Argon and Argon2

Designers: Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich
University of Luxembourg, Luxembourg

alex.biryukov@uni.lu, dmitry.khovratovich@uni.lu, khovratovich@gmail.com

https://www.cryptolux.org/index.php/Password
https://github.com/khovratovich/Argon

https://github.com/khovratovich/Argon2

Version 1.1 of Argon
Version 1.1 of Argon2

6th February, 2015

Contents

1 Introduction **3**

2 Argon **5**
- 2.1 Specification . . . 5
  - 2.1.1 Input . . . 5
  - 2.1.2 SubGroups . . . 6
  - 2.1.3 ShuffleSlices . . . 7
- 2.2 Recommended parameters . . . 8
- 2.3 Security claims . . . 8
- 2.4 Features . . . 9
  - 2.4.1 Main features . . . 9
  - 2.4.2 Server relief . . . 10
  - 2.4.3 Client-independent update . . . 10
  - 2.4.4 Possible future extensions . . . 10
- 2.5 Security analysis . . . 10
  - 2.5.1 Avalanche properties . . . 10
  - 2.5.2 Invariants . . . 11
  - 2.5.3 Collision and preimage attacks . . . 11
  - 2.5.4 Tradeoff attacks . . . 11
- 2.6 Design rationale . . . 14
  - 2.6.1 SubGroups . . . 14
  - 2.6.2 ShuffleSlices . . . 16
  - 2.6.3 Permutation $\mathcal{F}$ . . . 16
  - 2.6.4 No weakness, no patents . . . 16
- 2.7 Tweaks . . . 17
- 2.8 Efficiency analysis . . . 17
  - 2.8.1 Modern x86/x64 architecture . . . 17
  - 2.8.2 Older CPU . . . 17
  - 2.8.3 Other architectures . . . 17

3 Argon2 **19**
- 3.1 Specification . . . 19
  - 3.1.1 Inputs . . . 19
  - 3.1.2 Operation . . . 20
  - 3.1.3 Indexing . . . 21
  - 3.1.4 Compression function G . . . 22
- 3.2 Features . . . 22
  - 3.2.1 Available features . . . 23
  - 3.2.2 Server relief . . . 23
  - 3.2.3 Client-independent update . . . 24
  - 3.2.4 Possible future extensions . . . 24
- 3.3 Design rationale . . . 24
  - 3.3.1 Cost estimation . . . 24
  - 3.3.2 Mode of operation . . . 25
  - 3.3.3 Implementing parallelism . . . 26
  - 3.3.4 Compression function design . . . 26
  - 3.3.5 User-controlled parameters . . . 28

3.4 Security analysis 28
3.4.1 Security of single-pass schemes 28
3.4.2 Ranking tradeoff attack 29
3.4.3 Multi-pass schemes 30
3.4.4 Security of Argon2 to generic attacks 30
3.4.5 Security of Argon2 to tradeoff attacks 31
3.5 Performance 32
3.5.1 x86 architecture 32
3.6 Applications 32
3.7 Other details 33
3.7.1 Permutation $\mathcal{P}$ 33
3.7.2 Proof of Lemma 1 33

4 Change log 35
4.1 v1.1 of Argon2 – 6th February, 2015 35
4.2 v2 and Argon2 – 31th January, 2015 35
4.3 v1 – 15th April, 2014 35
4.4 v1 – 8th April, 2014 35

Chapter 1

Introduction

Passwords remain the primary form of authentication on various web-services. Passwords are usually stored in a hashed form in a server's database. These databases are quite often captured by the adversaries, who then apply dictionary attacks since passwords tend to have low entropy. Protocol designers use a number of tricks to mitigate these issues. Starting from the late 70's, a password is hashed together with a random *salt* value to prevent detection of identical passwords across different users and services. The hash function computations, which became faster and faster due to Moore's law have been called multiple times to increase the cost of password trial for the attacker.

In the meanwhile, the password crackers migrated to new architectures, such as FPGAs, multiple-core GPUs and dedicated ASIC modules, where the amortized cost of a multiple-iterated hash function is much lower. It was quickly noted that these new environments are great when the computation is almost memoryless, but they experience difficulties when operating on a large amount of memory. The defenders responded by designing *memory-hard* functions, which require a large amount of memory to be computed. The password hashing scheme scrypt [23] is an instance of such function.

Password hashing schemes also have other applications. They can be used for key derivation from low-entropy sources. Memory-hard schemes are also welcome in cryptocurrency designs [4] if a creator wants to demotivate the use of GPUs and ASICs for mining and promote the use of standard desktops.

Problems of existing schemes. A design of a memory-hard function proved to be a tough problem. Since early 80's it has been known that many cryptographic problems that seemingly require large memory actually allow for a time-memory tradeoff [18], where the adversary can trade memory for time and do his job on fast hardware with low memory. In application to password-hashing schemes, this means that the password crackers can still be implemented on a dedicated hardware even though at some additional cost. The scrypt function, for example, allows for a simple tradeoff where the time-area product remains almost constant.

Another problem with the existing schemes is their complexity. The same scrypt calls a stack of subprocedures, whose design rationale has not been fully motivated (e.g, scrypt calls SMix, which calls ROMix, which calls BlockMix, which calls Salsa20/8 etc.). It is hard to analyze and, moreover, hard to achieve confidence. Finally, it is not flexible in separating time and memory costs. At the same time, the story of cryptographic competitions [2, 25, 3] has demonstrated that the most secure designs come with simplicity, where every element is well motivated and a cryptanalyst has as few entry points as possible.

The ongoing Password Hashing Competition also highlighted the following problems:

- Should the memory addressing (indexing functions) be input-independent or input-dependent, or hybrid? The first type of schemes, where the memory read location are known in advance, is immediately vulnerable to time-space tradeoff attacks, since an adversary can precompute the missing block by the time it is needed [10]. In turn, the input-dependent schemes are vulnerable to side-channel attacks [24], as the timing information allows for much faster password search.

- Is it better to fill more memory but suffer from time-space tradeoffs, or make more passes over the memory to be more robust? This question was quite difficult to answer due to absence of generic tradeoff tools, which would analyze the security against tradeoff attacks, and the absence of unified metric to measure adversary's costs.

- How should the input-independent addresses be computed? Several seemingly secure options have been attacked [10].

- How large a single memory block should be? Reading smaller random-placed blocks is slower (in cycles per byte) due to the spacial locality principle of the CPU cache. In turn, larger blocks are difficult to process due to the limited number of long registers.
- If the block is large, how to choose the internal compression function? Should it be cryptographically secure or more lightweight, providing only basic mixing of the inputs? Many candidates simply proposed an iterative construction and argued against cryptographically strong transformations.
- Which operations should be used by the compression function in the goal of maximizing adversary's costs with fixed desktop performance? Should we use recent AES instructions, ARX primitives, integer and floating-point operations?
- How to exploit multiple cores of modern CPUs, when they are available? Parallelizing calls to the hashing function without any interaction is subject to simple tradeoff attacks.

Our solutions. We offer two new hashing scheme called Argon and Argon2. Argon is our original submission to PHC. It is a multipurpose hash function, that is optimized for highest resilience against tradeoff attacks, so that any, even small memory reduction would lead to significant time and computational penalties. Argon can be used for password hashing, key derivation, or any other memory-hard computation (e.g., for cryptocurrencies).

Argon2 summarizes the state of the art in the design of memory-hard functions. It is a streamlined and simple design. It aims at the highest memory filling rate and effective use of multiple computing units, while still providing defense against tradeoff attacks. Argon2 is optimized for the x86 architecture and exploits the cache and memory organization of the recent Intel and AMD processors. Argon2 has two variants: Argon2d and Argon2i. Argon2d is faster and uses data-depending memory access, which makes it suitable for cryptocurrencies and applications with no threats from side-channel timing attacks. Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks.

We recommend Argon for the applications that aim for the highest tradeoff resilience and want to guarantee prohibitive time and computational penalties on any memory-reducing implementation. According to our cryptanalytic algorithms, an attempt to use half of the requested memory (for instance, 64 MB instead of 128 MB) results in the speed penalty factor of 140 and in the penalty 2^{18}. The penalty grows exponentially as the available memory decreases, which effectively prohibits the adversary to use any smaller amount of memory. Such high computational penalties are a unique feature of Argon.

We recommend Argon2 for the applications that aim for high performance. Both versions of Argon2 allow to fill 1 GB of RAM in a fraction of second, and smaller amounts even faster. It scales easily to the arbitrary number of parallel computing units. Its design is also optimized for clarity to ease analysis and implementation.

Chapter 2

Argon

2.1 Specification

Argon is a password hashing scheme, which implements a memory-hard function with memory and time requirements as a parameter. It is designed so that any reduction of available memory imposes a significant penalty on the running time of the algorithm.

Argon is a parametrized scheme with two main parameters:

- Memory size m (`m_cost`). Argon can occupy any number of kilobytes of memory, and its performance is a linear function of the memory size.
- Number of iterations R (`t_cost`). It also linearly affects the time needed to compute the output (tag) value.

Note that the overall time is affected by both `t_cost` and `m_cost`.

Argon employs a t-byte permutation. In this submission it is the 5-round AES-128 with fixed key, so $t = 16$.

Argon employs a cryptographic hash function H. It is Blake2b [6].

2.1.1 Input

The hash function Π takes the following inputs:

- Password P of byte length p, $0 \leq p < 2^{32}$.
- Salt S of byte length s, $8 \leq s < 2^{32}$.
- Memory size m in kilobytes, $1 \leq m < 2^{24}$.
- Iteration number R, $2 \leq R < 2^{32}$.
- Secret value K of byte length k, $0 \leq k \leq 16$;
- Associated data X of byte length x, $0 \leq x < 2^{32}$;
- Degree of parallelism d, $1 \leq d \leq 32$.
- Tag length τ, $4 \leq \tau \leq 2^{32} - 1$.

and outputs a tag of length τ:

$$\Pi : \ (P, S, m, R, K, X, \tau) \to \text{Tag} \in \{0,1\}^{8\tau}.$$

The authentication server places the following string alongside the user's credentials:

$$\texttt{Storage} \leftarrow \tau \,||\, m \,||\, R \,||\, S \,||\, X \,||\, \text{Tag},$$

with the secret K stored in another place.

