Argon2

Nicolò Fornari

February 6, 2017

Introduction

Given the ubiquity of passwords as a mean of authentication for a variety of services and given that users often choose, and possibly reuse, passwords with low entropy, password protection is a topic for which there is constant interest and research.

The first strategy set in place to protect user's passwords from database breaches was to store the hash of the password instead of the password in plaintext, by using cryptographically secure hash functions, meaning holding the properties of pre-image, second pre-image and collision resistance. In this way a database compromised would require the attacker to mount a brute force or dictionary attack on the hashes. However there exists space/time trade-offs such as *rainbow tables* attacks, where the attacker precomputes a list of digests for a specific hash function and stores them into a look up table. Luckily the countermeasure for such attack simply consists of *salting* the passwords, giving to each one a degree of uniqueness which would prevent the attacker from precomputing a look up table for each possible salt.

Nonetheless defenders can not relax yet as hash functions are designed to be fast, allowing the attackers to crack passwords using dedicated hardware such as GPU, FPGA or ASIC. One solution is to slow down bruteforcing by making hardware based attacks too costly. This can be achieved by designing hash functions which require arbitrary amounts of memory and at same do not allow time memory trade-offs. In this report I will discuss about one of such functions: *Argon2*, the winner of the Password Hashing Competion.

Report layout

The report is divided into four sections: in Section 1 I give an overview of the components of Argon2 without dealing with the details, which are left to the official specifications in [1]. Section 2 is about implementation testing and debugging. Section 3 is about the security requirements for a password hashing scheme and how these are met by Argon2. Finally Section 4 consists of a comparison with *scrypt*, a password based key derivation function with some aspects in common with Argon2.

1 Overview of Argon2

In this section I will give an overview of the components used by Argon2: instead of describing the details, which can be found in the official specification [1], I will focus on the implementative choices.

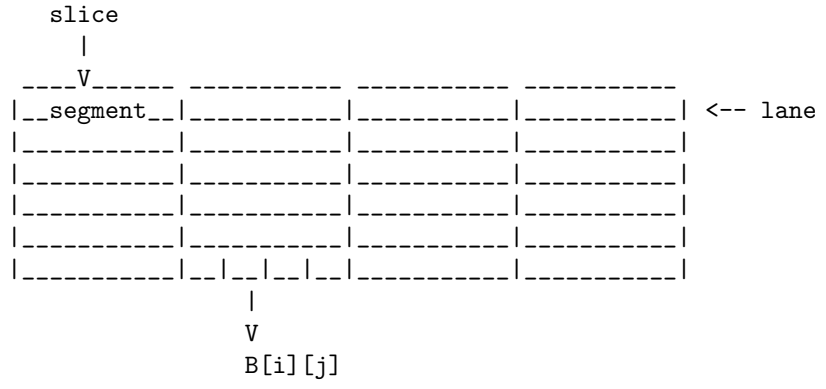
In order to spot at first glance where each *component* is, the code is divided in the following files:

- *main.c*: here the reader can have a first idea of the steps of the algorithm
- *core.c*: it contains three methods used to compute the blocks of memory of matrix B. First `matrix_init`, which computes columns 0 and 1 of B for $t = 0$, then `matrix_first_pass`, which is used to compute the remaining columns for $t = 0$, and finally `matrix_fill_blocks`, which fills all the blocks in a segment, for $t > 0$. Moreover the indexing function, which is made of `compute_J1_J2` and `fetch_ij_prime`, is located in this file.
- *hash_functions.c*: it contains a method to compute the intial hash H_0, namely `compute_H0`, and a method to compute H', `compute_H_prime`.
- *blake2b.c*: it contains our implementation of Blake2b based on its RFC [4]
- *compression_functions.c*: it contains the compression function G divided into three methods: `compression_G`, permutation `P` and `apply_G`.
- *usage.c*: it contains some functions to parse user parameters, check their correctness, set default values for the unspecified ones, display a usage help, display the settings with which Argon2 will run, etc.
- *utils.c*: it contains everything that does not fall in any of the previous categories, just a utility to print a byte array in hexadecimal and a check for each *malloc* call.

