Argon2
and a comparison with Yescrypt

Elena Canali

Partner: Cristian Mirto

1 Introduction

Passwords are the most widely used form of authentication that allow to control access to resources. A great amount of passwords has low entropy; at the same time attackers develop more and more sophisticated techniques and use advanced technologies. In this scenario password protection is a crucial issue to constantly develop and strengthen.

The first attempt of protection was storing the hash of passwords instead of the password in plaintext, using cryptographically secure hash functions. However this was not enough since attackers precomputed lookup tables containing a list of digest corresponding to specific hash functions (*rainbow tables*) and easily break many weak passwords.
This problem was overcame by introducing a *salt*, another variable amount that introduce some entropy and make lookup tables useless.
Another drawback was exposed when attackers used dedicated hardware, such as FPGA, multiple-core GPU, ASIC, for their performances: hash functions are designed to be fast and this kind of hardware are designed to perform a huge amount of computation. One possible solution is to replace commonly used hash function (that are as already mentioned fast) with functions that require certain amount of memory and do not allow time-memory tradeoff.
With the purpose to overcome this weaknesses, in 2013 started a *Password Hashing Competition* [1]. After the evaluation of the candidates based on security, simplicity and functionality properties, the winner is announced to be *Argon2.*

This report aims to present our implementation of the scheme Argon2 and a brief comparison with one of the other finalist of the competition *yescrypt.* In section 2, we will describe the structure of the scheme and the algorithm steps focusing on some unclear issue about official specification. Section 3 is devoted to some security consideration concerning Argon2. The last section, section 4, will contain a brief description of the Password Hashing Scheme yescrypt and a comparison with the winner of the Password Hashing Competition.

2 Overview on Argon2 Password Hashing Scheme

In this section we will provide an overview of the Password Hashing Scheme Argon2 deepening some unclear issue about the specification. Not mentioned details can be found in [5].

There are two main versions of Argon2, *Argon2i* and *Argon2d*. *Argon2id* and *Argon2ds* are two further minor versions. We will focus on Argon2d, Argon2i and Argon2id that are the versions we implemented.

- Argon2d is faster and uses data *d*epending memory access. This feature makes it suitable for cryptocurrencies and applications threatened by side-channel attacks.
- Argon2i uses data *i*ndepending memory access. For this feature it is therefore to be preferred for password based key derivation and password hashing.
- Argon2id is a hybrid between Argon2d and Argon2i. Because of this mix between the two previous versions, we will just briefly mention how it is construct and deeply explain Argon2d and Argon2i.

2.1 Input

Inputs of Argon2 are of two types: primary and secondary inputs. Primary inputs are:

- **P**: the password to be hashed. Our program accepts passwords from 0 to $2^{23} - 1$ bytes.
- **S**: the salt involved in password hashing process. Our program accepts passwords from 8 to $2^{23} - 1$ bytes.

Primary inputs are also the mandatory input of our program. If the user does not insert one of this two values, a message error is shown and the program stops. Secondary inputs are:

- **p**: degree of parallelism. Accepted values are in the range $[1, 2^{24} - 1]$
- **T**: tag length. Accepted values are in the range $[4, 2^{32} - 1]$. We will also refers to this parameter with τ
- **m**: memory size. Accepted values are in the range $[8p, 2^{32} - 1]$
- **t**: number of iteration. Accepted values are in the range $[1, 2^{32} - 1]$
- **v**: version number
- **K**: secret value. Not mandatory data; if specified, it must be in the range $[0, 32]$ bytes

- **X**: data associated to K. Accepted values are in the range $[0, 2^{32} - 1]$
- **y**: Argon2 type. Accepted values are 0 for Argon2d, 1 for Argon2i and 2 for Argon2id.

In our program, inputs can be inserted in two modalities. The first one is from command line using the commands *-z*, where z is one of the parameter specified above. As already mentioned, primary inputs are mandatory; if secondary inputs are not specified, default values are set. Another possibility to provide inputs is to use a .txt file: the user can specify in suitable file text the desired values and run the program using the command *-IF*.