Initial hashing phase. The values p, s, k, R, m, x, τ are known at the start of hashing and are encoded as 4-byte strings using the little-endian convention. Together with P, S, K, X they are fed into a cryptographic hash function H that produces a 256-bit output I:

$$I = H(p, P, s, S, k, K, x, X, d, m, R, \tau).$$

The input I is partitioned into 4-byte blocks I_0, I_1, I_2, I_3. Then these blocks are repeatedly used to construct $n = 1024m/t$ blocks of t bytes each with a counter:

$$A_i = I_{i \pmod 4} \,||\, C_i,$$

where C_i is the 8-byte encoding of the value i.

The scheme then operates on blocks A_i. It is convenient to view them as a matrix

$$A = \begin{pmatrix} A_0 & A_{n/32} & \cdots & A_{n-n/32} \\ A_1 & A_{n/32+1} & \cdots & A_{n-n/32+1} \\ \cdots & & & \\ A_{n/32-1} & A_{n/16-1} & \cdots & A_{n-1} \end{pmatrix}$$

Initial round. In the initial round the permutation $\mathcal{F}$ is applied to every block A_i. The transformation $\mathcal{F}$ is the reduced 5-round AES-128 with the fixed key

$K_0 =$ `(0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f)`.

MixColumns transformation is present in the last round.

Then the transformation SubGroups is applied. It operates on the rows of A, which are called *groups*, and is defined in Section 2.1.2.

Main rounds. In the main rounds the transformations ShuffleSlices and SubGroups are applied alternatively R times. ShuffleSlices operates on columns, which we call *slices* of A, and is defined in Section 2.1.3.

Finalization round. The first $n/2$ blocks of the memory are XORed into a 16-byte block B_0, and the second $n/2$ blocks are XORed into B_1. Then the hash function H is applied to $B_0||B_1$. We iterate H until τ bytes are generated, denoting the entire hash function by H':

$$\begin{aligned} B_0 &\leftarrow \bigoplus_{0 \le i < n/2} A_i; \\ B_1 &\leftarrow \bigoplus_{n/2 \le i < n} A_i; \\ B_2||B_3 &\leftarrow H(B_0||B_1); \\ B_4||B_5 &\leftarrow H(B_2||B_3); \\ &\cdots \\ \text{Tag} &\leftarrow B_2||B_4||B_6 \ldots = H'(B_0||B_1). \end{aligned}$$

The pseudocode of the entire algorithm given in Algorithm 1.

2.1.2 SubGroups

The SubGroups transformation applies the smaller operation Mix to each group of size 32. Mix takes t-byte blocks $A_0, A_1, \ldots, A_{31}$ as input and processes them as follows:

1. 16 new variables $X_0, X_1, \ldots, X_{15}$ are computed as specific linear combinations of $A_0, A_1, \ldots, A_{31}$.
2. New values to A_i are assigned:
$$A_i \leftarrow \mathcal{F}(A_i \oplus \mathcal{F}(X_{\lfloor i/2 \rfloor})).$$

```
**Input**: $t$-byte states $A_0, A_1, \dots, A_{n-1}$
$s \leftarrow n/32$
Initial Round:
**for** $0 \le j < n$ **do**
    $A_i \leftarrow \mathcal{F}(A_i)$
**end**
SubGroups:
**for** $0 \le j < n/32$ **do**
    $\mathrm{Mix}(A_j, A_{j+n/32}, \dots, A_{31n/32+j})$
**end**
**for** $1 \le i \le R$ **do**
    ShuffleSlices:
    **for** $0 \le j < d$ **do**
        $\mathrm{Shuffle}(A_{jn/d}, A_{jn/d+1}, \dots, A_{(j+1)n/d-1})$;
    **end**
    SubGroups:
    **for** $0 \le j < n/32$ **do**
        $\mathrm{Mix}(A_j, A_{j+n/32}, \dots, A_{31n/32+j})$
    **end**
**end**
Finalization:
$B_0 \leftarrow \bigoplus_{0 \le i < n/2} A_i$;
$B_1 \leftarrow \bigoplus_{n/2 \le i < n} A_i$;
$\mathrm{Tag} \leftarrow H'(B_0 || B_1)$.
```

Algorithm 1: Pseudocode of the Argon hashing scheme with R iterations

The X_i are computed as linear functions of A_i (motivation given in Section 2.6.1):

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \cdots \\ X_{15} \end{pmatrix} = L \cdot \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ \cdots \\ A_{31} \end{pmatrix},$$

where the addition is treated as bitwise XOR, and L is a $(0,1)$-matrix of size 16×32:

$$L = \begin{pmatrix} 00010001000100010001000100010001 \\ 01010000010100000101000001010000 \\ 10101010000000001010101000000000 \\ 01010101010101010000000000000000 \\ 00000011000000110000001100000011 \\ 00000000001100110000000000110011 \\ 00000000000000001100110011001100 \\ 00000000000011110000000000001111 \\ 00001111000011110000000000000000 \\ 00000000000000001111111100000000 \\ 01000100010001000100010001000100 \\ 00100010001000100010001000100010 \\ 00001111000000000000111100000000 \\ 00000000111100000000000011110000 \\ 11110000111100000000000000000000 \\ 10001000100010001000100010001000 \end{pmatrix} \qquad (2.1)$$

2.1.3 ShuffleSlices

The ShuffleSlices transformation operates on *slices*. For d slices, blocks $A_0, A_1, \dots, A_{n/d-1}$ form the first slice, then $A_{n/d}, A_{n/d+1}, \dots, A_{2n/d-1}$ are the second slice, etc. Each slice is shuffled independently of the others with the same algorithm Shuffle. Let us denote the 64-bit subwords of state B by B^0, B^1. The

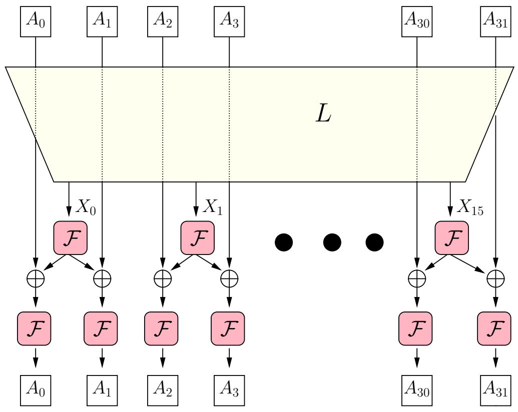

Figure 2.1: Mix transformation.

Shuffle algorithm is derived from the RC4 permutation algorithm and operates according to Algorithm 2.

```
Input: Slice $\langle B_0, B_1, \dots, B_{s-1}\rangle$.
$j \leftarrow s - 1$
for $0 \le i \le s - 1$ do
    $j \leftarrow (B_j^0 + B_i^0) \pmod{s}$;
    Swap$(B_i, B_j)$;
end
```

Algorithm 2: The Shuffle algorithm.

2.2 Recommended parameters

In this section we provide minimal values of `t_cost` as a function of `m_cost` that render Argon secure as a hash function. These values are based on the analysis made in Chapter 2.5. Future cryptanalysis may reduce the minimal `t_cost` values.

`m_cost` Memory used	1 1 KB	10 10 KB	100 100 KB	10^3 1 MB	10^4 10 MB	10^5 100 MB	10^6 1 GB
Minimal `t_cost`	3	3	3	3	3	3	3

Table 2.1: Minimally secure time and memory parameters.

We recommend the following procedure to choose `m_cost` and `t_cost`:

1. Figure out maximum amount of memory M_{max} that can be used by the authentication application.
2. Figure out the maximum allowed time T_{max} that can be spent by the authentication application.
3. Benchmark Argon on the resulting `m_cost` and minimal secure `t_cost`.
4. If the resulting time is smaller than T_{max}, then increase `t_cost` accordingly; otherwise decrease `m_cost` until the time spent is appropriate.

The reference implementation allows all positive values of `t_cost` for testing purposes.

2.3 Security claims

Collision and forgery resistance. We claim that Argon with an m-bit tag and recommended cost parameters provides $m/2$ bits of security against collision attacks, m bits of security against forgery attacks.

Time-memory tradeoff resistance. We claim that Argon imposes a significant computational penalty on the adversary even if he uses as much as half of normal memory amount. Our best tradeoff algorithms achieve the following penalties for $R = 3$ (Chapter 2.5):

Attacker's fraction \ Regular memory	128 KB	1 MB	16 MB	128 MB	1 GB
$\frac{1}{2}$	91	112	139	160	180
$\frac{1}{4}$	164	314	2^{18}	2^{26}	2^{34}
$\frac{1}{8}$	6085	2^{20}	2^{31}	2^{36}	2^{47}

Side-channel attacks. The optimized implementation of Argon on the recent Intel/AMD processors is supposed to use the AES instructions, which essentially prohibits the side-channel attacks on the AES core. The ShuffleSlices transformation is data-dependent and hence is potentially dependent on the cache behaviour. To partially thwart the timing attacks, our optimized implementation selects slices in the random order.

2.4 Features

We offer Argon as a password hashing scheme for architectures equipped with recent Intel/AMD processors that support dedicated AES instructions. The key feature of Argon is a large computational penalty imposed on an adversary who wants to save even a small fraction of memory. We aimed to make the structure of Argon as simple as possible, using only XORs, swaps, and (reduced) AES calls when processing the internal state.

2.4.1 Main features

Now we provide an extensive list of features of Argon.

Design rationality	Argon follows a design strategy, which proved well in designing cryptographic hash functions and block ciphers. After the memory is filled, the internal state undergoes a sequence of identical rounds. The rounds alternate nonlinear transformations that operate row-wise and provide confusion with block permutations that operate columnwise and provide diffusion. The design of nonlinear transformations aims to maximize internal diffusion and defeat time-memory tradeoffs. The data-dependent permutation part operates similarly to the well-studied RC4 state permutation.
Tradeoff defense	Thanks to fast diffusion and data-dependent permutation layers, Argon provides strong defense against memory-saving adversaries. Even the memory reduction by a factor of 2 results in almost prohibitive computational penalty. Our best tradeoff attacks on the fastest set of parameters result in the penalty factor of 50 when 1/2 of memory is used, and the factor of 150 when 1/4 of memory for is used.
Scalability	Argon is scalable both in time and memory dimensions. Both parameters can be changed independently provided that a certain amount of time is always needed to fill the memory.
Uniformity	Argon treats all the memory blocks equally, and they undergo almost the same number of transformations and permutations. The scheme operates identically for all state sizes without any artifacts when the size is not the power of two.
Parallelism	Argon may use up to 32 threads/cores in parallel by delegating them slice permutations and group transformations.
Server relief	Argon allows the server to carry out the majority of computational burden on the client in case of denial-of-service attacks or other situations. The server ultimately receives a short intermediate value, which undergoes a preimage-resistant function to produce the final tag.

