Masaryk University
Faculty of Informatics

Argon2 security margin for disk encryption passwords

Master's Thesis

Bc. Vojtěch Polášek

Brno, Spring 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Vojtěch Polášek

Advisor: Ing. Milan Brož

Abstract

«abstract»

Keywords

«keywords»

Contents

List of Tables

Contents

1 Introduction

2 Password hashing and key derivation functions

2.1 Definitions

This thesis deals with various cryptographic terms including password hashing and key derivation. This section briefly explains some of them. Note that reader of this thesis is expected to have at least basic knowlege of cryptography and computer security.

A *hashing function* is a function which receives an input of arbitrary length and produces an output of specified shorter length, effectively compressing the input. These functions are used in many areas such as effective data retrieval [22]. *Cryptographic hashing functions* are subset of *hashing functions* and they have to meet certain properties, namely preimage resistance, second preimage resistance and collision resistance. We will consider only cryptographic hash functions in this thesis.

Password hashing is a process in which a password is supplied to a hash function. This is de facto standard method of storing of saved passwords in operating systems and applications. In case that an attacker gets hold of such password hashes, it should be ractically infeasible for an attacker to derive he original password. Therefore hashing functions are used also in process of password verification, during which the entered password is hashed and compared with the stored hash.

This thesis is not going to deal with *password hashing*. However, many *key derivation functions* described below meet desired properties for being used in the password hashing process. However, this thesis is focused primarily on key derivation and verification of cryptographic keys during disk encryption.

Key derivation functions are based on *hash functions*. Their basic purpose is to take an input and produce an output which can be used as a cryptographic key. The input is usually a password or other material such as biometric sample converted into binary form. These materials could be of course used as cryptographic keys on their own

but they often lack properties of a good cryptographic key such as sufficient entropy or length.

A *cryptographic salt* is often used during process of key derivation. The purpose of salt is to prevent attacks which use precomputed tables such as Rainbow tables [30]. Salt introduces another factor which influences a derived key. It means that it is no longer dependent only on passhprase. For example suppose that 32 bit long integer is used as a salt. In that case there are 2^{32} possible keys derived from the same passphrase. This makes precomputing attacks effectively infeasible. Salt is usualy stored unobfuscated together with hashed material.

2.2 Why do we need PBKDFs?

Today, as more and more private information is stored on various kinds of media and transfered over the Internet, it is becoming crucial to protect it from being accessed or changed by unauthorised actors. Although there are several interesting authentication options such as biometrics, passwords or passphrases are still the most common method.

Considering passwords we are facing a problematic situation. Organisations and services provide guidelines or requirements which should help an user to choose a strong password [17] [37]. Important parameters are password length (in characters), password complexity, uniqueness and others. By complexity I mean amount and diversity of used characters (letters, numbers, symbols, emojis...) and by uniqueness I mean the fact that the password does not contain easily guessable or predictable sequences. See mentioned policies for example.

I will find some researches supporting following lines

As shown by researches, users tend to circumvent such policies by finding loopholes in them.

What more, passwords them selves are not good cryptographic material which should be used as a cryptographic key. There are surprisingly many reasons. They are usually not sufficiently long. Because they are composed of printable characters, they do not meet the requirement of being uniformly distributed. If they should be

remembered, they will probably contain dictionary words, which lessens their entropy even more. See [22, section 5.6.4] for short but interesting analysis.

PBKDF stands for password-based key derivation function. The goal of these functions is to derive one or more cryptographic keys from a password or a passphrase. This key should be pseudorandom and sufficiently long to make brute-force guessing as time-consuming as possible. As stated above, they are based on cryptographic hashing functions.

Lately PBKDFs are taking another specific task. Due to availability of GPUs, FPGAs and ASICs, there are new possibilities in running functions in parallel computing environment [see 28, chapter 4]. This increases effectivity of brute-force attacks. PBKDFs try to defend against such attacks by using salt and and function-specific parameters like iteration count etc. See section 2.4 for more details.

Examples of PBKDFs include Argon2, PBKDF2, Scrypt, Yscrypt and more. See chapter 3 for comparison of several functions.

2.3 PBKDFs and disk encryption

Disk encryption is a very good use case for usage of PBKDFs. Used encryption algorithms require cryptographic keys of certain length [19]. It is also important to consider the fact, that it is usually not desirable to change the encryption key often because reencryption of whole disk takes considerable amount of time. Let aside the fact that if an attacker gains permanent access to such an encrypted disk, the key cannot be changed at all and they may have extensive time period during which they can manage to crack the key.

By looking at [41] we can see that PBKDFs are used in many types of disk encryption software. Note that this list mentions only PBKDF2 as this has been most used PBKDF since recent times. PBKDF2 is for example used in LUKS version 1 [14], FileVault software used by macOS [1], CipherShed disk encryption software [9], Veracrypt disk encryption software [19] and more.

In 2013 there was initiated a new open competition called Password hashing competition. Its goal was to find a new password hashing function which would resist new attacks devised against those func-

tions [2]. The winner was function named Argon2. It is already used for example in LUKS version 2 [8].

2.4 Attacks on PBKDFs

History of password cracking is as old as computer passwords them selves. People crack passwords for two main reasons. Either they want to recover a forgotten password, or they want to recover password of someone else, later being able to use it for authentication purposes. There exist two main techniques for password cracking; brute force attacks and dictionary attacks. The first form of attacks tries to guess the password by trying all passwords of given length composed of all combinations of given characters. Dictionary attacks exploit the fact that people often use passwords containing words found in language dictionaries. It means that if an attacker tries paswords containing dictionary words or permutations, they have quite good chance of success.

At the begining passwords were stored on computer systems in plain text, protected only by the fact that users shouldn't be able to read passwords of other users. But soon it appeared that plain text passwords can be revealed for example by badly designed software permissions. This is shown in case of Allan Scherr, who misused capabilities of a printing program to print out whole password file [10]page 37. Since that time, passwords started to be hashed.

. This time marks begining of the never-ending fight between authors of hash functions and people trying to crack passwords hashed by them. First hash functions were really simple and they were definitely not cryptographically secure, such as hash mechanism used in Multics. This mechanism squared numerical form of each password and applied a bit mask with AND operation [39]. This increased number of guesses but only negligibly compared to modern functions.

The first cryptographic hash came with Robert Morris and his Crypt function. Crypt used up to Unix 6th edition mimicked the M209 cipher machine from World War II. and proved not very secure because the algorithm could be recoded in a way which allowed to test passwords in very short period of time (1.25 milliseconds per password) [33]. Later version used since Unix 7th edition employed the

DES block cipher. This cipher was at that time very slow if implemented in software. The password entry program also introduced two new concepts; automated proactive password strength checking and *cryptographic salt* (12 bit random number at that time). Currently for example the LUKS2 specification uses 32 bytes long salt.

During 1980s there happened some password cracking contests and hash functions were also improved. Some of them were made deliberately slow to slow down potential attacker. If this measure was not effective enough, more iterations of hashing function could be used. During 1990s many password cracking programs appeared including John the Ripper, Crack, LOphtCrack etc. [23]. Most of those programs tried to improve the cracking speed by optimizing underlying algorithms and later by using CPU parallelism.

The concept of *KDFs* started being studied in late 1990s. The RFC 2898 for PBKDF2 was released in 2000 and it started being used primarily for key derivation in many applications such as WinZip, Opendocument, Truecrypt or Android. However, it was also used for actual password hashing for example in Mac OS X 10.8. Note that PBKDF1 exists, but it is not recommended because of its limited key length (20 bytes at best) and it is provded solely for backward compatibility. PBKDF2 introduced configurable pseudorandom function, number of iterations and derived key length. These parameters allow flexibility while choosing trade-off between security and user experience.

However, in 2007 there appeared first password crackers using parrallel computing capabitilies of GPUs [4]. Other password crackers followed; Whitepixel in 2010, Oclhashcat in 2012, John the ripper in 2012. Until 2012 software could recover primarily MD5 and NTLM hashes, but Oclhashcat introduced recovery of many other hashes.

As far as PBKDF2 is considered, its resistance to password cracking using GPUs or even ASICs/FPGAs is currently not ideal [28]section 7. PBKDF2 does not offer support for parallelism while used as suggested. However, at the same time its low memory requirements and GPU friendly algorithm (see algorithm 1 bring advantage to the attacker. As shown in [28], it was possible to improve cracking speed of LUKS passwords fourty times. This required rewriting the function for GPUs. Moreover, it is shown that proper optimization of underlying algorithms can greatly increase PBKDF2 performance even without

GPUs. This was shown after analysis of closed source Oclhashcat in [36].

As shown in [40], there exist other attacks not connected with GPU. This paper shows that an attacker can save 50 % of PBKDF2 cpu operations if the PBKDF2 is not implemented according to suggested performance improvements described in RFC 2104 [24] and NIST FIPS PUB 198 [20]. In this case it is possible to precompute first message block of underlying keyed hash function (used as PRF) and replace it with resulting constant in subsequent operations. See lines 11–13 in algorithm 1. Note that HMAC function is not described in this thesis, see [40]subsection 4.1.

