature makes it slower as it makes more passes over the memory to be safe against trade-off attacks.
- **Argon2id**: it is last version. It is an hybrid of Argon2i and Argon2d. It uses a combination of data-depending and data-independent memory accesses, which gives some of Argon2i's resistance to side-channel cache timing attacks and much of Argon2d's resistance to GPU cracking attacks.

In addition, main cryptographic features of Argon2 can be summarized as follows:

1. **Performance**: Both Argon2i and Argon2d securely fill the memory very fast (they spend about 2 CPU cycles per byte). This makes Argon2 suitable for applications that need memory-hardness but can not allow much CPU time.
2. **Trade-off resilience**: Argon2 provides a great level of trade-off resilience. In fact, applying common trade-off attacks, it has been shown that even an ASIC-equipped attacker can not decrease the time-area product if the memory is reduced.
3. **Scalability**: the algorithm is scalable both in time and memory. In fact, the user is able to change independently both of these parameters.
4. **Parallelism**: it is possible to use up to 2^{24} threads in parallel.
5. **GPU/FPGA/ASIC resistance**: Argon2 is optimized for the x86 architecture. For this reason, using a dedicated hardware should not help the cracking procedure.

3.3 Input parameters

Argon2 has been designed to have two types of input: primary and secondary inputs. In particular, the primary inputs are:

- **Message $\boldsymbol{P}$**: the password that has to be hashed. It may have any length from 0 to $2^{32} - 1$ bytes.
- **Nonce $\boldsymbol{S}$**: the salt used in order to prevent pre-computed hash attack. It may have any length from 8 to $2^{32} - 1$ bytes.

Instead, the secondary inputs are:

- **Degree of parallelism $\boldsymbol{p}$**: Specifies the number of threads that could be used. It is suggested to restrict this value at most to the number of cores of the CPU used for the computation. Anyhow, at least theoretically, it could be any integer between 1 to $2^{32} - 1$.
- **Tag length** τ: maybe any integer between 4 to $2^{32} - 1$.
- **Memory size $\boldsymbol{m}$**: It is the memory filled (in Kbytes) by the algorithm. It can be any integer number between $8p$ to $2^{32} - 1$.
- **Number of iterations $\boldsymbol{t}$**: define the number of rounds of computation performed on the memory. It can be any integer number from 1 to $2^{32} - 1$.

3.4 Design

In the current section we will explain the logic and the steps executed by Argon2 algorithm.

Before starting, it is important to remark that Argon2 internally uses a compression function G (3.5), a hash function H (3.6) and a variable-length hash function H'.

All these components could be also chosen by the user but in the reference implementation of Argon2 are already set. In particular the H function is represented by the *Blake2b* hash function.

In order to get a clearer analysis of the Argon2 algorithm, the internals of these functions are described in the following dedicated sections. At the moment, for the comprehension of the algorithm, it is only important to know that G takes two 1024-byte long inputs value and computes a 1024-byte output. Instead, H hash function produces a 64-byte value.

The first step performed by Argon2 is to extract entropy from the input parameters chosen by the user. In practice, it concatenates the parameters and then it computes the H hash function:

$$H_0 = H(p, \tau, m, t, \big|P\big|, P, \big|S\big|, S) \tag{4}$$

where $\big|\mathrm{P}\big|$ and $\big|\mathrm{S}\big|$ are respectively the length of password and the salt used.

After that, Argon2 has to set up the matrix used into the whole process. All the operations performed by the Argon2 are done on this structure. It is defined starting from the m parameter chosen by the user as secondary input. Since the matrix is designed to be organized in 1024-byte blocks, the m has to be manipulated in order to satisfy this constraint. In particular, the number of blocks actually used for the matrix is found by rounding down the m value to the nearest multiple of *4p*:

$$m' = \lfloor \frac{m}{4 * p} \rfloor \tag{5}$$

Finally, the memory is organized as matrix $B[i][j]$ of blocks with p rows and $q = m'/p$ columns. The motivation behind this choice is that in this way each row p can be computed by an independent (but synchronized) thread of the operating system, making the computational process more efficient.

At the first iteration [1] the blocks are computed as follows:

$$B^1[i][0] = H'(H_0 \| 0 \| i), \qquad 0 \leq i < p \tag{6}$$

$$B^1[i][1] = H'(H_0 \| 1 \| i), \qquad 0 \leq i < p \tag{7}$$

$$B^1[i][j] = G(B^1[i][j-1], B^1[i'][j']), \qquad 0 \leq i < p, 2 \leq j < q \tag{8}$$

It is important to notice that the logic behind the choice of the block index $[i'][j']$ is quite complex. We can resume it by saying that it depends on the version of Argon2 used: *Argon2i* or *Argon2d*. In particular, *Argon2i* uses a data-independent addressing logic concretized with a simple PRNG. Instead, the data-dependent version, *Argon2d*, fixes the value to $B[i][j-1]$. In both cases, the chosen values are then passed to a mathematical function which maps them to the final value.

If $t > 1$ then the operations on the matrix are repeated for another round. The only difference with the previous step is that the new blocks are XORed with the old values instead of overwriting them:

$$B^t[i][0] = G(B^{t-1}[i][q-1], B[i'][j']) \oplus B^{t-1}[i][0]; \tag{9}$$

[1]The block computed in the iteration t is denoted by $B^t[i][0]$

$$B^t[i][j] = G(B^t[i][j-1], B[i'][j']) \oplus B^{t-1}[i][j]; \tag{10}$$

At the end of the T iterations over the memory, the final block B_{final} is computed as the XOR of the last column of the matrix:

$$B_{final} = B^T[0][q-1] \oplus B^T[1][q-1] \oplus ... \oplus B^T[p-1][q-1] \tag{11}$$

which is then passed to the H' function to get the output tag:

$$Tag \leftarrow H'(B_{final}) \tag{12}$$

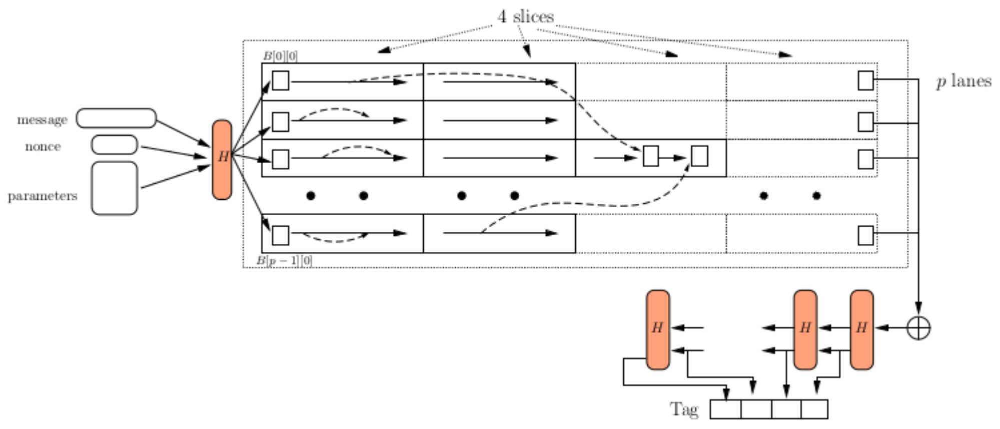

Figure 3: Argon2 execution flow

3.5 Compression function *G*

The compression function G is built upon the Blake2b round function P. P operates on the 128-byte input, which can be viewed as 8 16-byte registers:

$$P(A_0, A_1, ..., A_7) = (B_0, B_1, ..., B_7) \tag{13}$$

Compression function $G(X, Y)$ operates on two 1024-byte blocks X and Y . It first computes $R = X \oplus Y$. Then R is viewed as a 8×8 matrix of 16-byte registers $R_0, R_1, ..., R_{63}$. Then P is first applied row-wise, and then column-wise to get Z:

$$\begin{gathered}
(Q_0, Q_1, ..., Q_7) \leftarrow P(R_0, R_1, ..., R_7);\\
(Q_8, Q_9, ..., Q_{15}) \leftarrow P(R_8, R_9, ..., R_{15});\\
...\\
(Q_{56}, Q_{57}, ..., Q_{63}) \leftarrow P(R_{56}, R_{57}, ..., R_{63});
\end{gathered}$$

$$\begin{gathered}
(Z_0, Z_8, Z_{16}, ..., Z_{56}) \leftarrow P(Q_0, Q_8, Q_{16}, ..., Q_{56});\\
(Z_1, Z_9, Z_{17}, ..., Z_{57}) \leftarrow P(Q_1, Q_9, Q_{17}, ..., Q_{57});\\
...\\
(Z_7, Z_{15}, Z_{23}, ..., Z_{63}) \leftarrow P(Q_7, Q_{15}, Q_{23}, ..., Q_{63});
\end{gathered}$$

After that, G computes its output as $Z \oplus R$:

$$G : (X, Y) \to R = X \oplus Y \xrightarrow{\mathrm{P}} Q \xrightarrow{\mathrm{P}} Z \xrightarrow{\mathrm{P}} Z \oplus R \tag{14}$$

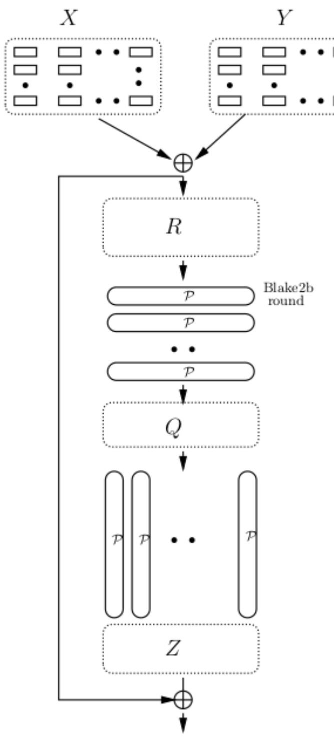

Figure 4: Compression function G

3.6 Variable-length hash function

Let us define H_x as an hash function with x-byte output. In the current implementation of Argon2 it is set to be the *Blake2b* function which supports a variable output x $1 \leq x \leq 64$. After that we can define H' as follows:

```
if $\tau < 64$ then
    $H'(X) = H_\tau(\tau \| X)$
else
    $r = \lceil \tau/32 \rceil$ - 2
    $V_1 \leftarrow H_{64}(\tau \| X)$
    $V_2 \leftarrow H_{64}(V_1)$
    ...
    $V_r \leftarrow H_{64}(V_{r-1})$
    $V_{r+1} \leftarrow H_{\tau-32r}(V_r)$
    $H'(X) = A_1 \| A_2 \| ... \| A_r \| V_{r+1}$
end if
```

Chapter 4

Testing Argon2's benchmark function

4.1 Threat model

As already described in Section 2.2.3, the KDF is a fundamental piece of the master key recovery process and its resistance against brute force attacks is one the main protections of the two level key hierarchy structure adopted by the LUKS format.

Since the initial versions of "Cryptsetup", the default KDF used into the encryption process was PBKDF2. Due to the weaknesses described in Section 2.3, the developer group has started to integrate into LUKS2 the support for the winner of the 2015 Password Hashing Competition: Argon2.

For now, *libcryptsetup* contains an embedded copy of the reference implementation of Argon2, which is easily portable to all architectures. The goal of the development group is to switch to an external implementation, once Argon2 becomes available in common cryptographic libraries.

As well as for *PBKDF2*, also Argon2 has a set of parameters which has to be tuned, so that the key derivation function can work and, as one might expect, the choice of them is strictly correlated to the strength of the hashed passwords.

All default parameters can be set on the command line by using the following options:

1. –pbkdf: choose between Argon2i or Argon2id;
2. –pbkdf-memory: select the amount of memory;
3. –pbkdf-parallel: set the number of thread used into the computation process;
4. –pbkdf-force-iterations: tune the number of performed iterations.

If the user does not supply all the needed parameters, then "Cryptsetup" runs a benchmark function that tries to find optimal values for the current system on which the encryption process takes place.

More details about the benchmark function and its usage will be discussed in Section 4.2. For now, it is only important to introduce the concept that some of the parameters used here (or even all) are selected by this benchmark function.