In our implementation we used the struct *argon2_ctx* to store, after some management, primary and secondary inputs. In *argon2_ctx* are also stored some other useful parameters: the lengths of the variable inputs, the prehashing digest and some parameters concerning the memory. We took advantage of another struct, called *blocks*, that represents the"unit" of the memory (see 2.2 for further detail about the memory arrangement). This structures together with some useful constant are stored in `common.h`, a header file included in two file of our project, `main.c` and `argon2_core.c`.

The management of the inputs and the outputs of Argon2 is performed in `main.c`. After some required checks on the inputs, it is called the function *getTag_argon2* that actually computes the Argon2 algorithm; it is implemented in `argon2_core.c` that actually contains the core of operations of the Password Hashing scheme.

2.2 Memory

As mentioned in the previous section, the user can specifies the amount of memory m' to use in the algorithm, which is m rounded down to the nearest multiple of $4p$. The memory is organized in a matrix B of p rows, or lanes, and $q = \frac{m'}{p}$ columns. Entries of B, $B[i][j]$, are 1024 byte array and are called *blocks*. Columns of B are split in 4 *slices*; the intersection between slices and lanes is called *segment*, that contains $\frac{q}{4}$ blocks. The designers of Argon2 chose to divide memory in 4 slices as this value gives low synchronization overhead while imposing time-area penalties on the adversary who reduces the memory even by the factor $\frac{3}{4}$.

2.3 Argon2 core

In this section we will briefly describe the operations computed by Argon2 focusing on some unclear issue. We will provide some detail of our implementation specifying the operation of the most significant functions.

The topic of this section will be the file `argon2_core`, since, as already mentioned, it collects the core of the algorithm, i.e. the operation performed to get the final tag. The main function of this file is *getTag_argon2* that calls other three functions performing the macro-steps of the algorithm.

2.3.1 *init_argon2*

The first function called is *init_argon2*. It first extracts entropy from the inputs computing a 64-byte long prehashing digest. In order to do that, the hash function *Blake2b* is applied to parameters and inputs P, S, K, X prepended with their length. The result of this operation is stored in the field H_0 of the struct *argon2_ctx*. After that, the first two column of B matrix are filled in the following manner:

$$B[i][0] = H'(H_0 \| \ 0 \ \| \ i, 1024), \ 0 \leq i < p,$$
$$B[i][1] = H'(H_0 \| \ 1 \ \| \ i, 1024), \ 0 \leq i < p.$$

H' is a variable length function build upon Blake2b; for a detailed description of this function refer to [5]. The last argument of H' (in this case 1024) indicates the length of the resultant tag: from specifications it is not clear the length of the H' output since it seems to be the parameter τ. We notice that this could not be possible since B have 1024 byte long blocks. Consulting the official implementation [2], we confirm that we misunderstood the specification and that our hypothesis was correct.

2.3.2 *fill_memory*

After this first set of operations, it is called *fill_memory*, the function that deals with computing the remaining blocks of B. In order to do that, Argon2 makes use of a compression function G and an indexing function ϕ: this functions will be discussed in detail in 2.4 and 2.5 respectively.
At first, in case of Argon2i or Argon2id, a helpful array J is computed (also for Argon2d the array J is computed; in this case its computation is very simple despite to the other two cases. For this reason we did not construct a specific function for this case): it will be used in the indexing function. A detail description of how it is constructed will be provided in 2.5.
Since blocks of the first two columns of B are already computed, *fill_memory* computes the remaining blocks in the following manner:

$$B[i][j] = G(B[i][j-1], \phi(i,j)), \ 0 \leq i < p, \ 2 \leq j < q$$

If the parameter t is greater then 1, further computation on B are performed, for $t-1$ times, in the following way:

$$B[i][0] = G(B[i][q-1], \phi(i,0)) \oplus B[i][0], \ 0 \leq i < p,$$
$$B[i][j] = G(B[i][j-1], \phi(i,j)) \oplus B[i][j], \ 0 \leq i < p, \ 1 \leq j < q.$$

This function is also responsible to handle parallelization since the computation is divided in p threads. p is not a mandatory parameter: we set 4 as default value for p.