In [40]subsection 4.2 there is shown that an attacker can omit considerable amount of XOR operations while using SHA1 as a pseudo-random function within PBKDF2 because this operation is sometimes performed on two blocks containing only zeroes. Additionally, more XOR operations can be omitted because of padding characters which are constant and some XOR operations in this case just zero out them selves. Finally, in subsection 4.3, it is shown that it is possible to precompute the word expansion part of the second message block of a keyed hash function. The block is password independent and can be thus precomputed. However, this saves only negligible amount of time compared to previous attacks described in this paragraph.

In 2009 Colin Percival suggested that to defend against usage of parallel computing, PBKDFs should fulfill requirements of memory-hard functions [31]. As a reaction to previously mentioned problems of PBKDF2, the Password hashing competition was held from 2013 to 2015 to select a new function for password hashing. The winner is Argon2 described in section 3.2 and it is indeed a memory-hard function. Of course that does not mean that memory-hard functions are not prone to attacks.

In a paper from 2016, authors show that there still exists an attack which can decrease computational complexity of Argon2I-B function. It was shown that at that time it was needed to configure at least ten passes of Argon2I-B to mitigate this attack. At the time of releasing the paper, the IRTF proposal suggested only six passes for "paranoid" situations. Fortunately authors of Argon2 reacted to those attacks and improved the function, sothat the attack is not effective against

Argon2I anymore except when runnin with only signle pass. Details and rationale can be found in [5]subsection 5.2.

3 State of the art PBKDFs

3.1 PBKDF2

PBKDF2 is a password-based key derivation function defined in RFC 8018 [21]. This RFC thoroughly describes two use cases of PBKDF2; password-based encryption scheme and password-based message authentication scheme. Other mentioned use cases include password checking and derivation of multiple keys from one password. As shown in 4.1.1, LUKS version 1 uses PBKDF2 for password checking and derivation of key for encryption or decryption of master key.

the function requires four input parameters; passphrase, cryptographic salt, iteration count and length of a key to be derived. Moreover, the function requires a pseudorandom function (PRF) which is used in process of key derivation.

The term *Passphrase* in this context means any data which are source for subsequent key derivation process. Usually it is a password entered by user. The *cryptographic salt* is represented by randomly generated number.

The purpose of *iteration count* parameter is to defend against brute force and dictionary attacks performed on PBKDF2. The iteration count prolongs the time which is needed to derive a single key. Technically, the iteration count signifies number of successive runs of chosen PRF for every block of the derived key. In general, the *iteration count* should be chosen as large as possible, taking into account the fact that the processing time should be acceptable for the end user [38]. According to the cited document, minimum iteration count should be 1000 iterations and for critical security systems a count of 10000000 iterations is appropriate.

The function is described by following algorithm. Verbal description is also provided. Following abbreviations and conventions are used in the algorithm and description:

P an octet string representing a passphrase

S an octet string representing a cryptographic salt

C a positive integer representing iteration count

dkLen a positive integer representing length of the derived key counted in octets

DKan octet string representing the derived key

PRF - a pseudorandom function

hLen - length of output of chosen pseudorandom function counted in octets

CEIL(x) the ceiling function returning the smallest integer which is greater or equal to X

F a helper function for better description

|| concatenation of strings

INT(x) a big§-endian encoding of integer x

At the begining the algorithm checks if the desired length of the key does not exceed $2^{32} - 1$. If it does, it exits immediatelly. Then it processes the input and creates output key in blocks. Every block has length of hLen octects, except for the last one which can have shorter length.

In the pseudocode there is defined function *F* which is applied to every block. Results of such applications are finally concatenated and returned as the resulting derived key. This function performs *c* iterations of underlying pseudorandom function *PRF*. The *PRF* takes a passhprse as the first parameter and result of previous iteration as the second parameter. The only exception is the first iteration where the second parameter is concatenation of salt and binary representation of the block index *i*. Results of all iterations are xored and returned as a particular block of the derived key. Finally, all blocks are concatenated and returned as the derived key.

Notice that the function F can be rewritten to be quickly computed in parallel computing environments such as GPUs. See [28]section 4.1 for more details.

input : P, S, C, dkLen
output: DK
if $dkLen > (2^{32} - 1) \times hLen$ **then**
  return *Derived key too long*
end
$L \leftarrow$ `CEIL(`$dkLen/hLen$`)`
`/* l is the number of hLen-octet blocks in the derived key */`
$r \leftarrow dkLen - (l - 1) \times hLen$ `/* r is the number of octets in the last block */`
for $i \leftarrow 1$ **to** l **do**
  $T_i \leftarrow$ `F(`p, s, c, i`)`
end
return $t_1 || t_2 || \ldots t_l[0 \ldots (r - 1)]$
Function $F\ (s, p, c, i)$
  $u_1 \leftarrow$ `PRF(`$P, S \parallel INT(i)$`)`
  for $j \leftarrow 2$ **to** c **do**
    $u_j \leftarrow$ `PRF(`P, u_{j-1}`)`
  end
  return $u_1 \oplus u_2 \oplus \ldots \oplus u_c$

Algorithm 1: PBKDF2 function algorithm

3.2 Argon2

As mentioned in very brief history of attacks on password hashes in section 2.4, the Argon2 function is the winner of Password Hashing Competition. Argon2 is a hash function belonging to the set of memory-hard functions as defined in [31]. As defined in section 2.1 PBKDFs are subset of hashing functions and Argon2 can be definitely used as PBKDF. Current version of the function is 1.3 and the latest IETF draft is [6].

The function quickly fills up given amount of memory and performs a sequence of computations over values stored in this memory. The Argon2 comes in three versions which differ in the way in which data in the memory matrix (described further below) is processed. Argon2D, Argon2I and Argon2ID. See subsection 3.2.3 for detailed description.

Argon2 is used as the default PBKDF in LUKS version 2. Note that according to [13] the default PBKDF can be configured during compilation.

The function is optimized for X86 architecture using improvements in handling of cache and memory access in recent Intel and AMD processors. To be exact, currently the Argon2 supports compilation on platforms with SSE2, SSSE3, XOP, AVX2 and AVX512F cpu instructions. All the instructions except for XOP are Intel specific, XOP is specific for AMD processors.

The function can be implemented on specialised hardware, but the previously mentioned fact makes this implementation possibly very slow and expensive and even specialised ASICs shouldn't acquire significant benefit even if they employ large areas of memory. However, no implementations of any Argon2 mode for specialised hardware are known so far.

3.2.1 Operation

The function expects following primary input parameters. Primary means that parameters must always be supplied by user.

P the message of any length from 0 to $2^{32} - 1$ bytes. In case of PBKDF this is the passphrase.

S nonce with length from 8 to $2^{32} - 1$ bytes. In case of PBKDF this is the *cryptographic salt*.

The function also accepts secondary inputs, which do not need to be supplied.

p degree of parallelism as an integer with a value from 1 to $2^{24} - 1$. The value determines the number of parallel computational chains to be run. Chains are not independent, synchronisation occurs.

τ tag length in bytes in range from 4 to $2^{32} - 1$. This determines an output of the function, in case of PBKDF key length.

m memory size as an integer number in range from $8p$ to $2^{32} - 1$. The integer determines amount of kilobytes of memory which should be used for computation of the function.

t number of iterations as an integer in range from 1 to $2^{32} - 1$. This parameter is used to tune the length of the function run by specifying number of iterations.

v one byte version umber, currently hardcoded to 0x13.

K signifies a secret value (key) with length from 0 to $2^{32} - 1$ bytes. By default no key is assumed.

X associated data with length from 0 to $2^{32} - 1$ bytes.

y type (mode) of Argon2 to be used. 0 for Argon2D, 1 for Argon2I, 2 for Argon2ID.

Argon2 makes use of the permutation function P which is based on Blake2b hashfunction [3]. The function actually copies blake2b design but additionally it uses 64 bit multiplications. This particular modification makes the function more complicated to implement and optimize for ASICs, while the running speed on X86 processors should be degraded only negligibly. See [6]section 3.6 for detailed explanation.

Another internal function of Argon2 is compression function G which internally uses previously mentioned function P. G accepts two

1024 bytes long inputs and produces one 1024 bytes long output. Let X and Y be the inputs. Firstly, the function XORs them:

$R = X \oplus Y$

R is treated as a 8×8 matrix of 16 byte registers. The function P is first applied to every row of the matrix and then to every column. The result is denoted as Z. Finally, the result is computed as $Z \oplus R$. For more detailed description see [6]section 3.5.

Argon2 makes use of two more hash functions. The H^X function where X denotes the output length in bytes and the whole function is again based on Blake2b. Finally, the variable length hash function H'^X based on H^X defined in [6]section 3.3 is also used.