The mere usage of the benchmark function is largely adopted by most of the users. For this reason, it is fundamental that the selected parameters guarantee an optimal balance between performance and usability, but also the maximal possible level of security.

As you might expect, this propriety is very difficult to reach, and there are no optimal algorithms to solve the problem.

For this reason, it is very important that the code which implements the benchmark function should undergo a large number of test cases which try to increase the reliability of the program.

Being the benchmark function primarily influenced by the environment, the test cases here are represented by different states of the operating system.

A state is the equivalent of a picture of the environment in a specific moment of its execution and it is characterized by a quite big number of features, like used percentage of memory, the number of physical cores of the CPU and its load, the running processes, ...

Each of these feature has an influence on the parameters found by the benchmark function and the combination of them constitutes a *state*.

Since the number of tests is very large, it is more convenient to find some limited number of cases, called *corner cases*. They constitute a subset of states which have more influence on the execution of the benchmark function and that may lead to unexpected errors.

There is a large number of issues that may happen during the execution, but we are interested only in those which may lead to a downgrade of the parameters found and consequently to a lowering of the strength of the encryption.

In summary, we stated that we are interested in finding the corner cases which cause a fault to the benchmark function, so that it produces some values lower than what it might do in a stable condition.

These particular states are highly sought after by a possible attacker who wants to decrypt the content of our disk. If he had the ability to influence the system, so that the values produced by the benchmark function are weak, then he would be able to attack the encrypted disk with much lower computational cost.

The threat model used to describe the attack scenario has the following constraints:

1. the attacker has the ability to create one or more malicious processes;

2. these processes have to affect the system so that it turns out to be slower;
3. this condition has to affect the benchmark/encryption process;
4. this condition should avoid to take the operating system to an inconsistent state, because otherwise it would be killed by the operating system itself [13].

4.2 Analysis of the benchmark function

In the last Section 4.1, we have discussed about the role and the importance of the benchmark function, giving the context where it lives.

Here instead, we will focus on the analysis, starting from the surface and the visible effects that it produces (the parameters it finds) and then we will dive into the source code involved in the execution.

To have a concrete example of how the benchmark function works in practice, the output of the execution of the "cryptsetup" tool is reported.

Choosing the "benchmark" option, "cryptsetup" allows the execution of the benchmark function (calling the "crypt_benchmark_pbkdf()" function) and prints the output parameters.

```
gabriele@localhost:~$ cryptsetup benchmark –debug

# Running argon2i() benchmark.
# PBKDF : memory cost = 32, iterations = 4, threads = 4 (took 5 ms)
# PBKDF : memory cost = 512, iterations = 4, threads = 4 (took 7 ms)
# PBKDF : memory cost = 8192, iterations = 4, threads = 4 (took 22 ms)
# PBKDF : memory cost = 93090, iterations = 4, threads = 4 (took 170 ms)
# PBKDF : memory cost = 136897, iterations = 4, threads = 4 (took 232 ms)
# PBKDF : memory cost = 147518, iterations = 4, threads = 4 (took 235 ms)
# PBKDF : memory cost = 156934, iterations = 4, threads = 4 (took 234 ms)
# PBKDF : memory cost = 167664, iterations = 4, threads = 4 (took 274 ms)
# Benchmark returns argon2i() 4 iterations, 1048576 memory, 4 threads
argon2i 4 iterations, 1048576 memory, 4 parallel threads (CPUs)

# Running argon2id() benchmark.
# PBKDF : memory cost = 32, iterations = 4, threads = 4 (took 2 ms)
# PBKDF : memory cost = 512, iterations = 4, threads = 4 (took 2 ms)
# PBKDF : memory cost = 8192, iterations = 4, threads = 4 (took 32 ms)
# PBKDF : memory cost = 64000, iterations = 4, threads = 4 (took 135 ms)
# PBKDF : memory cost = 118518, iterations = 4, threads = 4 (took 196 ms)
# PBKDF : memory cost = 151170, iterations = 4, threads = 4 (took 249 ms)
# PBKDF : memory cost = 151777, iterations = 4, threads = 4 (took 249 ms)
# PBKDF : memory cost = 152386, iterations = 4, threads = 4 (took 228 ms)
# PBKDF : memory cost = 167089, iterations = 4, threads = 4 (took 238 ms)
# PBKDF : memory cost = 175513, iterations = 4, threads = 4 (took 274 ms)
# Benchmark returns argon2id() 4 iterations, 1048576 memory, 4 threads
argon2id 4 iterations, 1048576 memory, 4 parallel threads (CPUs)
```

Reading the produced output, we can immediately notice some important details about it. The first thing is that the benchmark function is based on an iterative algorithm. As it can be noticed in both Argon2i and Argon2id executions, the algorithm fixes 2 of 3 parameters (the iteration number and the thread number) and then, at each iteration, it continuously increments the memory cost parameter. This process runs until the execution does not take more then two seconds. The last parameters that do not exceed this constraint are selected as the values found by the benchmark function. Since the benchmark cannot run for a long, the increment of memory between each step is not fixed but is approximated.

The second idea on which the the benchmark function is based on is portability: the developer of "cryptsetup" decided that Argon2 limits should be set so that the created LUKS2 device would be as portable as possible.

Indeed, it should be possible to open it on different systems with different hardware, having of course a differ unlocking timer. In practice, the goal searched is to, for example, drive formatted on Xeon with 32GB memory and then unlocking it on Raspberry-Pi.

As also noticed by the output of the execution of the benchmark, the researched parameters are the memory cost (quantity of memory used during the execution of Argon2), the number of iterations and the number of threads to be used.

To continue our analysis, it is important to define for each of these parameters, a space in which they can vary. For the memory cost the upper bound is set to 4GB. This value is a limit that results from the maximum amount of per-process memory that can be allocated in any 32-bit systems.

It is important to notice that this value could be much higher, for example, in a 64bit operating system where the virtual memory that each process has at disposal is 2^{64}. Since our main goal is portability, we have to limit it to the most restrictive value.