2.3.3 *finalize*

After t iteration over memory, it is called *finalize* that is responsible to perform the last steps of the algorithm. It first XOR the blocks of the last column of B obtaining the final block C. Then H' is applied to C in oder to get the final tag: this is the first time during the algorithm where the parameter τ is used.

2.4 Compression function G

The compression function G outputs one 1024 byte block, starting from two input blocks X and Y. The function makes use of a permutation $\mathcal{P}$ a slightly different version of the *Blake2b* round function, where modular additions presents in *Blake2b* are combined with 32-bit multiplications: for a complete description of $\mathcal{P}$ see [5]. This variation was introduced to increase the circuit depth (and thus the running time) of a potential ASIC implementation while maintaining roughly the same running time on CPU.
At first the compression function G XOR its two inputs, obtaining the 1024 byte block R. In order to easily understand how G works, R is viewed as a 16 byte 8×8 matrix: $\mathcal{P}$ is indeed applied rowwise (getting Q) and columnwise to get Z. The output is the XOR between R and Z. Summarizing,

$$G: \quad (X, Y) \rightarrow R = X \oplus Y \xrightarrow{\mathcal{P}} Q \xrightarrow{\mathcal{P}} Z \rightarrow Z \oplus R$$

2.5 Indexing function ϕ

The description of the algorithm so far is the same for the three versions of Argon2. The difference between Argon2d, Argon2i and Argon2id lies in the indexing function. The indexing function ϕ takes as input two values i and j that identifies one block of B and the counter of iteration w over memory: the outputs are other three values i', j' (position), w' (iteration) that identifies the resultant block. ϕ first computes a 64 bit value $J = J_1||J_2$ and then maps J_1, J_2 in the reference block indexes i' and j': we will describe this two steps separately.

Getting J:
$J = J_1||J_2$ is computed differently in the three versions of Argon2 scheme:

- **Argon2d**: J_1 is set to the first 32 bit of $B[i][j-1]$ at iteration w; the next 32 bit of the same block are J_2. If we are in $B[i][0]$, with $w > 1$, we'll take the bits of $B[i][q-1]$ computed at iteration $w-1$.

- **Argon2i**: in this case the value of J is computed at the beginning of a segment as follow: it is computed the 1024 byte block

$$G(\underbrace{\mathbf{0}}_{\text{1024 bytes}}, G(\underbrace{\mathbf{0}}_{\text{1024 bytes}}, \underbrace{w}_{\text{8 bytes}} || \underbrace{i}_{\text{8 bytes}} || \underbrace{s}_{\text{8 bytes}} || \underbrace{m'}_{\text{8 bytes}} || \underbrace{t}_{\text{8 bytes}} || \underbrace{y}_{\text{8 bytes}} || \underbrace{c}_{\text{8 bytes}} || \underbrace{\mathbf{0}}_{\text{968 bytes}}))$$

  where G is the compression function used in the counter mode, $s = \lfloor \frac{j}{4} \rfloor$ is the current slice, c is a counter starting from 1. The block is viewed as 128 segment $J_1||J_2$ to use in orderly manner any time the indexing function is called. When ϕ is called 128 times, i.e. there are no $J_1||J_2$ available, the compression function is called again whit counter c increased by 1.
  The description of this part of the algorithm present some discrepancy between the specification [5] and the official implementation [2]: in the implementation, in case of first pass, first slice, the fist two segment $J_1||J_2$ generated as already specified, are discarded. This detail is omitted in specification; we chose to follow official implementation algorithm [2] in order to be able to verify the correctness of our output.

- **Argon2id**: this version of Argon2 behaves as Argon2i in the first two slices of the first iteration over memory. In the other slices of the first iteration and for the other passes, the access to B is data dependent as in Argon2d.

Mapping J to the reference block index:
J is used to compute the values i', j' and w' that identifies the resultant block.

- If we are operating in the first slice during the first iteration, i' is set to i; otherwise it is set to $J_2 \mod p$.