The argon2 operation can be described as follows. The emphasized variables are Argon2 input parameters, primary and secondary parameters are not distinguished. The numbers in brackets denote the line numbers in algorithm 2.

1. initialisation of the block H_0
2. allocation of the memory according to m and p parameters. The real size of allocated memory is denoted with m' in 1024 byte blocks. Note that the memory is treated as a matrix $B[I][J]$ with p rows and $q = m'/p$ columns.
3. compute $B[i][0]$ for $0 <= i < p$
4. compute $B[i][1[$ for $0 <= i < p$
5. compute $B[i][j]$ for $0 <= i < p$ and $2 <= j < q$. This step is different for every Argon2 mode.
6. if $t > 1$ then repeat step 5 with slight change for every iteration
7. the final block C is computed
8. the output tag is computed

3.2.2 algorithm

This subsection describes Argon2 through pseudocode. In addition to input parameters mentioned earlier, following notations will be used:

|| concatenation of two strings

floor(x) function returns the largest integer which is not bigger than x

ceil(x) function returns the smallest integer which is not smaller than x

LE32(x) converts 32 bit long integer x to byte string in little endian

length(s) returns length of string s in bytes as a 32 bytes long integer

allocate(x) allocates x bytes of memory

3.2.3 Differences in Argon2 versions

Argon2 comes in three versions:

Argon2D data-dependent variant of Argon2. This version uses previously computed data while performing computations and memory access. It is recommended to be used for cryptocurrencies and other proof-of-work applications as well for hashing on backend servers. In general, this version is suited for an environment where no side-channel timing attacks are expected. It provides better protection against brute-force attacks usin specialised hardware and tradeoff attacks.

Argon2I data-independent version. This version does not rely on previously computed data while performing calculations in memory. The version is recommended for key derivation and password hashing, where side-channel timing attacks are more probable because an adversary can have physical access to the machine. The mode is slower because it performs multiple passes over memory to defend against tradeoff attacks.

Argon2ID combination of previous versions. In this mode the function behaves as Argon2I during computation of the first half of the first iteration over the memory. During remaining operation the Argon2D mode is used. The mode combines benefits of mitigation of side-channel timing attacks and brute force attacks.

input : P, S, p, τ, m, t, v, K, X, y
output: TAG

$H_0 \leftarrow \texttt{H64}(\texttt{LE32}(p) \parallel \texttt{LE32}(\tau) \parallel \texttt{LE32}(M) \parallel \texttt{LE32}(t) \parallel \texttt{LE32}(v) \parallel \texttt{LE32}(y) \parallel \texttt{LE32}(\texttt{length}(P)) \parallel P \parallel \texttt{LE32}(\texttt{length}(S)) \parallel S \parallel \texttt{LE32}(\texttt{length}(K)) \parallel K \parallel \texttt{LE32}(\texttt{length}(X)) \parallel X)$
$M' \leftarrow 4 \times p \times \texttt{floor}(m/4p)$
$B \leftarrow \texttt{allocate}(m')$
for $i \leftarrow 0$ **to** $p - 1$ **do**
  $B[i][0] \leftarrow H'^{128}(H_0 \parallel \texttt{LE32}(0) \parallel \texttt{LE32}(i))$
end
for $i \leftarrow 0$ **to** $-1p$ **do**
  $B[i][1] \leftarrow H'^{128}(H_0 \parallel \texttt{LE32}(1) \parallel \texttt{LE32}(i))$
end
for $i \leftarrow 0$ **to** $p - 1$ **do**
  for $j \leftarrow 2$ **to** q **do**
    $B[i][j] \leftarrow \texttt{G}(B[i][j-1], B[l][z])$
    /* indexes l and z are computed differently for every Argon2 version, see subsection 3.2.3 */
  end
end
if $t > 1$ **then**
  for $i \leftarrow 0$ **to** $p - 1$ **do**
    $B[i][0] \leftarrow \texttt{G}(B[i][q-1], B[l][z])$
    for $j \leftarrow 1$ **to** q **do**
      $B[i][j] \leftarrow \texttt{G}(B[i][j-1], B[l][z])$
    end
  end
end
$C \leftarrow B[0][q-1] \oplus B[1][q-1] \oplus \ldots \oplus B[p-1][q-1]$
return $H'^{\tau}(C)$

Algorithm 2: Argon2 function algorithm

The difference lies in line number xx in the algorithm 2. In particular indexes l and z are computed differently within different Argon2 versions. The allocated memory is represented as a matrix $B[I][J]$ with p rows and $q = m'/p$ columns. Before computing indexes l and z, values J_1 and J_2 have to be calculated. This is the difference among Argon2 versions. AFter calculating of these values, further processing needs to be done but this stays the same for all versions.

Rows of memory are called lanes and there are p lanes according to the parameter p, which signifies degree of parallelism. Moreover, the memory is divided into 4 vertical slices, where number of slices is denoted as S. Every intersection of a lane and a slice is called a segment. Segments belonging to the same slice are computed in parallel and they must not reference blocks of each other.

Argon2d computes J_1 and J_2 based on data stored in memory. Therefore it is data-dependent.

$J_1 \leftarrow int32(extract(B[i][j-1], 1))$

$J_2 \leftarrow int32(extract(B[i][j-1], 2))$

Where the function int32(s) converts the 32 bit string s into non-negative integer represented in little endian. The function extract(s, i) extracts ith set 32 bits long from bitstring s while s is indexed from 0.

Argon2i computes indexes by running two rounds of the compression function G in counter mode in the following way

$x \leftarrow G(ZERO, LE64(r)||LE64(l)||LE64(s)||LE64(m')||LE64(t)||LE64(y)||LE64(i)||ZE$

The result x is 64 bits long and therefore it can be viewed as two 32 bit strings

$x_1||x_2 \leftarrow x$

$j_1 \leftarrow int32(x_1)$

$j_2 \leftarrow int32(x_2)$

Inputs for the function G are described below. Note that no data from memory blocks are used. therefore Argon2i is data-independent.

r the pass number

l the lane number

s the slice number

m' total number of memory blocks

t total number of passes

y Argon2 mode

i counter starting from 1 in every segment

ZERO string of zeroes, in the first case 1024 bytes long, in the second case 968 bytes long

Argon2id behaves as Argon2i if the pass number is 0 and at the same time the slice number is 0 or 1. Othervise it behaves like Argon2D.

Further computations common for all Argon2 modes are described in [6]3.4.2.

3.3 Scrypt

Scrypt is a PBKDF introduced by Colin Percival in [31]. At the time of releasing the paper (2009) problems of PBKDF2 and its parallel computation were already well-known. Percival came with two new concepts. He defined a memory-hard algorithm as an algorithm which performs asymptotically almost the same number of operations compared to number of accessed memory locations. The second concept is called sequential memory-hard function. This definition describes a function which can be computed by a memory-hard sequential algorithm and at the same time even the fastest parallel algorithm cannot asymptotically reach a significantly lower cost.

Both concepts came as a reaction to increasing computing power of parallel hardware.

3.4 Other PBKDFs

4 Analysis of LUKS2

This chapter deals with the actual practical research performed as a part of the thesis. The important goal of this thesis is to create a price model for potential attacker trying to get unauthorised access to LUKS2 encrypted partition. Another goal is tightly related. As explained in section 4.1.2, LUKS2 can determine three parameters for Argon2 through its own benchmarking function. It is highly probable that most of Cryptsetup users do not provide their own Argon2 parameters and therefore use the benchmarking function.

Therefore, before the actual model is created, it is necessary to collect sufficient data about parameters used as a result of this benchmark under various real world conditions. The process and results are described in section 4.2. These parameters are further used while simulating attacks on LUKS2 encrypted volumes on real machines. This is described in section 4.3. Results are used to create an actual price model in chapter 5.

4.1 LUKS

LUKS stands for Linux Unified Key Setup. This project started to be developed by Clemens Fruhwirth as a reaction to several incompatible disk encryption schemes which coexisted at the same time at the beginning of 21st century. At certain point there existed three incompatible disk encryption schemes which varried from Linux distribution to Linux distribution. If an user created an encrypted disk, they couldn't be sure if they will be able to encrypt the disk with a different distribution or even with a new version of the same distribution.

LUKS began as a metadata format for storing information about cryptographic key setup. However, Fruhwirth discovered that to design a proper metadata format, he needs to knoww enough information about key setup process [15]. Therefore, he created TKS1 and TKS2. These are templates for the key setup process. Together with LUKS they ensure safe and standardized key management during disk encryption. After some user feedback, LUKS on-disk specification version 1.0 was created in 2005 [14]. Currently the latest version is LUKS on-disk specification version 2 [8].

The reference implementation of both versions of LUKS is called libcryptsetup. The userspace interface is called Cryptsetup. In the following two subsections, default parameters and information concerning command line switches are specific to this implementation.