Moreover, "cryptsetup" has a dynamic memory limit that decreases maximal amount of memory, according to available physical memory. In concrete, using the "adjusted_phys_memory()" function, the limit is lowered to the half of the physical memory. This function is very important because we have to keep in mind that on a 32bit system we could have less physical memory than 4GB, for example on a Raspberry-Pi, which has a 32bit architecture and only 1GB of RAM.

Without this limit, the "cryptsetup" process would have the capability to use all the 4GB of its virtual memory without having, in concrete, the physical ability to do that. For this reason, the "adjusted_phys_memory()" function is a de-facto prevention to out-of-memory (OOM) state, which can otherwise occur if the process requests more memory than the available amount and the operating system cannot satisfy it. If this condition happens and the OOM kills the process, it is not possible to guarantee that keys and passphrases are wiped from memory and that the execution of the encryption algorithm correctly ends.

Furthermore, with so huge amount of memory, current Argon2 implementation becomes very slow and the benchmark function will not find usable costs for required unlocking time.

The same reasoning could be done for the thread parameter: common portable devices have usually maximum quad-core CPU and thus it is reasonable to limit it to 4. As for the memory cost, the benchmark function will not use more threads than available physical cores so that the encrypted device can be unlocked on different systems.

The iteration count parameter is set by the benchmark function to a default value of 4. This is done because, at this moment, there are some attacks on Argon2, with an interaction count lower than 3. According to another paper, it should be at least 10, but in reality this value is not feasible because the unlocking time will be very high.

All these ideas are then concretized into the code of " crypt_argon2_check()", the core function involved into the execution of the benchmark process. In particular, this portion of code is responsible to search and produce all the Argon2 parameters. In order to better understand the attack that will be described into the Section 4.3, it is important to firstly analyze the internal of this function. The corresponding source code can be found here at the end of this section.

The path to this function is quite long; for this reason, only the stack-trace of the routine called before the execution of " crypt_argon2_check()" is reported below.

1. main()
2. run_action()
3. action->handler()
4. action_benchmark()
5. action_benchmark_kdf(CRYPT_KDF_ARGON2I, NULL, 256)

6. action_benchmark_kdf(CRYPT_KDF_ARGON2ID, NULL, 256)
7. crypt_benchmark_pbkdf()
8. crypt_pbkdf_perf()
9. crypt_argon2_check()

The parameters and the corresponding values on which the function works are reported in the following list:

1. const char *kdf: "argon2"
2. const char *password: "foo"
3. size_t password_length: 3
4. const char *salt: "0123456789abcdef0123456789abcdef"
5. size_t salt_length: 32
6. size_t key_length: 256 bit
7. uint32_t min_t_cost: pbkdf_limits.min_iterations: 4
8. uint32_t max_m_cost: max_memory_kb: DEFAULT_LUKS2_MEMORY_KB: 1048576 KByte
9. uint32_t parallel: parallel_threads: DEFAULT_LUKS2_PARALLEL_THREADS: 4
10. uint32_t target_ms: time_ms: DEFAULT_LUKS2_ITER_TIME: 2000 ms
11. uint32_t *out_t_cost: iterations_out: OUT
12. uint32_t *out_m_cost: memory_out: OUT

Most of them are self-explanatory. Those which are important to notice because of the narrow relation with the benchmark parameters are "min_t_cost" and "max_m_cost" which set respectively the minimum number of the iterations to execute during the Argon2 algorithm and the maximum number of Kbyte of memory to use. The "parallel" parameters indicate instead the number of threads to use. The "target_ms" value sets the maximum time that the benchmark execution can take. The last two parameters are pointers to the "uint32_t" variables which constitute the output of the benchmark function, respectively the "iterations" and the "memory".

Looking inside the code that realizes the "crypt_argon2_check" function, we can resume the fundamental operations executed and underline the relation with the output produced before. In particular, the first step is to set the basic value of the output parameters:

1. t_cost = min_t_cost = 4
2. m_cost = min_m_cost = 8 * parallel = 8 * 4 = 32kbyte

This condition is reflected into the first line printed during the execution:

```
# PBKDF benchmark: memory cost = 32, iterations = 4, threads = 4
```

After that, executing the lines that go from 31 to 73, the function finds some small parameters, so that the execution time is greater or equal to BENCH_MIN_MS constant, which is set to 250 ms. The output related to the execution of these lines is the following:

```
# PBKDF : memory cost = 512, iterations = 4, threads = 4 (took 7 ms)
# PBKDF : memory cost = 8192, iterations = 4, threads = 4 (took 22 ms)
# PBKDF : memory cost = 93090, iterations = 4, threads = 4 (took 170 ms)
# PBKDF : memory cost = 136897, iterations = 4, threads = 4 (took 232 ms)
# PBKDF : memory cost = 147518, iterations = 4, threads = 4 (took 235 ms)
# PBKDF : memory cost = 156934, iterations = 4, threads = 4 (took 234 ms)
# PBKDF : memory cost = 167664, iterations = 4, threads = 4 (took 274 ms)
```

As it can be noticed, the last line represents the first execution of Argon2 which exceeds 250 ms. Using these parameters (in this case memory cost = 167664, iterations = 4 and threads = 4) the benchmark function tries to estimate the target parameters using the "next_argon2_params" function.