Client-independent updates	Argon offers a mechanism to update the stored tag with an increased time and memory cost parameters. The server stores the new tag and both parameter sets. When a client logs into the server next time, Argon is called as many times as need using the intermediate result as an input to the next call.
Possible extensions	Argon is open to future extensions and tweaks that would allow to extend its capabilities. Some possible extensions are listed in Section 2.4.4.
CPU-friendly	Implementation of Argon benefits from modern CPUs that are equipped with dedicated AES instructions [16] and penalizes any attacker that uses a different architecture. Argon extensively uses memory in the random access pattern, which results in a large latency for GPUs and other memory-unfriendly architectures.
Secret input support	Argon natively supports secret input, which can be called *key* or *pepper* [14].

2.4.2 Server relief

The server relief feature allows the server to delegate the most expensive part of the hashing to the client. This can be done as follows (suppose that K and X are not used):

1. The server communicates S, m, R, d, τ to the client.
2. The client performs all computations up to the value of (B_0, B_1);
3. The client communicates (B_0, B_1) to the server;
4. The server computes Tag and stores it together with S, m, d, R.

The tag computation function is preimage-resistant, as it is an iteration of a cryptographic hash function. Therefore, leaking the tag value would not allow the adversary to recover the actual password nor the value of B by other means than exhaustive search.

2.4.3 Client-independent update

It is possible to increase the time and memory costs on the server side without requiring the client to re-enter the original password. We just compute

$$\mathrm{Tag}_{new} = \Pi(\mathrm{Tag}_{old}, S, m_{new}, R_{new}, \tau_{new}),$$

replace Tag_{old} with Tag_{new} in the hash database and add the new values of m and R to the entry.

2.4.4 Possible future extensions

Argon can be rather easily tuned to support the following extensions, which are not used now for design clarity or performance issues:

Support for other permutations	A larger permutation can be easily plugged into the scheme and would not affect its security. For example, hardware-friendly permutations of Keccak [9], Spongent [11], or Quark [5] may be used. Also, any cryptographically strong 256-bit hash function may replace Blake2b in the initialization and finalization phases.

2.5 Security analysis

2.5.1 Avalanche properties

Due to the use of a cryptographic hash function, a difference in any input bit is likely to spread over all memory blocks, thus making the avalanche immediate. This has been added as a tweak to the original submission.

2.5.2 Invariants

The SubGroups and ShuffleSlices transformations have several symmetric properties:

- Mix transforms a group of identical blocks (*flat group*) to a group of identical blocks, and vice versa.
- ShuffleSlices transforms a state of flat groups to a state of flat groups.

These properties are mitigated by the use of a counter in the initialization phase. The flat group condition is a 32-collision on 128 bits, which requires $2^{\frac{31\cdot 128}{32}} = 2^{124}$ controlled input blocks, whereas the counter leaves only 96 bits to the attacker.

2.5.3 Collision and preimage attacks

Since the 4-round AES has the expected differential probability bounded by 2^{-113} [21], we do not expect any differential attack on our scheme, which uses the 5-round AES. The same reasoning applies for linear cryptanalysis. Given the huge internal state compared to regular hash functions, we do not expect any other collision or preimage attack to apply on our scheme.

2.5.4 Tradeoff attacks

In this section we list several possible attacks on Argon, where an attacker aims to compute the hash value using less memory. To compute the actual penalty, we first have to compute the number of operations performed in each layer:

- Initial Round: n calls to $\mathcal{F}$, n memory reads and n memory writes (128-bit values);
- ShuffleSlices: $2n$ memory reads, $2n$ memory writes.
- SubGroups: $1.5n$ calls to $\mathcal{F}$, $5n$ XORs, n memory reads and n memory writes (128-bit values);
- Finalization: n XORs and n memory reads.

Therefore, R-layer Argon that uses $16n$ bytes of memory performs $(1.5R + 2.5)n$ calls to $\mathcal{F}$, $(5R + 6)n$ XORs, and $(6R + 5)n$ memory accesses.

The total amount of memory, which we denote by M, is also equal to $16n$ bytes.

Let us denote by Y^0 the input state after the initial round (before the first call of SubGroups), by X^l the input to ShuffleSlices in round l, and by Y^l the input to SubGroups in round l. The input to the finalization round is denoted by X^{R+1}.

In the further analysis we use the notation

- $T_{\mathcal{F}}(A)$, where A is either X^j or Y^j, denotes the amortized number of $\mathcal{F}$-calls needed to compute a single block of A according to the adversary's memory use.

In the regular computation we have

$$
\begin{aligned}
T_{\mathcal{F}}(Y^0) &= 1;\\
T_{\mathcal{F}}(X^l) &= 1.5 + T_{\mathcal{F}}(Y^{l-1});\\
T_{\mathcal{F}}(Y^l) &= T_{\mathcal{F}}(X^l);\\
T_{\mathcal{F}}(\text{Tag}) &= nT_{\mathcal{F}}(Y^R).
\end{aligned}
$$

Storing block permutations

Suppose that an attacker stores the permutations produced by the ShuffleSlices transformation, which requires $\frac{\lg n - 5}{128}M$ memory per round (Section 2.6.2). Let us denote by σ_i the permutation over X^i realized by ShuffleSlices. The attack algorithm is as follows:

- Sequentially compute $\sigma_1, \sigma_2, \ldots, \sigma_R$. To compute σ_l, use the knowledge of $\sigma_1, \sigma_2, \ldots, \sigma_{l-1}$.
- For each group in Y^R, compute its output blocks in X^{R+1} and accumulate their XOR.
- Compute the tag.

The permutation σ_l is computed slicewise by first storing $n/32$ slice elements and then computing a permutation over them. This requires n accesses to blocks from X^l. In order to get a block from X^l one needs to evaluate the corresponding $\mathcal{F}(X_i)$ in the Mix transformation, which in turn takes 8 unknown blocks from Y^{l-1}. Therefore,

$$T_{\mathcal{F}}(X^l) = 8T_{\mathcal{F}}(Y^{l-1}) + 2; \tag{2.2}$$

Given the stored σ_i, we have

$$T_{\mathcal{F}}(Y^i) = T_{\mathcal{F}}(X^i),$$

but As a result,

$$T_{\mathcal{F}}(X^l) \approx 1.25 \cdot 8^l,$$

and

$$T_{\mathcal{F}}(\sigma_l) = 1.25n \cdot 8^l.$$

The attacker computes X^{R+1} groupwise, so the cost of computing a single group of X^{R+1} is $32T_{\mathcal{F}}(X^R)+$ 48. Therefore,

$$T_{\mathcal{F}}(X^{R+1}) = T_{\mathcal{F}}(X^R) + 1.5;$$

$$\frac{T_{\mathcal{F}}(\text{Tag})}{n} \approx 1.25 \cdot 8^R + 1.5 + \sum_{i=1}^{R} T_{\mathcal{F}}(\sigma_l) = 1.25 \cdot 8^R + 1.5 + \sum_{i=1}^{R} 1.25 \cdot 8^i \approx 2.6 \cdot 8^R.$$

and the penalty is

$$\mathcal{P} = \frac{2.6 \cdot 8^R}{1.5R + 2.5} \approx 190 \text{ for } R = 3 \quad \text{and } 1253 \text{ for } R = 4. \tag{2.3}$$

This value should be compared to the fraction of memory used:

$$\frac{M_{reduced}}{M} = R\frac{\lg n - 5}{128} = R\frac{\lg M - 9}{128},$$

where M is counted in bytes.

Some examples for various n are given in Table 2.2.

Memory total n	64 KB 2^{12}	1 MB 2^{16}	16 MB 2^{20}	256 MB 2^{24}	1 GB 2^{26}
$R = 3$					
Memory used	10 KB	250 KB	5 MB	114 MB	500 MB
Penalty	190				
$R = 4$					
Memory used	13 KB	330 KB	6 MB	150 MB	650 MB
Penalty	1253				

Table 2.2: Computational penalty when storing permutations only.

Storing all X_i and block permutations

Suppose that an attacker stores all the permutations and as many values $\mathcal{F}(X_i)$ as possible in each round. Storing the entire level would cost $M/2$ memory. Let the attacker spend αM memory on $\mathcal{F}(X_i)$, $\alpha \leq 0.5$, in the initial round. The attack algorithm is as follows:

- Compute 2α fraction of $\mathcal{F}(X_i)$ in the initial round and store them.
- Sequentially compute $\sigma_1, \sigma_2, \dots, \sigma_R$. To compute σ_l, use the knowledge of $\mathcal{F}(X_i)$ and $\sigma_1, \sigma_2, \dots, \sigma_{l-1}$.
- For each group in Y^R, compute its output blocks in X^{R+1} and accumulate their XOR.
- Compute the tag.

Then for blocks of X^1 that come from stored groups, we get

$$T'_{\mathcal{F}}(X^1) = 1 + T_{\mathcal{F}}(Y^0);$$

and for the others

$$T''_{\mathcal{F}}(X^1) = 8T_{\mathcal{F}}(Y^0) + 2;$$

On average we have

$$T_{\mathcal{F}}(X^1) = (8 - 14\alpha)T_{\mathcal{F}}(Y^0) + 2 - 2\alpha = 10 - 16\alpha;$$

For the other rounds Equation (2.2) holds. Therefore, we have

$$\begin{aligned} T_{\mathcal{F}}(\sigma_l) &= (10 - 16\alpha)n \cdot 8^{l-1}; \\ T_{\mathcal{F}}(X^R) &\approx (10 - 16\alpha)n \cdot 8^{R-1}, \\ \frac{T_{\mathcal{F}}(\text{Tag})}{n} &\approx \cdot(20 - 32\alpha)8^{R-1}, \end{aligned}$$

and the penalty is

$$\mathcal{P} = \frac{(20 - 32\alpha)8^{R-1}}{1.5R + 2.5}. \tag{2.4}$$

For $R = 3$ we have the penalties depending on α in Table 2.3.

α	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{8}$
Penalty	36	86	110	146

Table 2.3: Computational penalty on an adversary who stores some $\mathcal{F}(X_i)$ in αM memory and additionally all the permutations.