4.1.1 usage of PBKDFs in LUKS version 1

PBKDF2 function is used as a key derivation function in LUKS version 1. It is used during master key initialisation, adding of a new password, master key recovery, and also during password changing because this operation is actually composed of previously mentioned operations. During all operations it internaly uses a hash algorithm specified by user during initialisation of the LUKS header. By default, SHA256 algorithm is used.

During initialisation, the PBKDF2 function is used to create a checksum of a master key. This key is subsequently used for symmetric encryption of actual data stored on the encrypted disk. The function receives following parameters:

masterKey a new randomly generated master key of user specified length

phdr.mk-digest-salt a random number 32 bytes long which is used to prevent attacks against password using precomputed tables [see 22, section 5.6.3]

phdr.mk-digest-iteration-count number of iterations for PBKDF2, see section 3.1

LUKSDIGESTSIZE length of he computed digest in bytes, default is 20

The generated 20 bytes long checksum is stored in the LUKS header together with the iteration count and salt. Please note that the *phdr.mk-digest-iteration-count* parameter is obtained by performing a benchmark with minimum of 1000 iterations.

During adding of a new password, PBKDF2 is used to process the passphrase supplied by user. It receives following parameters:

password a passphrase supplied by an user

ks.salt randomly generated salt used for this particular key slot with length of 32 bytes

ks.iteration-count number of PBKDF2 iterations

MasterKeyLength length of the derived key

The resulting key derived from the passphrase is used to encrypt the master key. This key has to be present in memory either because initialisation happened recently or it was successfully recovered through a different key stored in different key slot. Together with the encrypted master key, the salt and iteration count are also written into the key slot.

Note that the ks.iteration-count parameter can be influenced by user in several ways [13]. One possibility is to specify number of iterations directly with the command line option `--pbkdf-force-iterations`. Another option is to specify the iteration time through `-i` or `--iter-time` command line options. This option expects a number which signifies number of milliseconds which should be spent calculating the hash. A benchmark is used to calculate number of iterations which corresponds to this time. If no option is specified then the default iteration time of 2000 milliseconds is used.

The function is also used during the master key recovery. This process is performed while unlocking the encrypted partition. During key recovery the PBKDF is actually used twice for every key slot until the master key is decrypted or there are no more key slots to try. First it is used to derive a decryption key from a passphrase supplied by an user. Then this decryption key is used to decrypt the encrypted master key stored in the current key slot. The result is called the candidate master key because we are still not sure if the passphrase was correct. This candidate is again hashed with PBKDF and finally compared with hash of the master key stored in the LUKS header. If hashes match then the passphrase was entered correctly and the master key can be used.

In the first case the function receives following parameters:

pwd user supplied passphrase

ks.salt the salt value read from currently tried key slot

ks.iteration-count the iteration count read from currently tried key slot

masterKeyLength length of the derived key

In the second case the function receives following parameters:

masterKeyCandidate candidate master key, see above

ph.mk-digest-salt the salt value which was used during the initialisation phase and stored in the header

ph.mk-digest-iter number of PBKDF2 iterations used during the initialisation phase and also stored in the header

LUKS_DIGEST_SIZE 20 bytes

The process of changing a password is composed of previously mentioned operations and hence I am not mentioning it here in greater details. To sum it up, firstly the master key is recovered, then a new password is added to a new key slot and the previous one is revoked.

Both password-based encryption and password checking require additional cryptographic primitives which process the derived key. For encryption, reference implementation of LUKS uses aes-xts-plain64 and for hashing it uses sha256. Usage of PBKDF2 requires underlying pseudorandom function. In case of LUKS version 1, default PRF is SHA1. Alternatively, it is possible to choose SHA256, SHA512, ripemd160 or whirlpool.

4.1.2 Usage of PBKDFs in LUKS version 2

LUKS version 2 extends LUKS version 1 and uses similar principles. Therefore, I will focus on differences between version 1 and 2 which are related to usage of PBKDFs. For detailed list of changes see []luks2section 1.1.

The new version supports configurable algorithms for encryption, hashing and also key derivation. That means that the set of algorithms can be extended as new are developed and other are obsoleted. Note that there still exist some requirements for algorithms provided by cryptographic backend []luks2section 4.6. The backend has to support

SHA-1 and SHA-256 hashing functions, PBKDF2, Argon2i and Argon2id key derivation functions and AES-XTS symmetric encryption.

LUKS2 introduces PBKDF memory hard functions Argon2i and Argon2id which are described in 3.2. Argon2 functions should offer increased resistance to brute-force attacks.

The volume key digest is no longer limited by length of 20 bytes, because it no longer relies on SHA-1 hashing function. The processes described in subsection 4.1.1 are the same in LUKS version 2. The same applies for situations in which PBKDFs are used.

As stated in subsection 4.1.1, in case of PBKDF2 user can influence number of iterations directly or specify approximate time required for processing of the passphrase. This stays the same for LUKS2. Functions from Argon2 family introduce two additional parameters; memory cost and parallel cost. Both parameters can be specified through `--pbkdf-memory` and `--pbkdf-parallel` command line parameters respectively. The number of iterations is either benchmarked or it can be manually specified through `--pbkdf-force-iterations`.

The benchmarking function takes as an input a desired unlocking time specified in milliseconds. The function tries to find the best parameters while not exceeding this unlocking time. Note that the number of parallel threads will be always at most 4 and it will decrease if not enough CPUs are online at the time of its usage. In Cryptsetup version 2.6.0 the default unlocking time is set at 2000 miliseconds and the default memory cost is set at 1048576 kB. Both default values can be changed at compilation time.

4.2 Benchmarking ofLUKS2 and Argon2 parameters

In 4.1.2 three parameters for Argon2 were mentioned. They have crucial impact on resistance of the resulting hash against dictionary or brute-force attacks. Their effects are described in detail in section 3.2. From the Cryptsetup's point of view, these parameters impact not only security but user experience as well. If they are set to too low values, the security of encryption keys can be endangered. If they are set to tooo high values, the unlocking process can last undesirably long time or excesive amount of computing power and memory can be used. Cryptsetup does not suppose that all users know exact implications of

choosing particular parameters. Therefore, it introduces a benchmarking function. Note that advanced users can still choose parameters manually.

The benchmarking function takes as an input the desired unlocking time specified in milliseconds. It starts with default parameters and performs several PBKDF computations. By increasing and decreasing parameters during computations it tries to find the most secure (the highest) parameters while not exceeding the unlocking time. Additionally the function performs some estimations to reduce number of required computations because the benchmarking process should not last too long. Average length of benchmark is ... zjistit

Due to this approach the resulting parameters are dependent on several conditions, in particular available hardware and current workload of the system. This can be observed in ... However, as pointed out ..., the resulting parameters are currently very secure.

Currently there exist some limitations which are hardcoded into Cryptsetup. The minimal number of iterations for Argon2 is 4, maximum number is limited by limit of unsigned 32 bit integer for the given platform. The minimum memory parameter is 32 KiB, maximum is 4 GiB. The minimal degree of parallelism for Argon2 is 1, maximum is 4. All the limits are hardcoded in [11]. Note that the number of parallel threads will be always at most 4 and it will decrease if not enough CPUs are online at the time of the benchmark.

In Cryptsetup version 2.1.0 the default unlocking time is set at 2000 miliseconds. The default Argon2 memory cost is set at 1048576 kB. Both default values can be changed at compilation time.

4.2.1 Benchmarking tool

As a part of my thesis I performed collection of Argon2 parameters estimated by the function mentioned above. I created a small tool which can perform series of benchmarks with given parameters and output results in human-readable or machine-readable format. The source code of the tool is part of the thesis (bude na Githubu).

The tool accepts several parameters. It is possible to specify desired unlocking time, maximum degree of parallelism, maximum memory cost and number of benchmarks performed.

The tool can output results in a text format or in a CSV format including headers for better machine processing. The tool does not perform actual benchmarking, it utilizes API of Libcryptsetup library. Therefore the library is required for it to work. The tool does not create any cryptsetup volumes and does not require root privileges.

4.2.2 Methodology of collecting of real world parameters

I used the previously mentioned tool to collect real data on real physical hardware. I tried to simulate various environments in which Cryptsetup can be used to create and unlock encrypted volumes. I decided to use real hardware and not virtual machines because of possible inaccuracies during measurement caused by virtualization layer.

The first hardware configuration consisted of Lenovo Thinkpad P50 laptop with Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz with 4 cores, 8 threads. The laptop was equipped with 32 GiB of SODIMM DDR4 Synchronous 2133 MHz (0,5 ns) RAM [25].

The second configuration comprises Raspberry Pi 3, model B+. This device is equipped with quad core Broadcom CM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz processor and 1GB LPDDR2 SDRAM memory [35]. Note that parameters of this device are considerably lower but there still might be scenarios in which an user might wish to perform drive encryption utilizing this device (network attached storage, home media server). The device was running Raspbian GNU/Linux 9 (stretch).