At the end, the produced values of the function are located in the "out_t_cost" and "out_m_cost" pointers.

```c
static int crypt_argon2_check(const char *kdf,
            const char *password,
            size_t password_length,
            const char *salt,
            size_t salt_length,
            size_t key_length,
            uint32_t min_t_cost,
            uint32_t max_m_cost,
            Uint32_t parallel,
            uint32_t target_ms,
            uint32_t *out_t_cost,
            uint32_t *out_m_cost,
            void *usrptr)
{
  int r = 0;
  char *key = NULL;
  uint32_t t_cost, m_cost, min_m_cost = 8 * parallel;
  long ms;
  long ms_least = (long)target_ms * BENCH_PERCENT_ATLEAST / 100;
  long ms_atmost = (long)target_ms * BENCH_PERCENT_ATMOST / 100;

  if (key_length <= 0 || target_ms <= 0) return -EINVAL;
  if (max_m_cost < min_m_cost) return -EINVAL;

  key = malloc(key_length);

  if (!key) return -ENOMEM;
  t_cost = min_t_cost;
  m_cost = min_m_cost;

  /* 1. Find some small parameters,
   * s. t. ms >= BENCH_MIN_MS: */

  while (1) {
    r = measure_argon2(kdf,
          password,
          password_length,
          salt, salt_length, key,
          key_length, t_cost, m_cost,
          parallel, BENCH_SAMPLES_FAST,
          BENCH_MIN_MS, &ms);
    if (!r) {
      // Update parameters to actual measurement
      *out_t_cost = t_cost;
      *out_m_cost = m_cost;
      if (progress && progress((uint32_t)ms, usrptr))
        r = -EINTR;
    }

  if (r < 0)
    goto out;

  if (ms >= BENCH_MIN_MS)
    break;

  if (m_cost == max_m_cost) {
    if (ms < BENCH_MIN_MS_FAST)
      t_cost *= 16;
    else {
      uint32_t new = (t_cost * BENCH_MIN_MS) / (uint32_t)ms;

      if (new == t_cost)
        break;

      t_cost = new;
    }
  } else {
    if (ms < BENCH_MIN_MS_FAST)
      m_cost *= 16;
    else {
      uint32_t new = (m_cost * BENCH_MIN_MS) / (uint32_t)ms;
      if (new == m_cost)
        break;

      m_cost = new;
    }
    if (m_cost > max_m_cost) {
      m_cost = max_m_cost;
    }
  }
}
/*
 * 2. Use the params obtained in (1.)
       * to estimate the target params.
 * 3. Then repeatedly measure the candidate
       * params and if they fall out of the acceptance
       * range (+-5 %), try to improve the estimate:
 */
do {
  if (next_argon2_params(&t_cost, &m_cost,
             min_t_cost, min_m_cost,
             max_m_cost, ms, target_ms))
  {
    // Update parameters to final computation
    *out_t_cost = t_cost;
    *out_m_cost = m_cost;
```

```
      break;
    }

    r = measure_argon2(kdf, password,
           password_length,
           salt, salt_length, key,
           key_length, t_cost, m_cost,
           parallel, BENCH_SAMPLES_SLOW,
           ms_atleast, &ms);

    if (!r) {
      // Update parameters to actual measurement
      *out_t_cost = t_cost;
      *out_m_cost = m_cost;
      if (progress && progress((uint32_t)ms,
             usrptr))
        r = -EINTR;
    }

    if (r < 0)
      break;
  } while (ms < ms_atleast || ms > ms_atmost);
out:
  if (key) {
    crypt_backend_memzero(key, key_length);
    free(key);
  }
  return r;
}
```

4.3 Attacking the CPU

After studying and analyzing the involved algorithms, we start describing the attack.

We will use all the information gathered in order to create one or more corner cases. As already said, the goal is not to find a theoretical attack on the Argon2 encryption but, instead, it is to execute an instance of the encryption algorithm when the system is under a stress condition and then try to induce the benchmark function to produce as output of its computation lower values then what they would be in a normal condition.

Since the focal point on which Argon2 is based on is the memory, the first idea that someone might expect is that the crucial resource that an attacker has to compromise, in order to manipulate the benchmark process, has to be the memory.

After making several tests, this hypothesis was dropped. The first problem that comes up if you try to follow this path is a stability problem. In order to force the operating system into a memory stress condition, it is necessary that one or more processes fill up the memory with random data. This, at least theoretically, should be enough to force the Argon2 benchmark function to use less memory and then produce a lower memory output value for the benchmark. The problem that occurs is that it is very difficult to maintain this condition without having the process killed by the OOM manager of the operating system.

Furthermore, even if one is able to create and maintain a low memory condition into the operating system, it would not be enough to lower the values produced by the benchmark function.

As the executed tests suggest, the results obtained into a memory stress condition are as much as, if not the same, the values obtained into a quiet condition. This effect has to be awarded to the code into the "next_argon2_params" function: attacking the memory instead of the CPU, the resulting effect is that the space of memory on which the benchmark works is less while the CPU is not affected by the malicious process.

This condition implies that the computational time, given by the ratio between the memory used and the computational capacity of the CPU, is not affected by the attack. The "next_argon2_params" function, which has the role to estimate the final values based on those obtained in the previous iterations, takes the computational time found in a memory stress condition and adapts it, making a linear proportion, to what it would be if the memory would be more. The effect is that the resulting values found by the benchmark function in a memory stress condition are pretty the same the ones found in a quite state.

So the second idea was to follow the CPU path. This approach consists on creating a single process that is able to take up the CPU resource and lower the values of the benchmark function.

Going deeper into the source code, it is possible to notice that the "crypt_argon2_check" function uses "measure_argon2" in order to calculate the execution time of Argon2 encryption. Here is the part where the weakness resides. Indeed, analyzing the source code, we can notice that at line 16 the following comment: "We must use clock_gettime here, because Argon2 can run over multiple threads, and thus we care about real time, not CPU time!"

This is the hint on how the time is computed by the benchmark function: because of a user-friendly performance choice, the developers of Cryptsetup decided to take into account the difference between the starting time and the ending time of the whole process. At first, this might seem to be a correct solution but, instead, this choice leads to a valid corner case.

Using the "clock_gettime" function with the "CLOCK_MONOTONIC_RAW" parameter, the time retrieved is monotonic since some unspecified starting point. This clock is not affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock). Furthermore, this mode provides access to a raw hardware-based time that is not subject to NTP adjustments or the incremental adjustments performed by adjtime [6].

In concrete, this means that if two threads of the same process are executed in a time slot that is calculated with the previous method, it is not sure that this time is equal to the the time that each thread really spends into the the CPU.

To give a better idea of this phenomenon, we should step back on how the scheduler of the operating system works when it has to handle multiple processes [12].

The Argon2 instance is obviously not the only process in the environment. For this reason, the operating system has to somehow distribute equally the CPU resource among all of them, so that they could be executed. The Figure 5 gives a better idea of this condition.