Summary

The optimal strategy for an adversary that has βM memory, $\beta < 1$, is as follows:

- If

  $$\beta \le R\frac{\lg M - 9}{128},$$

  then the adversary is recommended to spend the memory entirely to store the permutations produced by ShuffleSlices. For $\beta = l\frac{\lg M-9}{128}$, $0 \le l \le R$, he gets the penalty about (Eq. (2.3))

  $$\mathcal{P}(l) = \frac{2.6 \cdot 8^l (n/32)^{R-l}}{1.5R + 2.5},$$

  where we assume that to compute a permutation online he has to make $n/32$ queries to the previous layer.

- If

  $$\beta > R\frac{\lg M - 9}{128},$$

  then the adversary stores all the permutations in $RM\frac{\lg M-9}{128}$ memory, and spend the rest $\alpha M = (\beta - R\frac{\lg M-9}{128})$ on storing $\mathcal{F}(X_i)$ in SubGroups. If $\alpha \le 1/2$, we have the penalty (Eq. (2.4)).

  $$\mathcal{P}(\alpha) = \frac{(20 - 32\alpha)8^{R-1}}{1.5R + 2.5}$$

  Some examples for different α and $R = 3$ are given in Table 2.3.

2.6 Design rationale

2.6.1 SubGroups

The SubGroups transformation should have the following properties:

- Every output block must be a nonlinear function of input blocks;
- It should be memory-hard in the sense that a certain number of blocks must be stored.

We decided to have the number of input blocks be a degree of 2, i.e. 2^k. We compute r linear combinations $X_1, X_2, \ldots, X_r$ of them:

$$X_i = \bigoplus \lambda_{i,j} A_j,\ \lambda_{i,j} \in \{0,1\}$$

and compute

$$A_i \leftarrow \mathcal{F}(A_i \oplus \mathcal{F}(X_{g(i)})).$$

Here function g maps $\{1,2,\ldots,2^k\}$ to $\{1,2,\ldots,r\}$. The function should be maximally balanced.

The values $X_1, X_2, \ldots, X_r$ should not be computable from each other without knowing a certain number of input blocks. As a result, to compute any output block A_i, an adversary either must have stored $\mathcal{F}(X_{g(i)})$ or has to recompute it online. The latter case determines the computational penalty on a reduced-memory adversary. The actual penalty results from the following parameters:

- The minimal weight w (*diffusion degree*) of vectors $\overline{\lambda_i} = (\lambda_{i,1}, \lambda_{i,2}, \ldots, \lambda_{i,2^k})$ or their linear combinations;
- The proportion $\frac{r}{2^k}$ (*group rate*) of middle $\mathcal{F}$ calls compared to the total number of $\mathcal{F}$ calls in the group.

Therefore, an adversary, who needs the output A_i either stores r block values per group and needs the input A_i, or stores nothing and needs w input blocks to compute one output block. To impose the maximum penalty on the adversary, we should maximize both r and w, but take into account that this means a small penalty to the defender as well.

Choice of linear combinations

The requirement of $\{X_i\}$ not be computable from each other translates into the following. Let $\overline{\lambda_i}$ be a vector of values of function f_i of k variables $y_1, y_2, \ldots, y_k$. These functions form a linear code Λ, and the minimal weight of a codeword is w.

We consider Reed-Muller codes $RM(d,k)$, which are value vectors of functions of k boolean variables up to degree d. The minimal codeword weight is 2^{k-d}.

For instance, $d = 1$ yields $k+1$ linearly independent codewords, which are value vectors of functions

$$f_1 = y_1,\ f_2 = y_2, \ldots, f_k = y_k,\ f_{k+1} = 1,$$

which all have weight 2^{k-1} and generate the following formulas for X_i:

$$\begin{aligned} X_1 &= A_2 \oplus A_4 \oplus \ldots A_{2^k}; \\ X_2 &= A_3 \oplus A_4 \oplus A_7 \oplus A_8 \ldots \oplus A_{2^k}; \\ \ldots \\ X_{k+1} &= A_1 \oplus A_2 \oplus A_3 \oplus \ldots \oplus A_{2^k}. \end{aligned}$$

For $d = 2$ we have $\binom{k}{2} + k + 1$ linearly independent codewords. They may not have the same weight, but it is easy to find those that all have weight 2^{k-2}: there are $k(2k-1)$ of them.

Diffusion properties. The more difficult problem is to select a set Λ of codewords such that each index (i.e. each A_i) is covered by maximal number of codewords. For $k = 5$ we have at maximum 16 linearly independent codewords of weight 8 and length 32 each. Since the sum of weights is 128, each index i may potentially be non-zero in exactly 4 codewords. However, the bitwise sum of such codewords would be the all-zero vector, so such codewords would be linearly dependent. For linearly independent codewords such minimal number is 3, and one of possible solution[1], which we use, is given in Table 2.4. It results in matrix L, given in Equation (2.1).

[1]The reference document of Argon v0 contained an incorrect set of codewords, which was the result of a bug in a codeword generation procedure. The current matrix has rank 16, as verified by the symbolic computation system SAGE.

$f_0 = y_1 y_2 :$	$(0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1)$ $A_3 \oplus A_7 \oplus A_{11} \oplus A_{15} \oplus A_{19} \oplus A_{23} \oplus A_{27} \oplus A_{31}$
$f_1 = y_1(y_3+1) :$	$(0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0)$ $A_1 \oplus A_3 \oplus A_9 \oplus A_{11} \oplus A_{17} \oplus A_{19} \oplus A_{25} \oplus A_{27}$
$f_2 = (y_1+1)(y_4+1) :$	$(1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0)$ $A_0 \oplus A_2 \oplus A_4 \oplus A_6 \oplus A_{16} \oplus A_{18} \oplus A_{20} \oplus A_{22}$
$f_3 = y_1 y_5 :$	$(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_1 \oplus A_3 \oplus A_5 \oplus A_7 \oplus A_9 \oplus A_{11} \oplus A_{13} \oplus A_{15}$
$f_4 = y_2 y_3 :$	$(0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1)$ $A_6 \oplus A_7 \oplus A_{14} \oplus A_{15} \oplus A_{22} \oplus A_{23} \oplus A_{30} \oplus A_{31}$
$f_5 = y_2 y_4 :$	$(0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1)$ $A_{10} \oplus A_{11} \oplus A_{14} \oplus A_{15} \oplus A_{26} \oplus A_{27} \oplus A_{30} \oplus A_{31}$
$f_6 = (y_2+1)y_5 :$	$(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0)$ $A_{16} \oplus A_{17} \oplus A_{20} \oplus A_{21} \oplus A_{24} \oplus A_{25} \oplus A_{28} \oplus A_{29}$
$f_7 = y_3 y_4 :$	$(0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1)$ $A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15} \oplus A_{28} \oplus A_{29} \oplus A_{30} \oplus A_{31}$
$f_8 = y_3 y_5 :$	$(0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_4 \oplus A_5 \oplus A_6 \oplus A_7 \oplus A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15}$
$f_9 = (y_4+1)y_5 :$	$(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0)$ $A_{16} \oplus A_{17} \oplus A_{18} \oplus A_{19} \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23}$
$f_{10} = y_1(y_2+1) :$	$(0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0)$ $A_1 \oplus A_5 \oplus A_9 \oplus A_{13} \oplus A_{17} \oplus A_{21} \oplus A_{25} \oplus A_{29}$
$f_{11} = (y_1+1)y_2 :$	$(0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0)$ $A_2 \oplus A_6 \oplus A_{10} \oplus A_{14} \oplus A_{18} \oplus A_{22} \oplus A_{26} \oplus A_{30}$
$f_{12} = y_3(y_4+1) :$	$(0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0)$ $A_4 \oplus A_5 \oplus A_6 \oplus A_7 \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23}$
$f_{13} = (y_3+1)y_4 :$	$(0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0)$ $A_8 \oplus A_9 \oplus A_{10} \oplus A_{11} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{27}$
$f_{14} = (y_3+1)(y_5+1) :$	$(1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus A_8 \oplus A_9 \oplus A_{10} \oplus A_{11}$
$f_{15} = (y_1+1)(y_2+1) :$	$(1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0)$ $A_0 \oplus A_4 \oplus A_8 \oplus A_{12} \oplus A_{16} \oplus A_{20} \oplus A_{24} \oplus A_{28}$

Table 2.4: Linearly independent combinations of 32 variables given by 16 functions of degree 2.

k	2	3	4	5	6
$d = 1$					
Diffusion degree	2	4	8	16	32
Independent vectors	3	4	5	6	7
Rate	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{5}{16}$	$\frac{3}{16}$	$\frac{7}{64}$
$d = 2$					
Diffusion degree	1	2	4	8	16
Independent vectors	4	7	11	16	22
Rate	1	$\frac{7}{8}$	$\frac{11}{16}$	$\frac{1}{2}$	$\frac{11}{32}$

2.6.2 ShuffleSlices

The ShuffleSlices transformation aims to interleave blocks of distinct groups. Since the blocks in a group are mixed in the SubGroups transformation, it is sufficient to shuffle the blocks in slices independently. This property also allows to parallelize the computation. Additionally, if the slices are stored in memory sequentially, then the Shuffle transformation works very fast if the entire slice fits into the CPU cache. As there are d slices, the entire ShuffleSlices transformation may avoid cache misses even if the total memory exceeds the cache size by a factor of d.

The exact transformation was taken from the RC4 algorithm [22], whose internal state S of n integers is updated as follows:

$$\begin{aligned} i &\leftarrow i + 1; \\ j &\leftarrow j + S[i]; \\ S[i] &\leftrightarrow S[j], \end{aligned}$$

where all indices and additions are computed modulo n. However, this algorithm is parallelizable:

- Read $S[1], S[2], \ldots, S[d]$ in parallel;
- Compute $j_1, j_2, \ldots, j_d$ also in parallel (d additions can be done in $\log_2 d$ time).
- Read $S[j_1], S[j_2], \ldots, S[j_d]$ in parallel.
- Make all swaps in parallel.

This approach works until $j_k \leq d$ for some k, which is unlikely for $d \ll \sqrt{s}$. To avoid the unnecessary parallelism, we modify the j rule as

$$j \leftarrow S[i] + S[j],$$

which requires to read $S[j]$ before updating j.

It is important to note that the resulting permutation can be stored in $\frac{n(\lg n - 5)}{8}$ bytes of memory, which is the $\frac{\lg M - 9}{128}$ part of the total memory cost M, measured in bytes.