I have performed following steps on both devices:

1. install required libraries and headers

2. download and unpack the source code of Cryptsetup version 2.1.0

3. configure and compile Cryptsetup without support for udev and blkid to minimize required dependencies, see below for remarks concerning configuration options

4. compile the benchmarking tool linking it to the Libcryptsetup compiled in the previous step

5. simulate hardware limitations (amount of available CPUs, amount of available memory)

6. run series of benchmarks over unlocking times of 1000, 2000, 3000, 4000, 5000, 10000 and 20000 miliseconds with 1, 2, 3 and 4 parallel threads with every benchmark repeated 100 times

During compilation, I decided to run the configure script with `--disable-udev` and `--disable-blkid` options. They disable support for udev and device signature detection through blkid. These features were not needed during benchmarking and they caused complications while installing necessary development libraries.

Simulating various hardware conditions by limiting CPU performance and available memory appeared to be an interesting challenge. My goal was to simulate different number of available CPU cores and different amount of available RAM memory. I wanted to collect results from various combinations of those parameters.

At first I tried to use Control Groups 2 feature of Linux kernel [18]. Unfortunately, I was not able to get desired results. In particular I was not able to guarantee that the benchmarking process will really use limited set of CPU cores. Then I tried to use the Cpulimit project [26]. Unfortunately I still was not able to guarantee that the benchmarking function uses only certain number of CPU cores.

At last I found the solution for limiting number of available CPU cores. On the laptop running Accessible Coconut, I was able to disable individual cores by modifying the file `/sys/devices/system/cpu/cpu`*N*`/online` Where the *N* signifies the number of the core to be turned off. Note that this method allowed me to change state not only of individual cores, but also of individual virtual cores created with hyperthreading technology.

This method could not be applied to raspberry, as the needed file had not existed, probably because of difference of underlying hardware platform. Therefore, I limited number of available CPUs by modifying the `maxcpus=`*n* kernel parameter. This device does not offer hyperthreading.

Due to the way in which the Cryptsetup detects available amount of memory, I decided to perform modification to Cryptsetup it self [12]. I decided to modify directly the function *crypt_getphysmemory_kb*

which returns available physical memory in Kb. I modified it sothat it reads the value from an environment variable, see the patch in the thesis git repository.

After the benchmark I used the Datamash tool to perform basic statistic computations on resulting CSV data [16]. Raw data as well as Datamash results can be found in the thesis Git repository.

4.2.3 Collected benchmarking results

Tables present some chosen results gained through methodology described in the previous subsection. Following conventions are used:

avg(T) average time cost

stddev(M memory cost

avg() average

stdev() standard deviation

The table 4.1 serves as an introduction and it shows which parameters to expect. The most important is the row concerning unlocking time of 2000 milliseconds as these values were used as baseline for simlating of attacks in the next chapter. Tables 4.2, 4.3 and ... demonstrate hardware limitations affecting Argon2 parameters. See the Git repository for more combinations of CPU and memory limitations.

Table 4.1: Benchmark results for laptop with 8 threads and 32 GiB of memory

Unlocking time	avg(t)	stdev(t)	avg(m)	stdev(m)	
1000	4	0	804032.75	3886	
2000	6	0	1048576	0	
3000	9	0	1048576	0	
4000	12	0	1048576	0	
5000	15	0	1048576	0	
10000	31.01	0.099	1048576	0	
20000	64	0	1048576	0	

Table 4.2: Benchmark results for laptop with 4 cores and 4 GiB of memory

Unlocking time	avg(t)	stdev(t)	avg(m)	stdev(m)	
1000	4	0	785173.44	4532	
2000	5	0	1048576	0	
3000	9	0	1048576	0	
4000	12	0	1048576	0	
5000	15	0	1048576	0	
10000	31	0	1048576	0	
20000	61.97	0.298	1048576	0	

Table 4.3: Benchmark results for laptop with 4 cores and 2 GiB of memory

Unlocking time	avg(t)	stdev(t)	avg(m)	stdev(m)	
1000	4	0	781944	1715,758	
2000	6	0	1024000	0	
3000	9	0	1024000	0	
4000	12	0	1024000	0	
5000	15	0	1024000	0	
10000	31.26	0.438	1024000	0	
20000	63.57	1.041	1024000	0	

Tables 4.4 and 4.5 compare benchmarking results for unlocking time of 1000 and 2000 ms respectively. Cryptsetup was compiled with different options explained below:

none no special options were added

SSE the `--enable-internal-sse-argon2` configuration option was added. If this option is specified, additional checks for special CPU features are performed (SSE2, SSE3, AVX2, AVX512). If they are detected, Argon2 is compiled with support for the most advanced one.

Table 4.4: Comparing results for various Cryptsetup compilation options with unlocking time of 2000 ms

flags	avg(t)	stdev(t)	avg(m)	stdev(m)	
none	6	0	1048576	0	
SSE	7	0	1048576	0	
native	6	0	1048576	0	
native + SSE	11	0	1048576	0	
external	6	0	1048576	0	

Table 4.5: Comparing results for various Cryptsetup compilation options with unlocking time of 1000 ms

flags	avg(t)	stdev(t)	avg(m)	stdev(m)	
none	4	0	804032.75	3886.047	
SSE	4	0	949740.83	2305.669	
native	4	0	863822.25	1414.500	
native + SSE	5	0	1048576	0	
external	4	0	846713.44	1359.307	

native the `-march=native` flag is passed to the compiler, optimizing the source code for the particular platform

external Cryptsetup is compiled against external Argon2 library

Notice that combining optimized versions of Argon2 with compilation for the particular platform produces the best results. It almost doubles the number of iterations compared with the benchmark without and it produces only one less iteration compared to unlocking time of 4000 milliseconds benchmarked without optimization. Unfortunately, such combination of configuration options is not very common. A quick review of several Linux distributions (Debian, Ubuntu, Fedora) shows that distributions package the Argon2 library separatly and link the Cryptsetup against it. Also due to the support of broadest array of devices possible platform-specific optimizations are not used.

The table 4.6 shows benchmarking results for the Raspberry Pi. Notice that the memory cost is significantly lower. It is important to

Table 4.6: Benchmark results for Raspberry Pi with 4 cores and 1 GiB of memory

Unlocking time	avg(t)	stdev(t)	avg(m)	stdev(m)	
1000	4	0	43291.74	3628	
2000	4	0	95296.58	8199	
3000	4	0	150860.59	10508	
4000	4	0	203604.14	6723	
5000	4	0	255361.51	10598	
10000	4	0	474446.97	2737	
20000	7.81	0.879	474722	0	

Table 4.7: Benchmark results for Raspberry Pi with optimizations

Unlocking time	avg(t)	stdev(t)	avg(m)	stdev(m)	
1000	4	0	25639.15	4745.422	
2000	4	0	53143.46	1493.866	
3000	4	0	82432.72	2296.222	
4000	4	0	111220.07	1818.082	
5000	4	0	140627.5	1984.865	
10000	4	0	292344.17	1726.421	
20000	4.17	0.375	474722	0	

note here that the benchmarking algorithm of Cryptsetup will never use more than half of available amount of physical memory to prevent out of memory killer from killing the actual benchmarking process. Note also that the algorithm tries to use maximum available memory first and then tries to increase number of iterations.

By comparing tables 4.6 and 4.7 it can be seen that there is not a significant difference in displayed values. The values from the table 4.6 were benchmarked after compilation with `-march=native` flag. The `--enable-sse-internal-argon2` option cannot be used on Raspberry Pi, because the ARM architecture does not offer this kind of CPU instructions. It also confirms the premise that Argon2 should be difficult to optimize for other platforms than X86.

4.3 Attacking LUKS2

This section describes the rest of the practical research. The next chapter then defines and presents the price model which is based on results collected in this chapter. The following three subsections describe utilized hardware and software. The last subsection presents the results.

The motivation behind performing of experimental simulations described in this chapter was to make the price model more precise and realistic. I found some results of benchmarks of Argon2 on CPU or GPU [32],, they are, however, almost four years old. Since that time there appeared new technologies in GPU and CPU segment and the Argon2 went through several upgrades. Another reason was that by using a real hardware I was able to finetune parameters such as number of CPU cores, amount of available memory, available specialized CPU instructions etc.

4.3.1 Utilized hardware

. For the purpose of this thesis I chose to utilize machines offered by the MetaCentrum VO organization because they are easily accessible to registered students and they offer diverse spectrum of computing resources. This proved useful especially considering the fact that according to the definition of an attacker in section 5.1 it is expected to utilize as powerful machines as possible.

The following list describes parameters of the most interesting machines used during the research. For description of all the machines offered in Metacentrum see [27].