Suppose to have a single core CPU and that, at a given instant of time, the system is encountering a CPU stress condition when the benchmark function is running. For simplicity, fix the number of threads for the benchmark function to one. It is quite clear that the greater is the number of processes that has to be executed, the lower is the number of time-slots that will be assigned to the Argon2 thread within the two second slots fixed by the benchmark function.

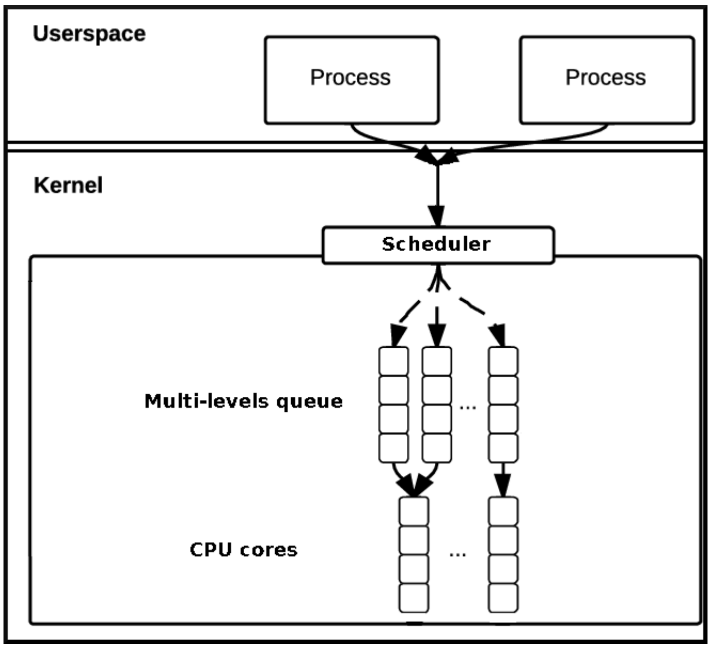

Figure 5: Multi-levels queue scheduler

The immediate consequence of this phenomenon is that the actual time in which the function should be executed is less and, since Cryptsetup is agnostic about the other processes running in the system, the benchmark will believe to be executed within a less performing system.

The side effect of this condition is that the values found would be lower than those found in a quite state.

Due to what we have found in the previous analysis, the idea of the attack is to execute a process that has the ability to keep busy the CPU (for example spawning a lot of threads) and in the same time execute the benchmark function. The goal is to see how the benchmark function is affected by the load of the CPU produced by the malicious process.

Before proceeding, it is important to say that all the tests have been executed with the "cryptsetup 2.1.0-git" version on a virtual machine with the following specifications:

1. CPU: AMB Ryzen 7 1700 (8-core processor)
2. Core: from 1 up to 8
3. Memory: 4GB
4. OS: Ubuntu 18.04.02 LTS

Since our goal is focused on the CPU, the tests are executed on different virtual machine environments on which the memory is fixed to 4GB of RAM and the number of cores of the CPU is between 1 to 8. Below is reported the integral source code of the "measure_argon2" function.

```c
static int
measure_argon2(const char *kdf, const char *password,
          size_t password_length, const char *salt,
          size_t salt_length, char *key,
          size_t key_length, uint32_t t_cost, uint32_t m_cost,
          uint32_t parallel, size_t samples, long ms_atleast,
          long *out_ms )
{
  long ms, ms_min = LONG_MAX;
  int r;
  size_t i;

  for (i = 0; i < samples; i++) {
    struct timespec tstart, tend;

    /* NOTE: We must use clock_gettime here,
     * because Argon2 can run over multiple threads,
     * and thus we care about real time, not CPU time!
     */

    if (clock_gettime(CLOCK_MONOTONIC_RAW, &tstart) < 0)
      return -EINVAL;


    r = crypt_pbkdf(kdf, NULL, password, password_length,
        salt, salt_length, key, key_length,
        t_cost, m_cost, parallel);


    if (r < 0)
      return r;

    if (clock_gettime(CLOCK_MONOTONIC_RAW, &tend) < 0)
      return -EINVAL;

    ms = timespec_ms(&tstart, &tend);

    if (ms < 0)
      return -EINVAL;

    if (ms < ms_atleast) {
      /* early exit */
      ms_min = ms;
      break;
    }

    if (ms < ms_min) {
      ms_min = ms;
    }
  }
  *out_ms = ms_min;
  return 0;
}
```

4.4 Results

The results of the CPU-based attack are reported into the following Table 1. As already said, the testing phase was performed upon a virtual environment which provided the ability to set the number of "physical" cores. This information is reported into the "Core" column.

For each "core" configuration several tests were executed. Each of them consists in the creation of a LUKS2 device using the Cryptsetup tool with Argon2 as KDF function. No fixed parameters were provided except for the "–pbkdf-parallels" that is specified into the "Parallels" column. This parameter has a range between 1 and the number of physical cores, so it has been performed a test for each of these values.

The other columns are self-explanatory, like "Processes", "Thread" and "RAM". Below each of these columns there are two sub-columns, representing the environment condition in which the encryption took place. The "Q" stands for "quiet" and "S" for "stress". The last column represents the "Memory" value found by the benchmark function and then used by Argon2.

As it can be noticed by looking at the last column, the results obtained using the CPU based attack are very close to what we expected from the static analysis of the source code of the "measure_argon2" function.

In particular, if we compare the quiet and the stress values of "Memory" using the Table 2, we can observe a decrease of the memory by a factor between 20-50%. This condition is even more evident into the Figure 6. As expected, the best results were obtained on an environment with a number of cores between 1 and 4 included. It is important to underline that these core configurations are the most common ones on the CPU market.