2.6.3 Permutation $\mathcal{F}$

We wanted $\mathcal{F}$ to resemble a randomly chosen permutation with no significant differential or linear properties or other properties that might simplify tradeoff attacks. Since we expect our scheme to run on modern CPUs, the AES cipher or its derivatives is a reasonable choice in terms of performance: it is known that the 10-round AES in the Counter mode runs at 0.6 cycles per byte on Haswell CPUs [16]. We considered having a wider permutation, but our optimized implementations of the ShuffleSlices that operates on wider blocks were only marginally faster, whereas the SubGroups transformation becomes far more complicated, and in some cases even is a bottleneck.

Therefore, AES itself is a natural choice. Clearly, all 10 rounds are not needed whereas the 4-round version is known to hide all the differential properties [21]. We decided to take 5 rounds.

2.6.4 No weakness, no patents

We have not hidden any weaknesses in this scheme.

We are aware of no known patents, patent applications, planned patent applications, and other intellectual-property constraints relevant to use of the scheme. If this information changes, we will promptly announce these changes on the mailing list.

2.7 Tweaks

The current version of Argon tweaks the previous version as follows:

- Number of slices in ShuffleSlices is a tunable parameter to fix the degree of parallelism available to the adversary.
- Number of slices and the tag length are now part of the input to further prevent the adversary from amortizing exhaustive search.
- Associated data X can now be processed as part of the input. It can be an arbitrary string, containing application context, user data, or any other parameters that shall be binded to the hash value.
- All the inputs are hashed with a cryptographic hash function (Blake2b) in order to extract the entropy from the password and the salt as soon as possible and put it into a short string I.
- Due to the use of a hash function, password and salt may have arbitrary (up to 2^{32} bytes) length.
- The string I is partitioned in only 4 parts, which fill the first 8 bytes of each 16-byte memory block, whereas the rest is taken by the counter. This has been done for simplicity.
- The Shuffle algorithm now employs a new formula to compute the value of the counter j, which should not allow to parallelize the algorithm, as it is the case for RC4.
- To produce the tag, memory is divided in two, then both parts are XORed into two 128-bit blocks. Both blocks are then input to a variable length hash function (modified Blake2b), which produces tags of arbitrary length.

2.8 Efficiency analysis

2.8.1 Modern x86/x64 architecture

Desktops and servers equipped with modern Intel/AMD CPUs is the primary target platform for Argon. When measuring performance we compute the amortized number of CPU cycles needed to run Argon on a certain amount of memory. An optimized implementation may use the following ideas:

- All the operations are performed on 128-bit blocks, which can be stored in xmm registers.
- The $\mathcal{F}$ transformation, which is the 5-round AES-128, can be implemented by as few as 6 assembler instructions. The high latency of the AES round instructions `aesenc` can be mitigated by pipelining them. The structure of SubGroups is friendly to this idea: each group first applies $\mathcal{F}$ to 16 internal variables in parallel, and then to 32 memory blocks in parallel. The 5-round AES in the counter mode runs at approximately 0.3 cycles per byte. We report the speed of SubGroups from 2.5 to 3 cycles per byte. About a half of this gap should be attributed to 144 128-bit XOR operations.
- We report the speed of ShuffleSlices from 4 to 8 cycles per byte.

Our implementation for $R = 3$, which is still far from optimal, achieves the speed from 30 cycles per byte (when less than 32 MBytes are used) to 45 cycles per byte (when 1 GB is used) on a single core of the x64 Sandy Bridge laptop with 4 GB of RAM.

2.8.2 Older CPU

The absence of dedicated AES instructions will certainly decrease the speed of Argon. Since the 5-round AES-128 in the Counter mode can run at about 5 cycles per byte, we expect the SubGroups to become a bottleneck in the implementation, with the speed records getting close to 40-50 cycles per byte.

2.8.3 Other architectures

We expect that Argon is unsuitable for the GPU architecture. Even though it is parallelizable and the SubGroups transformation can be run on separate cores, the following ShuffleSlices transformation would need to obtain outputs from each group. This assumes an active use of memory and thus very high latency of ShuffleSlices on GPUs. As a result, the GPU-based password crackers should significantly suffer in performance so that they would not be cost-efficient.

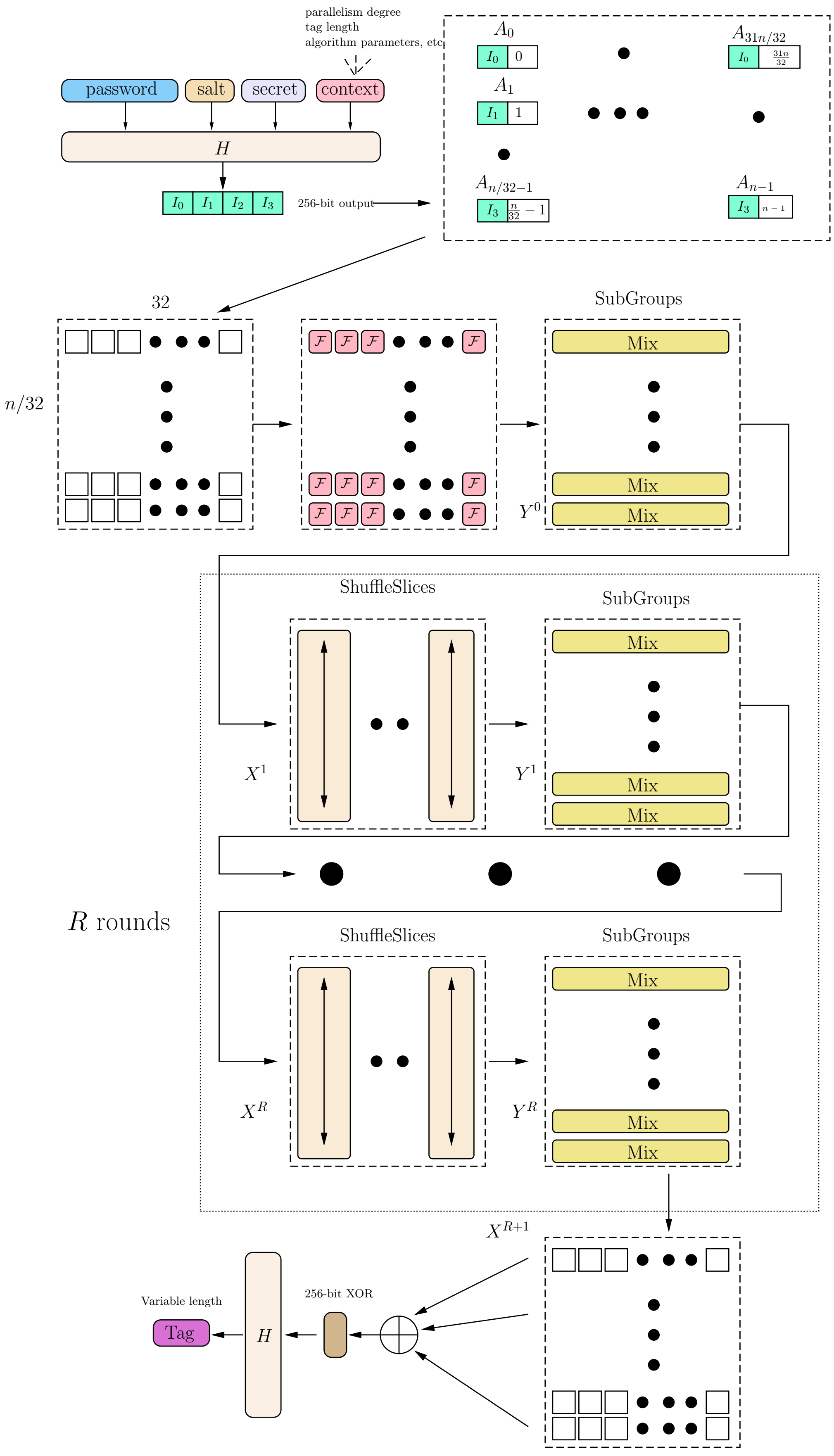

Figure 2.2: Overview of Argon. Transformation $\mathcal{F}$ is the 5-round AES-128.

Chapter 3

Argon2

In this chapter we introduce the next generation of memory-hard hash function Argon, which we call Argon2. Argon2 is built upon the most recent research in the design and analysis of memory-hard functions. It is suitable for password hashing, password-based key derivation, cryptocurrencies, proofs of work/space, etc.

3.1 Specification

There are two flavors of Argon2 – Argon2d and Argon2i. The former one uses data-dependent memory access to thwart tradeoff attacks. However, this makes it vulnerable for side-channel attacks, so Argon2d is recommended primarily for cryptocurrencies and backend servers. Argon2i uses data-independent memory access, which is recommended for password hashing and password-based key derivation.

3.1.1 Inputs

Argon2 has two types of inputs: primary inputs and secondary inputs, or parameters. Primary inputs are message P and nonce S, which are password and salt, respectively, for the password hashing. Primary inputs must always be given by the user such that

- Message P may have any length from 0 to $2^{32} - 1$ bytes;
- Nonce S may have any length from 8 to $2^{32} - 1$ bytes (16 bytes is recommended for password hashing).

Secondary inputs have the following restrictions:

- Degree of parallelism p may take any integer value from 1 to 64.
- Tag length τ may be any integer number of bytes from 4 to $2^{32} - 1$.
- Memory size m can be any integer number of kilobytes from $8p$ to $2^{32} - 1$, but it is rounded down to the nearest multiple of $4p$.
- Number of iterations t can be any integer number from 1 to $2^{32} - 1$;
- Version number v is one byte $0x10$;
- Secret value K may have any length from 0 to 32 bytes.
- Associated data X may have any length from 0 to $2^{32} - 1$ bytes.

Argon2 uses compression function G and hash function H. Here H is the Blake2b hash function, and $\mathcal{G}$ is based on its internal permutation. The mode of operation of Argon2 is quite simple when no parallelism is used: function G is iterated m times. At step i a block with index $\phi(i) < i$ is taken from the memory (Figure 3.1), where $\phi(i)$ is either determined by the previous block in Argon2d, or is a fixed value in Argon2i.