Elmo1 4×14 CPU Intel Xeon Gold 5120 2.20 GHz, hyperthreading 2 threads per core, supporting AVX512F instruction, 768 GB of RAM

Doom 2|*times*8 cores Intel Xeon E5-2650v2 2.60 GHz, hyperthreading 2 threads per core, supporting SSSE3 instruction, 64 GiB RAM, nVidia Tesla K20 with 4743 MiB of memory

Konos 2×10 cores Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz, hyperthreading 2 threads per core, supporting AVX2 instruction,

128 GB of RAM, NVIDIA GeForce GTX 1080 Ti with 11178 MiB of GPU memory

nympha 32× Intel(R) Xeon(R) Gold 6130 CPU at 2.10 GHz, hyperthreading 2 threads per core, supporting AVX512F instruction, 192 GiB of RAM

Manegrot 4× 8-core Intel Xeon E5-4627v2 at 3.30GHz, no hyperthreading, supporting SSSE3 instruction, 512 GiB of RAM

Uruk 8 × 18 cores Intel Xeon Gold 6140 3.70 GHz, hyperthreading 2 threads per core, supporting AVX512F instruction, 2.86 TiB of RAM

White 2 × 12 core Intel Xeon Gold 6138 2.0 GHz, no hyperthreading, supporting SSSE3 instruction, 512 GiB of RAM, GPU Tesla P100 with 16280 MiB of memory

Zubat 2 × 8 core Intel Xeon E5-2630v3 2.40GHz, hyperthreading 2 threads per core, supporting SSSE3 instruction, 128 GiB of RAM, GPU Tesla K20Xm with 5700 MiB of memory

While performing computations on CPU I always allocated whole computing host and requested maximum number of available CPUS. I requested always at least $(T/L) \times M$ GiB of memory where T denotes number of available CPU threads, L denotes degree of parallelism for instance of Argon2 being computed and M denotes memory cost. While simulating attacks on GPU I always reserved one instance of GPU, 1 CPU and 1 GiB of RAM memory. Reservations of GPUs are always exclusive. All tests used SSDs as underlying storage.

For the sake of completeness experiments were ran also on the laptop used in the process of collecting benchmarked parameters in the previous section. Parameters are described in subsection 4.2.2.

4.3.2 naive method

The first chosen attack method is naive but simple. It requires Libcryptsetup library to be present because it directly utilizes the API. The main idea is simply to try to recover the master key through normal means but to perform this action in multiple parallel threads.

I created the tool which performs this attack. The tool was based on proof of concept called Dict_search [7]. I modified it sothat it uses threads instead of processes and it measures every recovery attempt and provides statistics. I also improved its usability. The tool can be found in the thesis Git repository.

The tool expects following inputs where letters denoting parameters correspond to the command line options:

t type of encrypted volume (LUKS1, Truecrypt, LUKS2) This can be very easily extended to other types supported by Cryptsetup.

i base name of input device files

p input password file with one password per line

T number of parallel threads

The tool performs basic validation of input parameters and launches defined number of threads. Every thread receives a handler to the specified file containing the encrypted partition. Every thread opens the file with passwords and starts attempting to unlock the keyslot with passwords read from the input file. The file is internaly divided into segments and every thread reads a different segment. The tool runs until the passphrase is found or all the passhprases in the input file had been tried.

Memory footprint of this method can be expressed as $M \times T$, where M denotes the memory cost parameter used during creation of the encrypted partition and T denotes number of cracking threads specified while launching the tool. The number of utilized CPU threads can be expressed as $P \times T$, where P denotes the degreee of parallelism of Argon2 used during initialization of the encrypted drive and T is number of parallel cracking threads.

Because an attacker would try to speed up the cracking process as soon as possible, I compiled Cryptsetup with support for optimized Argon2 using the `--enable-sse-internal-argon2` and I applied a simple patch which disabled clearing of internal Argon2 memory after every Argon2 computation. The patch can be found in the Git repository of the thesis.

4.3.3 Method based on Argon2-gpu-bench tool

This method utilizes a tool called Argon2-gpu-bench written by Ondrej Mosnáček [29]. The tool was originally created to benchmark speed and energy consumption of Argon2 computing hashes. The background scenario was offline brute-force attack against Argon2 hash, which overlaps with goals of my thesis. that is the reason why I selected this tool.

The tool can compute Argon2 using three different modes. It can utilize standard CPU implementation and automatically choose the most optimized one according to supported CPU instructions. Clearing of internal memory is disabled for performance reasons. It also contains Argon2 implementation for OpenCL platform and three different GPU kernels (implementations) to be used with CUDA platform. Every GPU kernel handles the GPU memory in a different way, possibly producing different results under different parameters.

The *Master* kernel uses only shared memory of GPU. The *Warpshuffle* kernel uses only GPU registers. The *Warpshuffleshared* kernel uses combination of both. Additionally, while running in GPU or OpenCL mode, it is possible to select two modes of operation. the *Oneshot* computes individual Argon2 computation as one GPU / OpenCL task, whereas *By-segment* mode divides computation of Argon2 slices among individual GPU / OpenCL tasks.

Argon2-gpu-bench can also precompute values for Argon2i and Argon2id, speeding up the process. This feature is available only for CUDA and OpenCL modes.

Following list describes chosen input parameters for the application. Letters match command line options.

L degree of parallelism

M memory cost parameter

T number of Argon2 iterations

s number of computations

b number of hashes computed in every computation (batches)

Please notice the *s* and *b* parameters, they are important while interpreting results collected through this tool. The tool performs *s*

consecutive discrete computations. Every such computation is measured separately and the tool outputs its length including time of writing and reading of data for OpenCL and CUDA modes. In every such phase b hopefully parallel hash computations is performed. I say hopefully because it is not always possible or desirable. Lenght of every such small computation is measured and averaged at the end of the whole benchmark. So at the end the tool performs $s \times b$ computations.

The tool tries to run as many parallel threads as possible. The number of such threads is determined as $b \times L$. The amount of needed memory is determined as $b \times M$. However, there are upper limits for both values considering effectivity of calculations. While computing on CPU through standard Argon2 CPU implementation or OpenCL platform, the number of available CPU threads should be effectively the limit. While performing computations on GPU through OpenCL or CUDA, the tool cannot run if amount of memory required is higher than amount of memory available at the GPU.

originally the tool generated random strings 64 bytes long. I modified the tool sothat strings are loaded from provided file containing one password per line. My modifications can be found at [34].

4.3.4 Results

Here I describe and compare results collected by two methods described in previous subsections. These numbers serve as direct basis for the price model described in the next chapter. the following list shows column headings used in tables with their description and appropriate units:

T number of threads passed as parameter to cracker

b number of concurent Argon2 computations passed through `-b` parameter to the Argon2-gpu-bench tool

avg(tph) average time spent calculating one hash in milliseconds

stdev(tph) standard deviation of time spent computing one hash in milliseconds

avg(c) average time spent computing one batch of hashes in milliseconds

stdev(c) standard deviation of time spent computing one batch of hashes in milliseconds

Tables 4.8, 4.9 and 4.10 show us a baseline of results. Nympha and Konos were the most powerful machines for computing Argon2 I managed to find in MetaCentrum. We can see that cracking hashes with high memory parameter (1049576 KiB) is highly ineffective with the naive approach. The best result was achieved while running two concurrent unlocking operations, therefore occupying 8 threads. This configuration enabled computation of 1 hash per 1.107 seconds.

However, using powerful enough CPU supporting AVX512F instructions it is possible to get more interesting results. The table 4.9 is very interesting in particular. As we can see, we can get to the speed of 1 hash per 215 milliseconds, that is 4.651 hashes per second. From the theoretical point of view, the most effective computation should occur while running 16 concurrent Argon2 computations using $16 \times 4 = 64$ threads, as Nympha machine offers 64 CPU threads. As we can observe, at this degree of parallelism one hash can be computed in 296.3 milliseconds. However, increasing degree of parallelism up to 512 decreases the time up to 215 milliseconds. This is probably caused by desynchronization of threads during the computing process because it is not possible to have 512 threads running at the same time if every thread requires 1 GiB of memory and the machine has only 180 GiB of memory. apparently threads get managed by operating system scheduler sothat performance is not degraded.

Considering GPU it is possible to get even slightly better times. The GPU provided by Konos machine can run at most 10 parallel Argon2 computations calculating one hash in cca 307 milliseconds, that is 3.25 hashes per second. The table 4.11 shows us that thanks to precomputing values for Argon2i it is possible to reach 294.5 milliseconds per hash. All values in table 4.11 were computed with $b = 10$.