Of course we used a *makefile* with some flags: `-fopenmp`, for parallelization with openMP, `-g` for debugging with *gdb* and `-Wall`, for having a good level of warnings.

1.1 The memory

The memory is organized into a matrix B which has p rows, or lanes, and $q = \frac{m}{p}$ columns. The user can specify the amount of memory m' which is rounded down to m as the nearest multiple of $4p$, where p is the degree of parallelism. Each entry $B[i][j]$ is called a *block* and consists of a 1024 byte array. In order to enable parallelism the memory is partitioned in 4 vertical *slices*. The intersection of a slice and a lane is called a *segment*. Each segment clearly contains $\frac{q}{4}$ blocks.



1.2 The hash functions

Argon2 uses two different hash functions:

- H which is Blake2b. It is used to extract entropy from the message and nonce by hashing them. Moreover all the parameters supplied by the user are added to the input, prepended with their lengths.
- H', a variable length hash function based on H. The user can specify the digest length with the parameter τ. H' is used to initialize the first two columns of B for $t = 0$ and as last step of the algorithm to produce the final digest by hashing the XOR of all the blocks in the last column.

1.3 The indexing function

Argon2 come in two flavours:

- Argon2d is faster and uses data-$d$epending memory access which makes it suitable for cryptocurrencies and applications with no threats from side channel timing attacks.
- Argon2i uses data $i$ndependent memory access which is preferred for password hashing and password based key derivation.

On the implementation level the difference between Argon2d and Argon2i lies in the indexing function. The purpose of the indexing function is to compute i', j' given i, j of the current block, in a way that segments of the same slice, which are computed in parallel, can not reference blocks from each other.

The indexing function works in two steps:

1. It gets two 32-bit values J_1, J_2 which are calculated in different ways depending on the type of Argon2 in use. In our implementation this is performed by `compute_J1_J2`.
2. The values J_1, J_2 are mapped to i', j' by constructing a set of indices R from which j' is extracted using a non uniform distribution while i' is set to $J_2 \mod p$. In our implementation this is performed by `fetch_ij_prime`.

In Argon2d J_1 is set to the first 32-bit of $B[i][j-1]$ while J_2 to the next 32. Argon2i behaves differently: compression function G is composed with itself as

$$G([0, ..0], G([0, ..0], Y))$$

with Y a 1024 byte value obtained as the concatenation of various parameters and counters. The result of $G \circ G$ yields 128 values J_1,J_2 which are stored and used by the current segment. Each segment has 128 couples J_1, J_2 available, once they are finished a new call to the indexing function is required.
In our implementation we used the following data structure:

```
struct Segment {
    uint8_t **counter;
    uint8_t **used_pairs;
    uint8_t ***couples;
};
```

The 128 couples J_1, J_2 are stored as a 1024 byte array into `couples[i][j]` for each segment. Every time a pair is used, `used_pairs[i][j]` is incremented by one and of course set to 0 once it reaches 128. The last field of the structure is `counter` which is used as one of the variables forming Y.

Remark 1. *Recall the rule to construct a new block:*

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

The block $B[i][j-1]$ is always excluded from R to avoid passing to G the same value as first and second argument i.e. $G(X, X)$ which would cause a collision.

1.4 Compression function G

1.4.1 Data structures

The compression function $G(X, Y)$ operates on two 1024-byte blocks X and Y. First of all it computes $R = X \oplus Y$. Then R is viewed as a 8×8 matrix of 16-byte registers $R_0, .., R_{63}$

$$R = \begin{pmatrix} R_0 & \cdots & R_7 \\ \vdots & & \vdots \\ R_{56} & \cdots & R_{63} \end{pmatrix}$$

which is found in the code as `uint8_t R[8][8][16]`. Then permutation P is applied rowwise and columnwise. Of course P takes as input 8 values $S_0, ..S_7$, each of 16-byte. These values are viewed as a 4×4 matrix of 8-bytes registers

$$V = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix}$$

where $S_i = (v_{2i+1}||v_{2i})$. V translates into code as `uint8_t V[4][4][8]`.
In our implementation we had to pass from one data structure to another and viceversa:

`uint8_t R_array[1024]` $\longleftrightarrow$ `uint8_t R[8][8][16]`

`uint8_t S0[16],..,S7[16]` $\longleftrightarrow$ `uint8_t V[4][4][8]`

We achieved it quite easily using *for* loops and *memcpy* however this kind of "conversion" may impact on the implementation's performance.