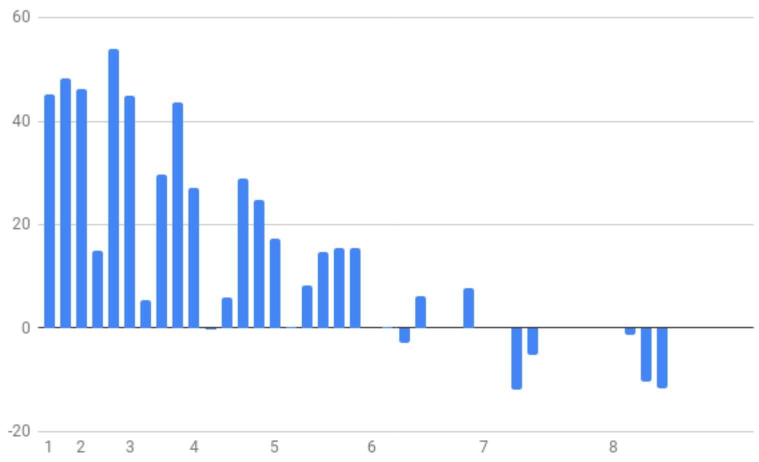

Figure 6: Memory drop

The motivation of this phenomenon is that with a greater number of core, the operating system is able to handle more efficiently the processes and distribute better the load generated over its processing units. On a CPU that has more then 4 cores, the result is that the *Cryptsetup* process has more time-slot of execution during the 2 second timeout and so the benchmark function produce better values.

In concrete, the greater the number of cores, the lower is the factor of impact of the CPU attack. As the Table 2 suggests, with more than 4 cores, the effects of this attack become less evident until they disappear.

Core	Parallels	Processes		Threads		RAM		Memory	
		Q	S	Q	S	Q	S	Q	S
1	default	221	221	504	1528	997/3945	1032/3945	458428	251236
	1	220	221	503	1528	997/3945	1032/3945	489360	252837
2	default	230	234	515	1553	874/3945	1024/3945	752961	404715
	1	230	234	513	1553	874/3945	1024/3945	492539	418803
	2	230	234	513	1553	875/3945	1026/3945	856849	394825
3	default	241	244	535	1570	876/3945	1027/3945	1048576	576322
	1	240	244	533	1570	876/3945	1028/3945	479482	453227
	2	240	244	532	1570	876/3945	1027/3945	771671	542157
	3	240	244	532	1570	876/3945	1027/3945	1048576	591241
4	default	249	253	552	1588	883/3945	1034/3945	1048576	765423
	1	249	253	549	1588	883/3945	1036/3945	481370	483082
	2	249	253	549	1587	884/3945	1036/3945	719577	676301
	3	249	253	549	1587	884/3945	1036/3945	1048576	746764
	4	252	253	570	1587	996/3945	1036/3945	1048576	790000
5	default	258	253	565	1596	881/3945	1016/3945	1048576	867498
	1	258	253	565	1596	882/3945	1017/3945	487123	485448
	2	260	253	577	1596	896/3945	1017/3945	835826	767578
	3	261	253	586	1596	980/3945	1017/3945	1040173	887812
	4	261	253	580	1596	980/3945	1018/3945	1048576	887494
	5	261	253	580	1596	980/3945	1018/3945	1048576	886118
6	default	269	272	583	1622	886/3945	1020/3945	1048576	1048576
	1	268	272	583	1622	885/3945	1021/3945	489346	488546
	2	268	268	583	1618	893/3945	1021/3945	716738	737572
	3	271	268	604	1618	995/3945	1022/3945	1048576	983618
	4	271	268	599	1617	985/3945	1022/3945	1048576	1048576
	5	271	264	597	1614	985/3945	1021/3945	1048576	1048576
	6	271	264	597	1614	984/3945	1021/3945	1048576	967374
7	default	278	280	620	1638	1005/3945	1050/3945	1048576	1048576
	1	278	280	616	1638	1012/3945	1053/3945	492393	492512
	2	278	280	612	1638	1013/3945	1053/3945	715017	799980
	3	278	274	612	1632	1013/3945	1052/3945	996053	1048576
	4	278	274	612	1632	1013/3945	1052/3945	1048576	1048576
	5	278	274	613	1632	1014/3945	1053/3945	1048576	1048576
	6	278	274	612	1632	1013/3945	1054/3945	1048576	1048576
	7	278	269	612	1627	1013/3945	1052/3945	1048576	1048576
8	default	284	282	615	1648	890/3945	1031/3945	1048576	1048576
	1	284	277	615	1644	891/3945	1033/3945	483082	489072
	2	287	277	636	1644	991/3945	1032/3945	695767	767743
	3	287	277	630	1644	991/3945	1032/3945	939952	1048576
	4	287	277	630	1644	991/3945	1032/3945	1048576	1048576
	5	287	277	630	1644	990/3945	1032/3945	1048576	1048576
	6	287	277	631	1644	991/3945	1032/3945	1048576	1048576
	7	287	277	630	1644	989/3945	1032/3945	1048576	1048576
	8	287	277	630	1644	989/3945	1033/3945	1048576	1048576

Table 1: Results

Core	Quiet	Stress	S/Q * 100	Memory drop (%)	Memory drop (Mbyte)
1	458428	251236	54.8038078	45.1961922	207192
	489360	252837	51.66687102	48.33312898	236523
2	752961	404715	53.74979581	46.25020419	348246
	492539	418803	85.02940884	14.97059116	73736
	856849	394825	46.07871399	53.92128601	462024
3	1048576	576322	54.96234894	45.03765106	472254
	479482	453227	94.52429914	5.475700861	26255
	771671	542157	70.25753203	29.74246797	229514
	1048576	591241	56.38513565	43.61486435	457335
4	1048576	765423	72.99642563	27.00357437	283153
	481370	483082	100.3556516	-0.3556515778	-1712
	719577	676301	93.98591117	6.014088833	43276
	1048576	746764	71.21696472	28.78303528	301812
	1048576	790000	75.340271	24.659729	258576
5	1048576	867498	82.73105621	17.26894379	181078
	487123	485448	99.65614434	0.3438556586	1675
	835826	767578	91.83466415	8.165335847	68248
	1040173	887812	85.35234043	14.64765957	152361
	1048576	887494	84.63802338	15.36197662	161082
	1048576	886118	84.50679779	15.49320221	162458
6	1048576	1048576	100	0	0
	489346	488546	99.83651649	0.1634835066	800
	716738	737572	102.9067804	-2.906780441	-20834
	1048576	983618	93.80512238	6.194877625	64958
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	967374	92.25597382	7.744026184	81202
7	1048576	1048576	100	0	0
	492393	492512	100.0241677	-0.02416768719	-119
	715017	799980	111.8826545	-11.88265454	-84963
	996053	1048576	105.273113	-5.273112977	-52523
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
8	1048576	1048576	100	0	0
	483082	489072	101.2399551	-1.239955121	-5990
	695767	767743	110.3448425	-10.34484245	-71976
	939952	1048576	111.5563348	-11.55633479	-108624
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0
	1048576	1048576	100	0	0

Table 2: Memory drop results

Chapter 5

Considerations and proposed solution

The disk encryption process is the main defense used in order to protect sensitive information stored into a physical device.