Table 4.8: Cracker running on Nympha cracking drive created with benchmark parameters on laptop

T	avg(tph)	
1	1522.347	
2	1107.5161	
4	1208.7151	
8	2308.0038	
16	4688.9099	
32	8182.64344	
64	16306.40980	
128	35668.50723	

Table 4.9: Argon2-gpu-bench running on Nympha in CPU mode

b	avg(tph)	stdev(tph)	avg(c) | stdev(c)		
1	1641.408	29.667225	1641.408	29.667225	
2	893.785107	21.578192	1787.570	43.156384	
4	525.360194	10.351916	2101.441	41.407664	
8	351.323321	10.219132	2810.587	81.753061	
16	296.345655	4.376179	4741.530	70.018871	
32	252.618991	2.967656	8083.808	94.964996	
64	235.185256	2.112931	15051.856	135.227588	
128	224.986376	2.046735	28798.256	261.982150	
256	219.888724	1.691552	56291.514	433.037402	
512	217.429176	1.660830	111323,76	850.345233	

Table 4.10: Argon2-gpu-bench running on Konos in CUDA mode

b	avg(tph)	stdev(tph)	avg(c) | stdev(c)		
1	2996.175	0.057949	2996.175	0.057949	
2	1499.818	0.025972	2999.637	0.051945	
3	1000.870	0.049965	3002.610	0.149898	
4	752.094474	0.015133	3008.378	0.060533	
5	602.25337	0.007514	3011.267	0.037569	
6	502.370514	0.106702	3014.223	0.640211	
7	430.885932	0.007669	3016.202	0.053687	
8	382.38511	0.079782	3059.081	0.638259	
9	340.40505	0.060572	3063.645	0.545148	
10	306.682325	0.04252	3066.823	0.4252	

Table 4.11: Argon2-gpu-bench running on Konos in CUDA mode comparing various options

option	avg(tph)	stdev(tph)	avg(c) | stdev(c)	
master	306.682325	0.04252	3066.823	0.4252
master + precomputes	294.535537	0.04566	2945.355	0.45661
master + oneshot	305.809563	0.009897	3058.096	0.98977
master + oneshot + precomputes	334.033881	0.008523	3340.339	0.085239
warpshuffle	306.163346	0.83872	3061.633	8.387203
warpshuffle + precomputes	294.932399	0.054746	2949.324	0.547467
warpshuffleshared	306.103173	0.834074	3061.032	8.340742
warpshuffleshared + precomputes	297.277366	0.003769	2972.774	0.037696

Comparing table 4.9 with tables 4.12 and 4.13 shows that there probably exist several factors influencing performance of Argon2 hashing on different CPUs. Note that Manegrot does not support AVX512F instruction nor AVX2 instruction. Therefore the performance is degraded even taking into account higher frequency of its CPU. Uruk is getting close to results of Nympha, but even while having more CPUs with higher frequency, it is still lacking cca 20 milliseconds. I suppose this can be due to the fact that Nympha contains 32 single core CPUs while Uruk contains 8 CPUs with 18 cores each.

The table 4.14 summarizes results of running Argon2-gpu-bench on several different GPU models. We can see that results can vary from 294 milliseconds per hash to 4.5 seconds per hash. All runs were computed using maximum amount of GPU memory and precomputing of values for Argon2i was enabled.

Table 4.12: Argon2-gpu-bench running on Manegrot

b	avg(tph)	stdev(tph)	avg(c) | stdev(c)		
1	3385.726	119.515742	3385.726	119.515742	
2	1622.290	46.643630	3244.581	93.287261	
4	818.661144	20.524608	3290.041	82.098433	
8	657.445886	14.087495	5259.567	112.699966	
16	629.229857	11.710085	10067.678	187.361361	
32	606.408693	9.880113	19405.078	316.163620	
64	594.680230	6.248621	38059.535	399.911800	
128	585.261050	6.576771	74913.42	841.826712	
256	580.427723	6.157127	148589.52	1576.225	

Table 4.13: Argon2-gpu-bench running on Uruk

b	avg(tph)	stdev(tph)	avg(c) | stdev(c)		
1	1420.081	158.506821	1420.081	158.506821	
2	522.104180	83.144792	1044.208	166.289585	
4	323.664122	49.838216	1294.656	199.352865	
8	370.464088	9.467304	2963.713	75.738437	
16	288.062681	36.603874	4609.003	585.661994	
32	262.378906	20.554056	8396.125	657.729801	
64	248.382633	32.444834	15896.489	2076.469	
128	262.579528	56.605614	33610.180	7245.519	
256	243.868085	20.708301	62430.24	5301.325	
512	266.685205	31.304294	136542.84	16027.799	

Table 4.14: comparing Argon2-gpu-bench on various GPUs

Device	b	avg(tph)	stdev(tph)	avg(c)	stdev(c)	
Doom	4	4463.151	0.566438	17852.602	2.265753	
Konos	10	294.535537	0.04566	2945.355	0.45661	
White	15	332.779834	0.388264	4991.698	5.823963	
Zubat	5	1922.469	0.60108	9612.346	0.300544	

Table 4.15: Cracker running on Nympha cracking drive with parameters benchmarked on Raspberry Pi

T	avg(tph)	
1	0.542015	
2	1.118348	
4	1.182632	
8	5.176745	
16	7.727428	
32	12.965203	
64	1039.793171	
128	2065.304367	

Table 4.16: Comparing cracking methods for parameters benchmarked with Raspberry Pi

Device	b	avg(tph)	stdev(tph)	avg(c)	stdev(c)
Nympha	512	12.559353	0.191637	6430.389	98.118714
Konos (GPU)	118	3.132034	0.7081000	369.580075	0.835626
White (GPU)	172	2.433951	0.000472	418.639785	0.81221
Zubat (GPU)	18	15.791761	0.34889	947.505723	2.093391

5 The price of an attack

This chapter focuses on creating a price model for potential attacker trying to gain unauthorized access to a disk volume encrypted with LUKS2 with Argon2 used as PBKDF. First an attacker and available hardware and software options are briefly described. Then the actual price model is introduced and applied to real world examples. The price model is based on real parameters collected during benchmarking in section 4.2 and mainly on computing Argon2 hashes based on these parameters using powerful hardware configurations described in section 4.3.

5.1 Attacker definition

For the purpose of this thesis an attacker is defined as an entity which gained unauthorized access to a LUKS2 volume header with Argon2 used as PBKDF. The header can be part of actual encrypted volume or it can be a detached header stored for example for backup purposes. The volume is not damaged. The attacker has moderate knowledge of information security and computers in general but does not want to invest the time in searching for vulnerabilities in Argon2 or Cryptsetup. The attacker decides to use brute-force or dictionary attack.

The attacker currently does not have any hardware with sufficient computing power and amount of RAM to be used for cracking the passphrase. However, suppose that the attacker has sufficient financial resources to purchase needed hardware or rent cloud computing resources. there is no upper limit set for financial resources.

5.1.1 Hardware

While considering cracking a password or a hash, there are several types of hardware to consider. An attacker can use powerful CPUs which can be cheaper solution compared to other possibilities but they are also usually slower because of their generic nature. The fact which is crucial while considering Argon2 is that they can have access to relatively large volume of RAM without degrading their computing

performance. It is also less time-consuming to eventually optimize the hash algorithm for CPU than for other types of hardware.

The next option often used in this process is to use GPUs. Their architecture is suitable for computing many parallel identical tasks as is the case for hash cracking. GPUs showed to be effective against PBKDF2 [28]. However, the price of a GPU might be higher than the price of a CPU. To use full computing power of GPU it is important that the data being processed is copied into GPU memory which is limited. This fact could reduce their effectivity in this particular case considering possible high memory demands of Argon2.

Last two options are FPGAs and ASICs. They represent two groups of hardware which can be optimized for highly specific tasks. FPGAs are in general less powerful but they can be easily reprogrammed after production. ASICs can incorporate parts which can be later reprogrammed but definitely not to the same extend as FPGAs. ASICs are also more expensive than FPGAs when produced in small volumes. Moreover, to the best of the author's knowledge, there does not exist any publicly available implementation of Argon2 for FPGAs or ASICs and it would take for an attacker nonnegligible amount of time to create one.

5.1.2 Software

There does not exist any publicly available software offering feature to crack passphrases of LUKS2 headers. There exists a simple proof of concept program distributed with Cryptsetup called Dict_search [7]. It mounts dictionary attack against specified device supporting Truecrypt and LUKS1. After slight modifications it can be used also against LUKS2 volumes. It uses API of libcryptsetup. Therefore it obviously introduces slight overhead by creating and deleting structures pertaining to Cryptsetup. Modification of this tool is used in this thesis in subsection 4.3.2

Then there is a possibility of extracting the master key from the captured LUKS2 header and trying to find the right password by comparing it to it after being hashed. This removes some slight overload mentioned above. Support for cracking of Argon2 hashes appeared in well-known open source password auditing software John the ripper

in July 2016. This allows to use some infrastructure offered by the framework but the hash function is not optimized in any way.

Ondrej Mosnáček created an experimental program for benchmarking speed of Argon2 while running on CPUs and GPUs [29]. The project can use both CUDA and OpenCL technologies to run on multiple GPUs or CPUs. The program currently does not perform password cracking, passwords are generated randomly and they are not compared to any hash. But slight modifications to the project could turn it into a password cracker. I modified this tool sothat it reads passwords from a provided file, see [34].