- In order to compute j', it is necessary to create a set of indexes $\mathcal{R}$ in the following manner:

  - If $i' = i$ and $w = 1$, $\mathcal{R}$ includes all the blocks computed in this lane that are not overwritten yet, excluding $B[i][j-1]$. If $i' = i$ and $w > 1$ $\mathcal{R}$ includes the indexes of already computed blocks in the last three segments plus already constructed blocks in this segment excluding $B[i][j-1]$. This last particular miss in the specification [5]: also in this case we followed the algorithm of the official implementation [2].
  - If $i' \neq i$, $\mathcal{R}$ includes all the blocks in the last 3 segments computed and finished in lane i'. If $B[i][j]$ is the first block of a segment, then the very last block from $\mathcal{R}$ is excluded.

  Afterwards, the value of j' is computed as

$$j' = |\mathcal{R}| \cdot \left(1 - \frac{(J_1)}{2^{64}}\right)$$

3 Security analysis

This section will report some observation about security of Argon2 Password Hashing Scheme. Only some security aspects will be reported: for a full security analysis refers to [5].

3.1 Memory hardness and Ranking tradeoff attacks

Adversaries to be considered in security analysis are the one with access to the most powerful resources: we suppose therefore to deal with ASIC implementations.

In order to estimate the attack-cost, the time-area product is used as a metric. In algorithm design and in security analysis, defenders aim to reach the upper limit of memory reduction factor that an attacker can achieve without increasing the time-area product. This is the roughly speaking the concept of *memory-hard* functions. More formally, if 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, the scheme is called *memory-hard* if, supposing that:

- An attacker uses reduced memory αM, with $\alpha < 1$
- The corresponding running time increases by a $D(\alpha)$ factor

then we have $D(\alpha) > \frac{1}{\alpha}$ as $\alpha \to 0$. Clearly in this case the time-area product does not decrease as the maximum gain

$$\epsilon_{max} = max_{\alpha} \frac{1}{\alpha D(\alpha)}$$

is less then 1.

In order to analyze resistance of Argon2 against ASIC-equipped adversary and Argon2 memory hardness, we considered the best known attack, the ranking tradeoff attack [4]. The idea behind this attack is that each time one block is computed, the attacker makes a decision, to store it or not. If the block is not stored, it is computed the access complexity of this block (number of calls needed to compute the block). If the block is stored, the memory employed by the storing is to be considered; on the other hand, the access complexity is 0. For a detailed description of this attack see [4].

Penalties due to this kind of attack are different for the two main version of Argon2:

- **Argon2d**
  the following table report values of α and $D(\alpha)$ of one-pass scheme for the ranking tradeoff attack:

α	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$
$D(\alpha)$	1.5	2.8	5.5	10.3	17	27

We conclude that the adversary is able to reduce memory by factor of 3 still keeping time-area product the same.

- **Argon2i**
  Argon2i is more vulnerable to tradeoff attacks due to its independent indexing function; however 3 passes over memory rather than one allows to reduce time-area product cost by 3 at best.

3.2 Lookup table attacks

Assuming the user of Argon2 properly uses salts (i.e. uses them to introduce entropy), identical passwords produces different digests. This makes the pre-computation of lookup table, i.e. the pre-computation of a list of digest with the purpose to break the hash function, an infeasible task. This pre-computation is unachievable both for the exponential number of computation and for CPU and memory-hardness property of Argon2. This considerations ensure good protection even for low entropy passwords.

3.3 Cryptographic security

Hashing scheme are required in general to be collision resistant, preimage resistant and second preimage resistant.
Despite the fact that the compression function G is not claimed to have this property, the scheme is still secure as attacker has not control over inputs of G. Suppose an attacker manages to find some collision of G i.e. suppose he wants to find two arbitrary x_1, x_2 values such that $G(x_1) = G(x_2)$. To be able to extend the collision to the initial inputs, he has to compute the preimage of Blake2b, that is an infeasible task since Blake2b is a hash function proved to be cryptographically secure (so collision resistant). Similar consideration can be achieved for preimage and second preimage resistance.