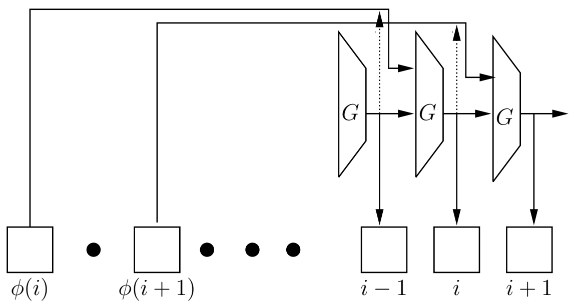

Figure 3.1: Argon2 mode of operation with no parallelism.

3.1.2 Operation

Argon2 follows the extract-then-expand concept. First, it extracts entropy from message and nonce by hashing it. All the other parameters are also added to the input. The variable length inputs P and S are prepended with their lengths:

$$H_0 = H(d, \tau, m, t, v, \langle P\rangle, P, \langle S\rangle, S, \langle K\rangle, K, \langle X\rangle, X).$$

Here H_0 is a 64-byte value.

Argon2 then fills the memory with m 1024-byte blocks. For tunable parallelism with p threads, the memory is organized in a matrix $B[i][j]$ of blocks with p rows (*lanes*) and $q = \lfloor m/d \rfloor$ columns. Blocks are computed as follows:

$$\begin{aligned}
B[i][0] &= H'(H_0 || \underbrace{i}_{\text{4 bytes}} || \underbrace{0}_{\text{4 bytes}}), \quad 0 \le i < p;\\
B[i][1] &= H'(H_0 || \underbrace{i}_{\text{4 bytes}} || \underbrace{1}_{\text{4 bytes}}), \quad 0 \le i < p;\\
B[i][j] &= G(B[i][j-1], B[i'][j']), \quad 0 \le i < p,\ 2 \le j < q.
\end{aligned}$$

where block index $[i'][j']$ is determined differently for Argon2d and Argon2i, G is the compression function, and H' is a variable-length hash function built upon H. Both will be fully defined in the further text.

If $t > 1$, we repeat the procedure; however the first two columns are now computed in the same way:

$$\begin{aligned}
B[i][0] &= G(B[i][q-1], B[i'][j']);\\
B[i][j] &= G(B[i][j-1], B[i'][j']).
\end{aligned}$$

When we have done t iterations over the memory, we compute the final block B_m as the XOR of the last column:

$$B_m = B[0][q-1] \oplus B[1][q-1] \oplus \cdots \oplus B[d-1][q-1].$$

Then we apply H' to B_m to get the output tag.

Variable-length hash function. We define H', the variable-length hash function based on Blake2, as follows. Let V_i be a 512-bit block, and A_i be its first 256 bits, and $\tau < 2^{32}$ be the tag length in bytes. Then we define

$$\begin{aligned}
H'(X): \quad & V_0 \leftarrow H(\tau || X);\\
& V_1 \leftarrow H(V_0);\\
& \dots\\
& V_m \leftarrow H(V_{m-1}), \quad m = \lceil \tau/64 \rceil - 1\\
& \text{Tag} \leftarrow A_0 || A_1 || A_2 || \dots || A_m.
\end{aligned}$$

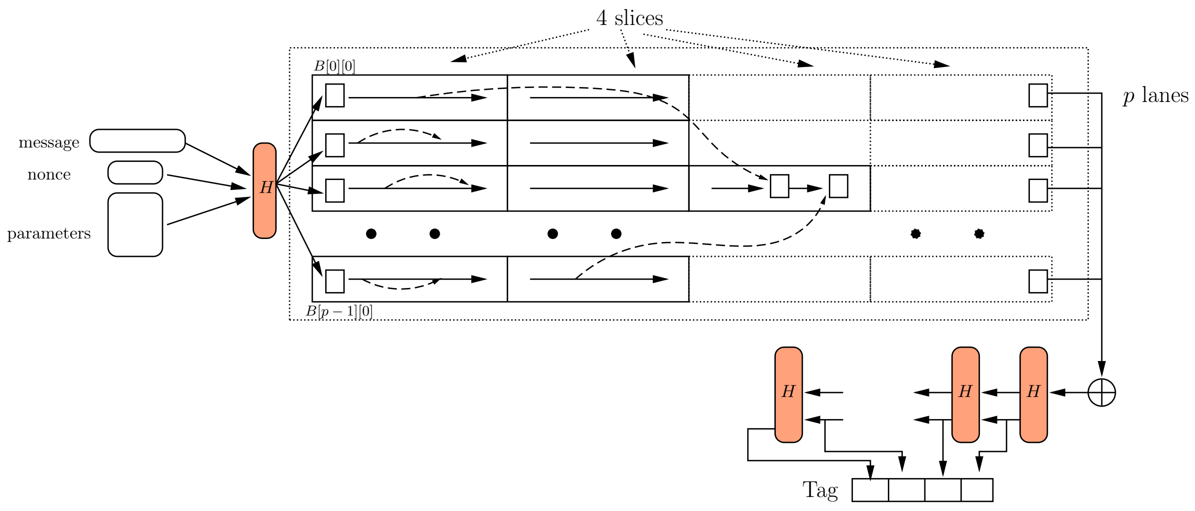

Figure 3.2: Single-pass Argon2 with p lanes and 4 slices.

3.1.3 Indexing

Now we explain how the index $[i'][j']$ of the reference block is computed. First, we determine the set of indices $\mathcal{R}$ that can be referenced for given $[i][j]$. For that we further partition the memory matrix into $l = 4$ vertical *slices*. Intersection of a slice and a segment is *segment* of length q/l. Thus segments form their own matrix Q of size $p \times l$. Segments of the same slice are computed in parallel, and may not reference blocks from each other. All other blocks can be referenced. Suppose we are computing a block in a segment $Q[r][s]$. Then $\mathcal{R}$ includes blocks according to the following rules:

1. All blocks of segments $Q[r'][*]$, where $r' < r$ and $*$ takes all possible values from 0 to $p - 1$.
2. All blocks of segments $Q[r'][*]$, where $r' > r$ and $*$ takes all possible values from 0 to $p - 1$ — if it is the second or later pass over the memory.
3. All blocks of segment $Q[r][s]$ (current segment) except for the previous block;
4. For the first block of the segment $\mathcal{R}$ does not include the previous block of any lane.

Let R be the size of $\mathcal{R}$. The blocks in $\mathcal{R}$ are numbered from 0 to $R - 1$ according to the following rules:

- Blocks outside of the current segment are enumerated first;
- Segments are enumerated from top to down, and then from left to right: $Q[0][0]$, then $Q[1][0]$, then up to $Q[p-1][0]$, then $Q[0][1]$, then $Q[1][1]$, etc.
- Blocks within a segment are enumerated from the oldest to the newest.

Then Argon2d selects a block from $\mathcal{R}$ randomly, and Argon2i – pseudorandomly as follows:

- In Argon2d we take the first 32 bits of block $B[i][j-1]$ and denote this block by J. Then the value $J \pmod{R}$ determines the block number from $\mathcal{R}$.
- In Argon2i we run $G(X, G(X, Y))$ — the 2-round compression function G, where X is all-zero block, and Y is constructed as

$$(\underbrace{r}_{\text{4 bytes}} || \underbrace{l}_{\text{4 bytes}} || \underbrace{s}_{\text{4 bytes}} || \underbrace{i}_{\text{4 bytes}} || \underbrace{0xFFFFFFFF}_{\text{4 bytes}} || \underbrace{0}_{\text{1004 bytes}},$$

  where r is the pass number, l is the lane, s is the slice, and i is is the counter starting in each segment from 0. Then we increase the counter so that each application of G^2 gives 256 values of J, which are used to reference available blocks exactly as in Argon2d.

3.1.4 Compression function G

Compression function G is built upon the Blake2b round function $\mathcal{P}$ (fully defined in Section 3.7.1). $\mathcal{P}$ operates on the 128-byte input, which can be viewed as 8 16-byte registers (see details below):

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

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

$$\begin{aligned}
(Q_0, Q_1, \ldots, Q_7) &\leftarrow \mathcal{P}(R_0, R_1, \ldots, R_7);\\
(Q_8, Q_9, \ldots, Q_{15}) &\leftarrow \mathcal{P}(R_8, R_9, \ldots, R_{15});\\
&\ldots\\
(Q_{56}, Q_{57}, \ldots, Q_{63}) &\leftarrow \mathcal{P}(R_{56}, R_{57}, \ldots, R_{63});\\
\\
(Z_0, Z_8, Z_{16}, \ldots, Z_{56}) &\leftarrow \mathcal{P}(Q_0, Q_8, Q_{16}, \ldots, Q_{56});\\
(Z_1, Z_9, Z_{17}, \ldots, Z_{57}) &\leftarrow \mathcal{P}(Q_1, Q_9, Q_{17}, \ldots, Q_{57});\\
&\ldots\\
(Z_7, Z_{15}, Z_{23}, \ldots, Z_{63}) &\leftarrow \mathcal{P}(Q_7, Q_{15}, Q_{23}, \ldots, Q_{63}).
\end{aligned}$$

Finally, G outputs $Z \oplus R$:

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

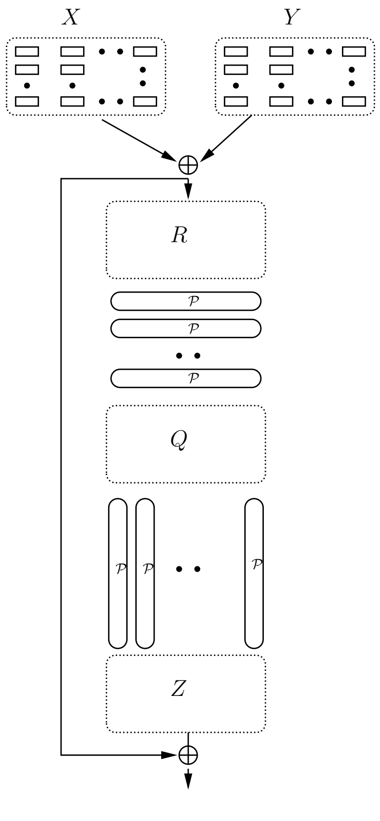

Figure 3.3: Argon2 compression function G.

3.2 Features

Argon2 is a multi-purpose family of hashing schemes, which is suitable for password hashing, key derivation, cryptocurrencies and other applications that require provably high memory use. Argon2 is optimized for the x86 architecture, but it does not slow much on older processors. The key feature of Argon2 is its performance and the ability to use multiple computational cores in a way that prohibit time-memory tradeoffs. Several features are not included into this version, but can be easily added later.