5.2 Price model

Based on previous assumptions of an attacker's options I tried to create a price model which will estimate costs connected with finding the right passphrase to unlock the LUKS2 encrypted volume. These costs include purchase of devices and electricity costs. Following variables were defined to formalize the model:

P number of passwords to be tried

H initial price of one machine (CPU, RAM, accessories expressed in dollars

E price of electricity expressed in dollars per kilowatt

N number of machines

S speed of one machine expressed as number of computed hashes per second

5.3 Real world cost estimation

6 Conclusions

Bibliography

[1] J. Appelbaum and R.-P. Weinman. (Dec. 29, 2006). Unlocking filevault - an analysis of apple's disk encryption system, [Online]. Available: https://events.ccc.de/congress/2006/Fahrplan/attachments/1244-23C3VileFault.pdf (visited on 10/28/2018).

[2] J.-P. Aumasson. (Dec. 6, 2015). Password hashing competition, [Online]. Available: https://password-hashing.net/ (visited on 09/08/2018).

[3] J.-p. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. (Jan. 29, 2013). Blake2: Simpler, smaller, fast as md5, [Online]. Available: https://blake2.net/blake2.pdf (visited on 12/26/2018).

[4] A. Belenko. (2007). Faster password recovery with modern gpu, [Online]. Available: https://www.elcomsoft.com/presentations/faster_password_recovery_with_modern_GPUs.pdf (visited on 12/21/2018).

[5] A. Biryukov, D. Dinu, and D. Khovratovich. (Mar. 24, 2017). Argon2: The memory-hard function for password hashing and other applications, [Online]. Available: https://www.cryptolux.org/images/0/0d/Argon2.pdf (visited on 09/29/2018).

[6] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "The memory-hard argon2 password hash and proof-of-work function", IETF Secretariat, Internet-Draft draft-irtf-cfrg-argon2-04, Nov. 2018. [Online]. Available: http://www.ietf.org/internet-drafts/draft-irtf-cfrg-argon2-04.txt.

[7] M. Broz. (2019). Dict_search tool at cryptsetup git master, [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/tree/master/misc/dict_search (visited on 04/20/2019).

[8] M. Broz, *Luks2 on-disk format specification*, version 1.0.0, Aug. 2, 2018. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/blob/

a1b606803f6d50e0476867fe9d284399504559a3/docs/on-disk-format-luks2.pdf (visited on 09/17/2018).

[9] CipherShed Project, *Ciphershed - secure encryption software*, version 0.7.3.0, Dec. 19, 2014. [Online]. Available: https://github.com/CipherShed/CipherShed/raw/v0.7.3.0-dev/doc/userdocs/guide/CipherShed-User-Guide-0.7.3.0.pdf (visited on 10/28/2018).

[10] F. Corbató, M. Dagget, R. Daley, P. Denning, D. A. Grier, R. Mills, R. Roach, and A. Scherr. (2011). Compatible time-sharing system (1961-1973), Fiftieth aniversary commemorative overview. D. Walden and T. Van Vleck, Eds., [Online]. Available: http://www.multicians.org/thvv/compatible-time-sharing-system.pdf (visited on 12/21/2018).

[11] Cryptsetup developers. (Feb. 2019). Lib/crypto_backend/pbkdf_check.c cryptsetup git master branch, [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/blob/master/lib/crypto_backend/pbkdf_check.c (visited on 04/04/2019).

[12] Cryptsetup developers. (Feb. 2019). Lib/utils.c cryptsetup git master branch, [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/blob/master/lib/utils.c (visited on 04/13/2019).

[13] *Cryptsetup(8) maintenance commands*.

[14] C. Fruhwirth, *Luks1 on-disk format specification*, version 1.2.3, Jan. 20, 2018. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/wikis/LUKS-standard/on-disk-format.pdf (visited on 09/18/2018).

[15] C. Fruhwirth, "New methods in hard disk encryption", Aug. 2005. [Online]. Available: http://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf (visited on 10/03/2018).

[16] A. Gordon, *Datamash*, version 1.4, 2018. [Online]. Available: https://www.gnu.org/software/datamash/ (visited on 02/10/2019).

[17] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y.-Y. Choong, K. K. Greene, and M. F. Theofanos.

(Jun. 2017). Nist special publication 800-63b, Digital identity guidelines - authentication and lyfecycle management, [Online]. Available: https://doi.org/10.6028/NIST.SP.800-63b (visited on 10/25/2018).

[18] T. Heo. (Oct. 2015). Linux/cgroup-v2.rst at master, Control group v2, [Online]. Available: https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/cgroup-v2.rst (visited on 04/04/2019).

[19] IDRIX, *Veracrypt - free open source disk encryption with strong security for the paranoid*, Veracrypt Documentation - Header Key Derivation, Salt, and Iteration Count. [Online]. Available: https://www.veracrypt.fr/en/Header%20Key%20Derivation.html (visited on 10/28/2018).

[20] P. J. Bond, U. Secretary, A. L. Bement, and W. Mehuron Director, "Fips pub 198", May 2002.

[21] B. Kaliski and A. Rush, "Pkcs #5: Password-based cryptography specification", RFC Editor, RFC 8018, Jan. 2017, p. 40. [Online]. Available: https://tools.ietf.org/html/rfc8018 (visited on 10/28/2018).

[22] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, ser. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014, ISBN: 9781466570269. [Online]. Available: https://books.google.cz/books?id=OWZYBQAAQBAJ.

[23] D. Kennedy. (May 29, 2015). Of history & hashes: A brief history of password storage, transmission, & cracking, [Online]. Available: https://www.trustedsec.com/2015/05/passwordstorage/ (visited on 12/20/2018).

[24] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication", RFC Editor, RFC 2104, Feb. 1997, p. 11.

[Online]. Available: https://tools.ietf.org/html/rfc2104 (visited on 12/21/2018).

[25] Lenovo. (2019). Thinkpad p50,model:20eq0022++, [Online]. Available: http://psref.lenovo.com/Detail/ThinkPad/ThinkPad_P50?M=20EQ0022%2B%2B (visited on 04/14/2019).

[26] A. Marletta. (Jun. 17, 2015). Opsengine/cpulimit: Cpu usage limiter for linux, [Online]. Available: https://github.com/opsengine/cpulimit (visited on 02/21/2019).

[27] MetaCentrum VO. (2019). List of hardware, [Online]. Available: https://metavo.metacentrum.cz/pbsmon2/hardware (visited on 04/19/2019).

[28] O. MOSNÁČEK, "Key derivation functions and their gpu implementations", Bachelor thesis, Masaryk university, Faculty of informatics, Brno, 2015. [Online]. Available: https://is.muni.cz/th/sah52 (visited on 09/05/2018).

[29] O. Mosnáček. (2019). Argon2-gpu, [Online]. Available: https://gitlab.com/omos/argon2-gpu (visited on 03/16/2019).

[30] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off", in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630, ISBN: 978-3-540-45146-4.

[31] C. PERCIVAL, "Stronger key derivation via sequential memory-hard functions", Jan. 2009.

[32] A. Peslyak. (Aug. 19, 2015). Argon2 cpu/gpu benchmarks, [Online]. Available: https://lists.openwall.net/phc-discussions/2015/08/19/2 (visited on 04/20/2019).

[33] A. Peslyak and S. Marechal. (2012). Password security: Past, present, future, [Online]. Available: https://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf (visited on 10/21/2018).

[34] V. Polášek. (2019). Argon2-gpu, [Online]. Available: https://gitlab.com/vojtapolasek/argon2-gpu (visited on 04/20/2019).

[35] raspberry.org. (2018). Raspberry pi 3, model b+, [Online]. Available: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf (visited on 03/20/2019).

[36] A. Ruddick and J. Yan, "Acceleration attacks on pbkdf2: Or, what is inside the black-box of oclhashcat?", in *Proceedings of the 10th USENIX Conference on Offensive Technologies*, ser. WOOT'16, Austin, TX: USENIX Association, 2016, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3027019.3027020 (visited on 11/28/2018).

[37] SANS Institute. (Oct. 2017). Password construction guidelines, [Online]. Available: https://www.sans.org/security-resources/policies/general/pdf/password-construction-guidelines (visited on 10/25/2018).

[38] M. S. Turan, E. B. Barker, W. E. Burr, and L. Chen, "Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications", Gaithersburg, MD, United States, Tech. Rep., 2010.

[39] T. Van Vleck. (Feb. 15, 1995). Multics security, [Online]. Available: http://www.multicians.org/security.html (visited on 12/21/2018).

[40] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the weaknesses of pbkdf2", Dec. 2015. DOI: 10.1007/978-3-319-26823-1_9.

[41] Wikipedia contributors. (2018). List of pbkdf2 implementations — Wikipedia, the free encyclopedia, [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_PBKDF2_implementations (visited on 10/28/2018).