In the open-source world there are a lot of alternatives that solve this problem but the most widespread is *dm-crypt*, used with its management tool *cryptsetup*, in conjunction with the LUKS format.

The two-layer mechanism, used by LUKS to store the *master-key*, uses a *KDF* in order to protect it from brute-force attacks.

In order to overcome the weaknesses introduced by the advent of the GPU into the old KDF schemes like *PBKDF2*, the NIST institute organized in 2013 a "Password Hashing Competition" with the goal to select a new hash scheme which would be secure against cracking-optimized systems. Among all the proposals, *Argon2* has been selected as the winner of the competition.

As for all the other cryptographic algorithms, the developers of *cryptsetup* integrated into the source code the benchmark function of this new *KDF*. This function has the goal to find sub-optimal values for the initial parameters of the *Argon2* algorithm.

The research presented in this Thesis had the goal to test and find possible security issues related to the actual implementation of the benchmark function of *Argon2*.

In concrete, we focused our attention on finding particular states of the operating system, defined as *corner-cases*, which might lead the benchmark function to find lower values than what it might do in a stable condition.

After a careful static analysis of the source code of the functions involved (4.2), we produced a CPU-based attack against the benchmark function (4.3).

Thanks to the mode used to retrieve the system-time (calling the *clock_gettime* function with *CLOCK_MONOTONIC_RAW* argument), we discovered that a malicious process that keeps the CPU busy, for example spawning a lot of threads, has the ability to influence the values found by the benchmark function.

As shown in Section 4.4, this attack allowed us to lower the value of the memory by a factor between 20-50% on an environment with a number of cores that ranges from 1 to 4. This phenomenon is due to the fact that on a system with more than 4 cores, the scheduler of the operating system is able to handle the processes more efficiently and distribute the load generated over its processing units in a better way.

For this reason, we propose a statistical based solution. In particular, the idea behind it is to run the benchmark function of *Argon2* on multiple isolated virtual environments, each of them with a different physical setup. After collecting a big amount of benchmark results, it would be necessary to select which features of the system would more affect the results of the benchmark function. By doing this, it would be possible to correlate the output values of the benchmark with the features of the environment.

Applying a statistical method, it would be then possible to create a mathematical function that, given the actual input features, would return the predicted output values for the *Argon2* algorithm.

The limit of this approach is that the considered statistical function has to trust the input features given by the operating system. An attacker with a local access and root privileges would be able to intercept the system-call done by the function in order to retrieve the physical parameters and give back altered ones. This scenario is however out of the scope of this solution because an attacker with that privilege levels would be able to break all the security mechanisms and execute arbitrary actions on the system.

Bibliography

[1] Joël Alwen and Jeremiah Blocki. "Efficiently computing data-independent memory-hard functions". In: *Annual International Cryptology Conference*. Springer. 2016, pp. 241–271.

[2] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 292–302.

[3] Alex Biryukov and Dmitry Khovratovich. "Tradeoff cryptanalysis of memory-hard functions". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2015, pp. 633–657.

[4] Simone Bossi and Andrea Visconti. "What users should know about full disk encryption based on LUKS". In: *International Conference on Cryptology and Network Security*. Springer. 2015, pp. 225–237.

[5] Milan Broz. *Cryptsetup*. URL: `https://gitlab.com/cryptsetup/cryptsetup`.

[6] *C function clock_gettime*. URL: `https://linux.die.net/man/3/clock_gettime`.

[7] Eoghan Casey and Gerasimos J Stellatos. "The impact of full disk encryption on digital forensics". In: *ACM SIGOPS Operating Systems Review* 42.3 (2008), pp. 93–98.

[8] Clemens Fruhwirth. *LUKS On-Disk Format Specification Version 1.1*. 2008.

[9] Clemens Fruhwirth. *LUKS On-Disk Format Specification Version 1.2*. 2011.

[10] Laszlo Hars. "Discryption: Internal hard-disk encryption for secure storage". In: *Computer* 40.6 (2007), pp. 103–105.

[11] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. `http://www.rfc-editor.org/rfc/rfc2104.txt`. RFC Editor, 1997. URL: `http://www.rfc-editor.org/rfc/rfc2104.txt`.

[12] *Multi-Queue Scheduler for Linux.* URL: `http://lse.sourceforge.net/scheduling/mq1.html`.

[13] *Out of Memory Management.* URL: `https://www.kernel.org/doc/gorman/html/understand/understand016.html`.

[14] Colin Percival. *Stronger key derivation via sequential memory-hard functions.* 2009.

[15] Christophe Saout. "dm-crypt: a device-mapper crypto target, 2007". In: *URL http://www. saout. de/misc/dm-crypt* (2014).

[16] Marc Stevens et al. "The first collision for full SHA-1". In: *Annual International Cryptology Conference*. Springer. 2017, pp. 570–596.

[17] Andrea Visconti and Federico Gorla. "Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2". In: *IEEE Transactions on Dependable and Secure Computing* (2018).

[18] Andrea Visconti et al. "Examining PBKDF2 security margin—Case study of LUKS". In: *Journal of Information Security and Applications* 46 (2019), pp. 296–306.

[19] Andrea Visconti et al. "On the weaknesses of PBKDF2". In: *International Conference on Cryptology and Network Security*. Springer. 2015, pp. 119–126.

[20] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding collisions in the full SHA-1". In: *Annual international cryptology conference*. Springer. 2005, pp. 17–36.