1.4.2 Permutation P

Permutation P is based on the round function of Blake2b and it involves the following operations:

- additions modulo 2^{64}
- 64-bit rotations to the right
- XOR
- x_L, the 64-bit integer x truncated to the 32 least significant bits.

Regarding the last operation I find interesting to explain how I accomplished it:

```
uint64_t xL(uint64_t x) {
        uint32_t maxVal = 0xffffffff;
        return x & (uint64_t)maxVal;
}
```

I compute the maximum value for an `uint32_t`, then I cast it to `uint64_t` so that zeros are added to the most significant bits. Then I perform a bit to bit AND between x and `(uint64_t)maxVal`: in this way only the 32 least significant bits of x survive. Here is an example with a reduced number of bits:

```
0000000011111111 <- maxVal after the cast
&      ...       &
1100101101101101 <- random value for x
-----------------
0000000001101101 <- result
```

The only difference between permutation P and its original design as in Blake2b lies in mixing modular additions with 64-bit multiplications. The motivation behind this choice is to increase the circuit depth of a potential ASIC implementation while having roughly the same running time on a CPU.

2 Testing and debugging

During the development we encountered many program crashes, mainly due to *segmentation fault* or to *corrupted pointers*. Having to face such issues I learned the basics of *gdb*: how to run a program with arguments, how to set and delete breakpoints (at line numbers or at function calls) and how to print the content of variables at run time. This small set of commands allowed us to spot a good number of bugs which made the program crashed. However we did not kill all the bugs in this way: some others were subtler and harder to detect!

I wrote a simple python script [4] to generate different possible inputs to feed to Argon2 with the purpose of detecting possible crashes and checking that the produced digests were consistent between invocations[1].
Of course this approach is very naïve but the idea is to test every possible branch in the program flow. With this method we found a couple of bugs: the first one was triggered for some values of $\tau > 64$. The error lied in the discrepancy between thinking as mathematicians and thinking as programmers:

```
uint32_t r = (uint32_t)ceil(currentTau/32)-2;
```

with `currentTau = 65` we expected `r=1` but it was not: as a matter of fact we did not take into account that division between integers returns an integer making *ceil* having no effect and thus having `r=0`. The fix was to cast the result of the division to float.

The second bug appeared for Argon2i for large values of memory used: $m > 200$ MB. It was caused by a missing instruction to set the counter `used_pairs` to 0 once the 128 pairs J_1, J_2 were used. Of course this bug was not so subtle as the previous one but still hard to spot as a missing instruction in roughly 1000 lines of code.

[1] We experienced this behaviour due to errors in the use of *memcpy*

3 Security analysis

In this section I report a list of security requirements desiderable for a password hashing scheme and I discuss why and how Argon2 matches them.

3.1 Cryptographic security

A password hashing scheme $h : X \to Y$ should possess the following properties:

- *preimage resistance*: given $y \in Y$ it is difficult to find $x \in X$ such that $h(x) = y$
- *second preimage resistance*: given $a \in X$ it is difficult to find $x \in X$ such that $h(x) = h(a)$
- *collision resistance*: it is difficult to find any $x, x' \in X$ such that $h(x) = h(x')$

Despite the fact that the compression function G is neither claimed to be preimage or collision resistance the overall scheme is secure: as a matter of fact the attacker does not have control over the inputs of G. If the attacker manages to find a collision in G for two arbitrary m_1, m_2 such that $G(m_1) = G(m_2)$ then, in order to extend the collision to the overall scheme, he would have to find a corresponding preimage of $H(x)$ (where x is some value determined by backtracking m_1 or m_2 through the algorithm) which is infeasible considering that H is a cryptographically secure hash function. The same reasoning can be applied to preimage and second preimage attacks.