3.2.1 Available features

Now we provide an extensive list of features of Argon.

Design rationality	Argon2 follows a simple design with three main ideas: generic mode of operation, tradeoff-resilient parallelism with synchronization points, and tradeoff-resistant compression function. All design choices are based on the intensive research and explained below.
Performance	Argon2 fills memory very fast. Data-independent version Argon2i securely fills the memory spending about a CPU cycle per byte, and Argon2d is twice as fast. This makes it suitable for applications that need memory-hardness but can not allow much CPU time, like cryptocurrency peer software.
Tradeoff resilience	Despite high performance, Argon2 provides reasonable level of tradeoff resilience. Our tradeoff attacks previously applied to Catena and Lyra2 show the following. With default number of passes over memory (1 for Argon2d, 3 for Argon2i, an ASIC-equipped adversary can not decrease the time-area product if the memory is reduced by the factor of 4 or more. Much higher penalties apply if more passes over the memory are made.
Scalability	Argon2 is scalable both in time and memory dimensions. Both parameters can be changed independently provided that a certain amount of time is always needed to fill the memory.
Parallelism	Argon2 may use up to 64 threads in parallel, although in our experiments 8 threads already exhaust the available bandwidth and computing power of the machine.
Server relief	Argon2 allows the server to carry out the majority of computational burden on the client in case of denial-of-service attacks or other situations. The server ultimately receives a short intermediate value, which undergoes a preimage-resistant function to produce the final tag.
Client-independent updates	Argon2 offers a mechanism to update the stored tag with an increased time and memory cost parameters. The server stores the new tag and both parameter sets. When a client logs into the server next time, Argon is called as many times as need using the intermediate result as an input to the next call.
Possible extensions	Argon2 is open to future extensions and tweaks that would allow to extend its capabilities. Some possible extensions are listed in Section 3.2.4.
GPU/FPGA/ASIC-unfriendly	Argon2 is heavily optimized for the x86 architecture, so that implementing it on dedicated cracking hardware should be neither cheaper nor faster. Even specialized ASICs would require significant area and would not allow reduction in the time-area product.
Secret input support	Argon2 natively supports secret input, which can be *key* or any other secret string.
Associated data support	Argon2 supports additional input, which is syntactically separated from the message and nonce, such as environment parameters, user data, etc..

3.2.2 Server relief

The server relief feature allows the server to delegate the most expensive part of the hashing to the client. This can be done as follows (suppose that K and X are not used):

1. The server communicates S, m, R, d, τ to the client.
2. The client performs all computations up to the value of (B_0, B_1);
3. The client communicates (B_0, B_1) to the server;
4. The server computes Tag and stores it together with S, m, d, R.

The tag computation function is preimage-resistant, as it is an iteration of a cryptographic hash function. Therefore, leaking the tag value would not allow the adversary to recover the actual password nor the value of B by other means than exhaustive search.

3.2.3 Client-independent update

It is possible to increase the time and memory costs on the server side without requiring the client to re-enter the original password. We just compute

$$\mathrm{Tag}_{new} = \Pi(\mathrm{Tag}_{old}, S, m_{new}, R_{new}, \tau_{new}),$$

replace Tag_{old} with Tag_{new} in the hash database and add the new values of m and R to the entry.

3.2.4 Possible future extensions

Argon can be rather easily tuned to support the following extensions, which are not used now for design clarity or performance issues:

Support for other hash functions	Any cryptographically strong hash function may replace Blake2b for the initial and the final phases. However, if it does not support variable output length natively, some care should be taken when implementing it.
Support for other compression functions and block sizes	Compression function can be changed to any other transformation that fulfills our design criteria. It may use shorter or longer blocks.
Support for Read-Only-Memory	ROM can be easily integrated into Argon2 by simply including it into the area where the blocks are referenced from.

3.3 Design rationale

Argon2 was designed with the following primary goal: to maximize the cost of exhaustive search on non-x86 architectures, so that the switch even to dedicated ASICs would not give significant advantage over doing the exhaustive search on defender's machine.

3.3.1 Cost estimation

We have investigated several approaches to estimating the real cost of the brute-force. As the first approximation, we calculate the hardware design&production costs separately from the actual running costs. The choice of the platform is determined by adversary's budget: if it is below 100000 EUR or so, then GPU and/or FPGA are the reasonable choices, whereas attackers with a million budget (governments or their subsidiaries) can afford development and production of the dedicated hardware (ASIC).

If ASICs are not considered, the hashing schemes that use more than a few hundred MB of RAM are almost certainly inefficient on a GPU or a FPGA. GPUs efficiency gain (compared to regular desktops) is rather moderate even on low-memory computations, which can be parallelized. If we consider a scheme with a low degree of parallelism and intensive memory use, then the high memory latency of the GPU architecture would make the brute-force much less efficient.

Argon2 protects against all types of adversaries, including those who can afford design and production of dedicated hardware for password cracking or cryptocurrency mining. However, modelling the attack cost is not easy. Our first motivation was to calculate the exact running costs [10] by outlining a hypothetical brute-force chip, figuring out the energy consumption of all components and calculate the total energy-per-hash cost. However, we found out that the choice of the memory type and assumptions on its properties affect the final numbers significantly. For example, taking a low-leakage memory [26] results in the underestimation of the real power consumption, because we do not know exactly how it scales exactly with increasing the size (the paper [26] proposes a 32-KB chip). On the other hand, the regular DDR3 as a reference also creates ambiguity, since it is difficult to estimate its idle power consumption [1].

To overcome these problems, we turn to a more robust metric of the time-area product [27, 8]. In the further text, we'll be interested in the area requirements by memory, by the Blake2 core [7], and the maximum possible bandwidth existing in commercial products.

- The 50-nm DRAM implementation [15] takes 550 mm^2 per GByte;
- The Blake2b implementation in the 65-nm process should take about 0.1 mm^2 (using Blake-512 implementation in [17]);
- The maximum memory bandwidth achieved by modern GPUs is around 400 GB/sec.

3.3.2 Mode of operation

We decided to make a simple mode of operation. It is based on the single-pass, single-thread filling memory array $B[]$ with the help of the compression function G:

$$\begin{aligned} B[0] &= H(P,S);\\ \text{for } j &\quad \text{from } 1 \text{ to } t\\ &\quad B[j] = G\big(B[\phi_1(j)], B[\phi_2(j)], \cdots, B[\phi_k(j)]\big), \end{aligned} \tag{3.1}$$

where $\phi_i()$ are some indexing functions.

This scheme was extended to implement:

- Tunable parallelism;
- Several passes over memory.

Number of referenced blocks. The number k is a tradeoff between performance and bandwidth. First, we set $\phi_k(j) = j-1$ to maximize the total chain length. Then we decide on the exact value of k. One might expect that this value should be determined by the maximum L1 cache bandwidth. Two read ports at the Haswell CPU should have allowed to spend twice fewer cycles to read than to write. In practice, however, we do not get such speed-up with our compression function:

k	Cycles/block	Bandwidth (GB/sec)
2	1194	4.3
3	1503	5.1

Thus extra block decreases the speed but increases the bandwidth. Larger values of k would further reduce the throughput and thus the total amount of memory filled in the fixed time. However, an adversary will respond by creating a custom CPU that will be marginally larger but having k load ports, thereby keeping the time the same. Thus we conclude that the area-time product of attacker's hardware is maximized by $k=2$ or $k=3$, depending on the compression function. We choose $k=2$ as we aim for faster memory fill and more passes over the memory.

Number of output blocks. We argue that only one output block should be produced per step if we use $k=2$. An alternative would be to have some internal state of size $t' \geq t$ and to iterate F several times (say, l') with outputting a t-bit block every round. However, it can be easily seen that if $l't < t'$, then an adversary could simply keep the state in the memory instead of the blocks and thus get a free tradeoff. If $l't = t'$, then, in turn, an adversary would rather store the two input blocks.

Indexing functions. For the data-dependent addressing we set $\phi(l) = g(B[l])$, where g simply truncates the block and takes the result modulo $l-1$. We considered taking the address not from the block $B[l-1]$ but from the block $B[l-2]$, which should have allowed to prefetch the block earlier. However, not only the gain in our implementations is limited, but also this benefit can be exploited by the adversary. Indeed, the efficient depth $D(q)$ is now reduced to $D(q)-1$, since the adversary has one extra timeslot. Table 3.1 implies that then the adversary would be able to reduce the memory by the factor of 5 without increasing the time-area product (which is a 25% increase in the reduction factor compared to the standard approach).

For the data-independent addressing we use a simple PRNG, in particular the compression function G in the counter mode. Due to its long output, one call (or two consecutive calls) would produce hundreds of addresses, thus minimizing the overhead. This approach does not give provable tradeoff bounds, but instead allows the analysis with the tradeoff algorithms suited for data-dependent addressing.

3.3.3 Implementing parallelism

As modern CPUs have several cores possibly available for hashing, it is tempting to use these cores to increase the bandwidth, the amount of filled memory, and the CPU load. The cores of the recent Intel CPU share the L3 cache and the entire memory, which both have large latencies (100 cycles and more). Therefore, the inter-processor communication should be minimal to avoid delays.

The simplest way to use p parallel cores is to compute and XOR p independent calls to H:

$$H'(P,S) = H(P,S,0) \oplus H(P,S,1) \oplus \cdots \oplus H(P,S,p).$$

If a single call uses m memory units, then p calls use pm units. However, this method admits a trivial tradeoff: an adversary just makes p sequential calls to H using only m memory in total, which keeps the time-area product constant.

We suggest the following solution for p cores: the entire memory is split into p lanes of l equal slices each, which can be viewed as elements of a $(p \times l)$-matrix $Q[i][j]$. Consider the class of schemes given by Equation (3.1). We modify it as follows:

- p invocations to H run in parallel on the first column $Q[*][0]$ of the memory matrix. Their indexing functions refer to their own slices only;
- For each column $j > 0$, l invocations to H continue to run in parallel, but the indexing functions now may refer not only to their own slice, but also to all jp slices of previous columns $Q[*][0], Q[*][1], \ldots, Q[*][j-1]$.
- The last blocks produced in each slice of the last column are XORed.