4 yescrypt: algorithm and comparison with Argon2

yescrypt [7] was one of the finalist of the Password Hashing Competition. It was not the winner of the competition but it received a special recognition thanks to some of its interesting peculiarities that we will highlight in the following sections. yescrypt is designed to be completely compatible with scrypt: it computes indeed classic scrypt and native yescrypt hashes. Specification and analysis of yescrypt are, for this reason, a constant comparison between this two scheme. This makes their comprehension something hard unless scrypt is well known.

This is the reason why we will constantly mention scrypt architecture [6] in this section.

The definition of scrypt, and consequently of yescrypt, consists of a stack of subprocesses that makes the view of the algorithm a bit complicated. In the following sections we will briefly describe the main structure of the scheme and some of the most important functions used in it

4.1 Main algorithm

Inputs of the scheme are the same of scrypt (and Argon2); some addition parameters are introduced in order to solve some problematic issues of scrypt and to adjust innovation of yescrypt:

- **t**: time cost parameter; it 32-bit number controlling yescrypt's computation time while keeping its peak memory usage the same. $t = 0$ is optimal in terms of achieving the highest normalized area-time cost for ASIC attackers.
- **N**: memory cost parameter. It is not a "new" parameter but its allowed value are different w.r.t. scrypt: at the beginning, in scrypt specification it was claimed that this parameter could be an arbitrary integer; some inefficiencies were later identified for values different from power of 2. yescrypt therefore accepts powers of 2 as value for N.
- **haveROM**: point out if ROM (Read Only Memory) is used in the computation or not. It is a pre-filled read-only lookup table used along with scrypt's usual sequential-write, random-read lookup table (Random Access Memory, RAM).
- **YESCRYPT_RW/YESCRYPT_WORM**: flag allowing to enable the individual extra features of yescrypt. Setting the former flag changes usage of RAM to "read-write"; the latter stands for "write once, read many" and is the conservative enhancement of classic scrypt.

The main structure of scrypt and yescrypt algorithm is basically the same. It is linear and simple despite the functions used in it. We will report the pseudocode:

```
PB[0],...,PB[p-1] = PBKDF2_HMAC_SHA256(psw, salt)

for i=0 to p-1{
  B[i] = SMix(PB[i],N)
}
DK = PBKDF2_HMAC_SHA256(psw, PB[0] || ... || PB[p-1])
```

The algorithm uses the well known key derivation function PBKDF2 to generate p blocks, where p is the parallelism parameter. Each block is transformed using *SMix*. Then the blocks *PB* are concatenated and used as salt for PBKDF2 giving in output the derived key *DK*. Other cryptographic primitives, HMAC and SHA_256, are used during the computation. Cryptographic security of yescrypt (collision resistance, preimage and second preimage resistance) is based on that of SHA-256, HMAC, and PBKDF2. The rest of processing, while crucial for increasing the cost of password cracking attacks, may be considered non-cryptographic.

4.2 Main functions and parameters

4.2.1 *ROMix* and ROM

The description of *SMix* is based on *ROMix*, a crucial function for both scrypt and yescrypt. We will therefore briefly describe *ROMix* first.

This function contains the first crucial change introduced with yescrypt, the interleaving of RAM and ROM. The use of RAM makes *ROMix* sequential memory-hard i.e. makes it a function where the fastest sequential algorithm is memory-hard and additionally where it is impossible for a parallel algorithm to asymptotically achieve a significantly lower cost. ROM was introduced to use alternative type of memory other then RAM. It also provide excellent scalability to multi CPUs and CPU cores. However this changes provide good scalability of attacks to more computing power while not having to provide more memory. In order to avoid this kind of problems, yescrypt interleave the two memory access type.