3.2 Defense against lookup table attacks

With a proper use of nonces identical messages produce different digests for different nonces making the precompution of lookup tables for all possible nonces an infeasible task. Moreover the CPU and memory hardness properties of Argon2 would make any precomputation effort in itself infeasible.

3.3 Memory hardness

Attakers using dedicated hardware can be divided in two categories: those with a limited budget thus opting for GPU and FPGA implementations and others with more resources opting for more efficient and more expensive ASIC implementations.

In order to estimate the attack-cost the time-area product is used as a metric. Defenders look for the upper limit to the memory reduction factor an attacker can achieve without increasing the time-area product. If that limit is sufficiently low then the scheme is sufficiently memory hard.

The time-area product is denoted by AT, where A is the ASIC area corresponding to some memory M, while T is the running time. According to Biryukuv et al [1] a scheme is called *memory-hard* if, given a cracking implementation requiring:

- reduced memory αM (with $\alpha < 1$)
- increased running-time cost $D(\alpha)$ and thus having a gain $g = max_\alpha \frac{1}{\alpha D(\alpha)}$ of time-area product

it holds that $D(\alpha) > \frac{1}{\alpha}$ as $\alpha \to 0$. In order to evaluate a scheme's memory-hardness, defenders have to determine the maximum gain g that attackers can achieve using time-memory tradeoffs attacks.

Argon2d
Attacks targeting *data-dependent* schemes can reduce the time-area product as long as $D(\alpha) \leq \frac{1}{\alpha}$. Let's look at the time and computational penalties for a generic tradeoff attack against Argon2 indexing function:

α	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$
$C(\alpha)$	1.5	4	20.2	344	4660	2^{18}
$D(\alpha)$	1.5	2.8	5.5	10.3	17	27

The designers of Argon2 claim that an ASIC-equipped adversary can not decrease the time-area product if the memory is reduced by a factor of 4 or more, with much higher penalties for an increased number of passes over memory.

Argon2i
The Argon2i variant is more vulnerable to tradeoff attacks due to its independent indexing function however 3 passes over memory rather than one are chosen as default.

3.4 Side channel attacks

As already mentioned Argon2 comes in two flavours: Argon2d and Argon2i. If side channel attacks are possible threats in the use case of Argon2, then the user should opt for Argon2i. Let's try to understand the reason better: Argon2d uses data dependent memory access in its indexing function which has the advantage of prohibiting precomputation of blocks, thus requiring less memory, but on the other side the disadvantage of being vulnerable to side channel attacks, eg. cache timing attacks. Moreover Argon2d makes by default a single pass over memory and does not overwrite it[2], becoming vulnerable to Gargage Collector attacks.

4 Comparison with *scrypt*

Scrypt is a password-based key derivation function created by Colin Percival. The algorithm is designed to make hardware attacks costly by requiring significant amounts of memory.

The definition of scrypt consists of a stack of subprocesses which may confuse at first reading:

$$scrypt(P, S, N, r, p, dkLen) = MFcrypt_{HMAC_{SHA256}, Smix_r}(P, S, N, p, dkLen)$$

with $SMix_r$ defined as:

$$SMix_r(B, N) = ROMix_{BlockMix_{Salsa20/8, r}}(B, N)$$

However I substituted with patience the definitions of each procedure coming up with a clearer idea of the algorithm which I report here as pseudocode.

[2]Unless the *memory wiping* option is enabled by the user

```
1  function BlockMix(B[0],..,B[2r-1]) // it takes 2r k-bit blocks
2          X = B[2r-1]
3
4          for i=0..2r-1
5                  X = Salsa20/8(X ^ B[i])
6                  Y[i] = X
7          end
8
9          return (Y[0],..,Y[2r-1])
10 end
11
12 function SMix(B[0],..,B[2r-1],N) // it takes 2r k-bit blocks and N
13         X = (B[0],..,B[2r-1])
14
15         for i=0...N-1
16                 V[i] = X
17                 X = BlockMix(X)
18         end
19
20         for i=0..N-1
21                 j = Integerify(X) mod N
22                 X = BlockMix(X ^ V[j])
23         end
24
25         return X
26 end
27
28
29 PB[0],..,PB[p-1] = PBKDF2(HMAC_SHA256, Passphrase, Salt, 1, p* PBLen)
30
31 for i=0..p-1
32         PB[i] = SMix(PB[i],N)
33 end
34
35 DK = PBKDF2(HMAC_SHA256, Passphrase, PB[0] || .. || PB[p-1],1, dkLen)
```

The algorithm uses `PBKDF2` to generate p blocks of length $PBLen$. Each block is passed to `SMix` and transformed *independently* from the others. Then the blocks PB are concatenated and used as salt for `PBKDF2` giving in output the derived key DK.

The *for* loop at line 31 is where parallelization happens: as a matter of fact the blocks PB can be computed independently one from the other.

The parameter N determines how much memory is needed: if we look at the for loop at line 15 a vector V of length N is created and its elements are accessed in a pseudorandom order - line 21 and 22 - by a function `Integerify` which is a bijiection from $\{0,1\}^k \to \{0,..,2^k-1\}$.

Argon2 and scrypt

Both Argon2 and scrypt allow the user to choose the degree of parallelism and how much memory to use. In scrypt parallelism is implemented in a straightforward manner: each thread takes care of computing a single block PB. Argon2 makes use of parallelism in a more advanced fashion:

with p threads and l joining points (the segments). The designers of Argon2 chose $l = 4$ as it is a value that gives low synchronization overhead while imposing time area penalties on the adversary who reduces the memory even by the factor $\frac{3}{4}$.
Moreover Argon2 allows the user to choose the number n of passes over memory, which increases the running time linearly. These considerations are summed in the following table:

Comparison	**Argon2**	**scrypt**
Degree of parallelism	p	p
Sync points	4	$\times$
Memory	m	N
Core hash function	$Blake2b$	$Salsa_{20/8}$
Variable digest length	✓	✓
Number of passes	✓	$\times$

Conclusions
Argon2 implements parallelism in a more advanced manner than scrypt however Argon2 is optimized for $x86$ architectures thus it is not a good candidate as KDF for disk encryption on ARM-based smartphones. To my understanding both Argon2 and scrypt are secure KDF, provided that are used with proper parameters.

References

1. A. Biryukov et al., *Argon2: the memory-hard function for password hashing and other applications*,
   `https://password-hashing.net/argon2-specs.pdf`
2. Colin Percival, *Stronger key derivation via sequential memory-hard functions*,
   `http://www.tarsnap.com/scrypt/scrypt.pdf`
3. Jos Wetzels, *The Password Hashing Competition and Argon2*,
   `eprint.iacr.org/2016/104.pdf`
4. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)
   `https://tools.ietf.org/html/rfc7693`

Appendix

Python script to test Argon against crashes and inconsistent digests.

```
import os

def params(a,m,p,l,t):
        cmd = "./argon2 -v 0 -a %d -m %d -p %d -l %d -n %d" % (a,m,p,l,t)
        print cmd

        p1 = os.popen(cmd,"r")
        digest1 = p1.readline()[:-1]
        p2 = os.popen(cmd,"r")
        digest2 = p2.readline()[:-1]

        if digest1 == digest2:
                code = " OK"
        else:
                code = " ERR"
        log.write(cmd+code+" \n")


def fuzz():
        argon = [0,1] # respectively argon2d, argon2i
        memory = [20,30,40,50]
        parallelism = [1,2,3,4,5,6,7,8]
        length = [32,64,80,128,256]
        passes = [1,2,3,4]

        for a in argon:
                for m in memory:
                        for p in parallelism:
                                for l in length:
                                        for t in passes:
                                                params(a,m*8*p,p,l,t)



log = open('log','w')
fuzz()
log.close()
```