We will provide the pseudocode of the yescrypt's *ROMix* function. The following parameters will be used:

- H: hash function; k is the length of its output. In yescrypt are used two hash function: *BlockMix_Salsa20/8*, based on *Salsa20/8* (already used in scypt) and *BlockMix_pwxform*, based on *pwxform* (introduced in yescrypt scheme). See [7] for a detailed description of this hash functions.
- **Integerify(X)**: bijective function from $\{0,1\}^k$ to $\{0,\ldots,2k-1\}$.
- **B**: input of **k** bit length
- **N**: integer work metric
- **VROM**: optional ROM lookup table of **NROM** blocks
- **Nloop**: even value derived from N

```
for i=0 to N-1{
  V[i] = X
    if (haveROM){
      j = Integerify(X) mod NROM
      X = X XOR VROM[j]
    }elseif (YESCRYPT_RW){
```

```
7        j = Wrap(Integerify(X),i)
8        X = X XOR V[j]
9      }
10   X = H(X)
11 }
12 for i=0 to Nloop-1{
13   if (haveROM){
14       j = Integerify(X) mod NROM
15       X = X XOR VROM[j]
16     }else{
17       j = Integerify(X) mod N
18       X = X XOR V[j]
19       if (YESCRYPT_RW){
20         V[j] = X
21       }
22     }
23   X = H(X)
24 }
```

The function *Wrap(x,i)* is defined as $(x \mod p2floor(i)) + (i - p2floor(i))$; as the name suggests, *p2floor(i)* is the largest power of 2 not greater than argument.
If we skip the if loop at line 4, the one at line 13 and finally the one at line 19 and if we substitute *Nloop* with *N*, we get the *ROMix* function of scrypt.

4.2.2 *SMix* and YESCRYPT_RW

Another variation w.r.t. scrypt implementation is due to the YESCRYPT_RW flag that introduces additional random writes into RAM, which classic scrypt did not perform at all. To be noted that it is usable (and is almost always beneficial to use) regardless of whether a ROM is in use or not.

Moreover the YESCRYPT_RW flag moves the parallelism to a slightly lower level, inside *SMix*. This reduces flexibility but reach an efficient computation (for both attackers and defenders) preventing time-memory tradeoff (TMTO) attack on *ROMix*: it is required that the full amount of memory be allocated is needed for the specified p, regardless of whether that parallelism is actually being fully made use of or not.
To introduce the above change, the original *SMix* is split in two algorithms, *SMix1* and *SMix2*. scrypt's *SMix* is defined as $ROMix_H(B, N)$; yescrypt's *SMix1* and *SMix2* are defined splitting steps contained in *ROMix*: for further details about *SMix* functions refers to [6] (for scrypt) and [7] (for yescrypt).

4.3 Comparison between yescrypt and Argon2

The selection criteria of the Password Hashing Competition includes [3]:

- Defense against GPU/FPGA/ASIC attackers
- Defense against time-memory tradeoffs
- Defense against side-channel leaks
- Defense against cryptanalytic attacks
- Elegance and simplicity of design
- Quality of the documentation
- Quality of the reference implementation
- General soundness and simplicity of the algorithm
- Originality and innovation

The defense characteristics are reached by both schemes. yescrypt analysis highlighted some weaknesses on cache-timing side-channels; only a small initial portion of computation turned out to be cache-timing resistant.

The strength and at the same time the weakness of yescrypt lies in the fact that the author tried to solve some problematic issues concerning scrypt instead of find another way to hash passwords.
The result of this attempt was good an many problems were solved: the total compatibility with scrypt allows great flexibility and many applications of this password hashing scheme. yescrypt received indeed a special recognition for its rich feature set and easy upgrade path from scrypt.
On the other hand, this strategy increases the already high complexity of the algorithm. Since one of the judged aspect in the competition was "elegance and simplicity of design" [3], yescrypt was penalized.

The deep and thorough security analysis of Argon2, especially against TMTO, and the uses only AES as external primitive (often available as CPU instructions) was rewarded and Argon2 was the winner of the Password Hashing Competition.

References

[1] https://password-hashing.net/.

[2] https://github.com/P-H-C/phc-winner-argon2.

[3] https://password-hashing.net/report-finalists.html.

[4] Biryukov, A., and Khovratovich, D. Tradeoff cryptanalysis of memory-hard functions. *Advances in Cryptology – ASIACRYPT* (2015).

[5] Biryukov, A., Khovratovich, D., and Dinu, D. Argon2: the memory-hard function for password hashing and other applications.

[6] Percival, C. Stronger key derivation via sequential memory-hard functions.

[7] Peslyak, A. yescrypt - a password hashing competition submissions.