This idea is easily implemented in software with p threads and l joining points. It is easy to see that the adversary can use less memory when computing the last column, for instance by computing the slices sequentially and storing only the slice which is currently computed. Then his time is multiplied by $(1 + \frac{p-1}{l})$, whereas the memory use is multiplied by $(1 - \frac{p-1}{pl})$, so the time-area product is modified as

$$AT_{new} = AT \left(1 - \frac{p-1}{pl}\right)\left(1 + \frac{p-1}{l}\right).$$

For $2 \leq p, l \leq 10$ this value is always between 1.05 and 3. We have selected $l = 4$ as this value gives low synchronisation overhead while imposing time-area penalties on the adversary who reduces the memory even by the factor 3/4. We note that values $l = 8$ or $l = 16$ could be chosen.

If the compression function is collision-resistant, then one may easily prove that block collisions are highly unlikely. However, we employ a weaker compression function, for which the following holds:

$$G(X,Y) = F(X \oplus Y),$$

which is invariant to swap of inputs and is not collision-free. We take special care to ensure that the mode of operation does not allow such collisions by introducing additional rule:

- First block of a segment can not refer to the last block of any segment in the previous slice.

We prove that block collisions are unlikely under reasonable conditions on F in Section 3.4.4.

3.3.4 Compression function design

Overview

In contrast to attacks on regular hash functions, the adversary does not control inputs to the compression function G in our scheme. Intuitively, this should relax the cryptographic properties required from the compression function and allow for a faster primitive. To avoid being the bottleneck, the compression function ideally should be on par with the performance of memcpy() or similar function, which may run at 0.1 cycle per byte or even faster. This much faster than ordinary stream ciphers or hash functions, but we might not need strong properties of those primitives.

However, we first have to determine the optimal block size. When we request a block from a random location in the memory, we most likely get a cache miss. The first bytes would arrive at the CPU from RAM within at best 10 ns, which accounts for 30 cycles. In practice, the latency of a single load instruction may reach 100 cycles and more. However, this number can be amortized if we request a large block of sequentially stored bytes. When the first bytes are requested, the CPU stores the next ones in the L1 cache, automatically or using the `prefetch` instruction. The data from the L1 cache can be loaded as fast as 64 bytes per cycle on the Haswell architecture, though we did not manage to reach this speed in our application.

Therefore, the larger the block is, the higher the throughput is. We have made a series of experiments with a non-cryptographic compression function, which does little beyond simple XOR of its inputs, and achieved the performance of around 0.7 cycles per byte per core with block sizes of 1024 bits and larger.

Design criteria

Let us fix the block size t and figure out the design criteria. It appears that collision/preimage resistance and their weak variants are overkill as a design criteria for the compression function F. We recall, however, that the adversary is motivated by reducing the time-area product. Let us consider the following structure of the compression function $F(X, Y)$, where X and Y are input blocks:

- The input blocks of size t are divided into shorter subblocks of length t' (for instance, 128 bits) $X_0, X_1, X_2, \ldots$ and $Y_0, Y_1, Y_2, \ldots$.
- The output block Z is computed subblockwise:

$$\begin{aligned} Z_0 &= G(X_0, Y_0); \\ Z_i &= G(X_i, Y_i, Z_{i-1}),\ i > 0. \end{aligned}$$

This scheme resembles the duplex authenticated encryption mode, which is secure under certain assumptions on G. However, it is totally insecure against tradeoff adversaries, as shown below.

Attack on the iterative compression function. Suppose that an adversary computes $Z = F(X, Y)$ but Y is not stored. Suppose that Y is a tree function of stored elements of depth D. The adversary starts with computing Z_0, which requires only Y_0. In turn, $Y_0 = G(X'_0, Y'_0)$ for some X', Y'. Therefore, the adversary computes the tree of the same depth D, but with the function G instead of F. Z_1 is then a tree function of depth $D + 1$, Z_2 of depth $D + 2$, etc. In total, the recomputation takes $(D + s)L_G$ time, where s is the number of subblocks and L_G is the latency of G. This should be compared to the full-space implementation, which takes time sL_G. Therefore, if the memory is reduced by the factor q, then the time-area product is changed as

$$AT_{new} = \frac{D(q) + s}{sq} AT.$$

Therefore, if

$$D(q) \leq s(q - 1), \tag{3.2}$$

the adversary wins.

One may think of using the $Z_{m-1}[l - 1]$ as input to computing $Z_0[l]$. Clearly, this changes little in adversary's strategy, who could simply store all Z_{m-1}, which is feasible for large m. In concrete proposals, s can be 64, 128, 256 and even larger.

We conclude that F with an iterative structure is insecure. We note that this attack applies also to other PHC candidates with iterative compression function. Table 3.1 and Equation 3.2 suggests that it allows to reduce the memory by the factor of 12 or even higher while still reducing the area-time product.

Our approach We formulate the following design criteria:

- *The compression function must require about t bits of storage (excluding inputs) to compute any output bit.*
- *Each output byte of F must be a nonlinear function of all input bytes, so that the function has differential probability below certain level, for example $\frac{1}{4}$.*

These criteria ensure that the attacker is unable to compute an output bit using only a few input bits or a few stored bits. Moreover, the output bits should not be (almost) linear functions of input bits, as otherwise the function tree would collapse.

We have not found any generic design strategy for such large-block compression functions. It is difficult to maintain diffusion on large memory blocks due to the lack of CPU instructions that interleave many registers at once. A naive approach would be to apply a linear transformation with certain branch number. However, even if we operate on 16-byte registers, a 1024-byte block would consist of 64 elements. A 64×64-matrix would require 32 XORs per register to implement, which gives a penalty about 2 cycles per byte.

Instead, we propose to build the compression function on the top of a transformation P that already mixes several registers. We apply P in parallel (having a P-box), then shuffle the output registers and apply it again. If P handles p registers, then the compression function may transform a block of p^2 registers with 2 rounds of P-boxes. We do not have to manually shuffle the data, we just change the inputs to P-boxes. As an example, an implementation of the Blake2b [7] permutation processes 8 128-bit registers, so with 2 rounds of Blake2b we can design a compression function that mixes the 8192-bit block. We stress that this approach is not possible with dedicated AES instructions. Even though they are very fast, they apply only to the 128-bit block, and we still have to diffuse its content across other blocks.

3.3.5 User-controlled parameters

We have made a number of design choices, which we consider optimal for a wide range of applications. Some parameters can be altered, some should be kept as is. We give a user full control over:

- Amount M of memory filled by algorithm. This value, evidently, depends on the application and the environment. There is no "insecure" value for this parameter, though clearly the more memory the better.
- Number T of passes over the memory. The running time depends linearly on this parameter. We expect that the user chooses this number according to the time constraints on the application. Again, there is no "insecure value" for T.
- Degree d of parallelism. This number determines the number of threads used by an optimized implementation of Argon2. We expect that the user is restricted by a number of CPU cores (or half-cores) that can be devoted to the hash function, and chooses d accordingly (double the number of cores).
- Length of password/message, salt/nonce, and tag (except for some low, insecure values for salt and tag lengths).

We allow to choose another compression function G, hash function H, block size b, and number of slices l. However, we do not provide this flexibility in a reference implementation as we guess that the vast majority of the users would prefer as few parameters as possible.

3.4 Security analysis

3.4.1 Security of single-pass schemes

We consider the following types of indexing functions:

- Independent of the password and salt, but possibly dependent on other public parameters (*data-independent*). Thus the addresses can be calculated by the adversaries. Therefore, if the dedicated hardware can handle parallel memory access, the adversary can prefetch the data from the memory. Moreover, if she implements a time-space tradeoff, then the missing blocks can be also precomputed without losing time. In order to maximize the computational penalties, the designers proposed various formulas for indexing functions [13, 20], but several of them were found weak and admitting easy tradeoffs.
- Dependent on the password (*data-dependent*). A frequent choice is to use some bits of the previous block: $\phi_2(j) = g(B[j-1])$. This choice prevents the adversary from prefetching and precomputing missing data. The adversary figures out what he has to recompute only at the time the element is needed. If an element is recomputed as a tree of F calls of average depth D, then the total processing time is multiplied by D. However, this method is vulnerable to side-channel attacks, as timing information may help to filter out password guesses at the early stage.

- Hybrid schemes, where the first phase uses a data-independent addressing, and next phases use a data-dependent addressing. The side-channel attacks become harder to mount, since an adversary still has to run the first phase to filter out passwords. However, this first phase is itself vulnerable to time-space tradeoffs, as mentioned above.

In the case of *data-dependent* schemes, the adversary can reduce the time-area product if the time penalty due to the recomputation is smaller than the memory reduction factor. The time penalty is determined by the depth D of the recomputation tree, so the adversary wins as long as

$$D(q) \leq q.$$

In contrast, the cracking cost for *data-independent* schemes, expressed as the time-area product, is easy to reduce thanks to tradeoffs. The total area decreases until the area needed to host multiple cores for recomputation matches the memory area, whereas the total time remains stable until the total bandwidth required by the parallelized recomputations exceeds the architecture capabilities.

Let us elaborate on the first condition. When we follow some tradeoff strategy and reduce the memory by the factor of q, the total number of calls to G increases by the factor $C(q)$. Suppose that the logic for G takes A_{core} of area (measured, say, in mm^2), and the memory amount that we consider (say, 1 GB), takes A_{memory} of area. The adversary reduces the total area as long as:

$$C(q)A_{core} + A_{memory}/q \leq A_{memory}.$$

The maximum bandwidth Bw_{max} is a hypothetical upper bound on the memory bandwidth on the adversary's architecture. Suppose that for each call to G an adversary has to load $R(q)$ blocks from the memory on average, where q is the memory reduction factor. Therefore, the adversary can keep the execution time the same as long as

$$R(q)Bw \leq Bw_{max},$$

where Bw is the bandwidth achieved by a full-space implementation.

Lemma 1.

$$R(q) = C(q).$$

This lemma is proved in Section 3.7.2.

3.4.2 Ranking tradeoff attack

To figure out the costs of the ASIC-equipped adversary, we first need to calculate the time-space tradeoffs for our class of hashing schemes. To the best of our knowledge, the first generic tradeoffs attacks were reported in [10], and they apply to both data-dependent and data-independent schemes. The idea of the ranking method [10] is as follows. When we generate a memory block $X[l]$, we make a decision, to store it or not. If we do not store it, we calculate the access complexity of this block — the number of calls to F needed to compute the block, which is based on the access complexity of $X[l-1]$ and $X[\phi(l)]$. The detailed strategy is as follows:

1. Select an integer q (for the sake of simplicity let q divide T).
2. Store $X[kq]$ for all k;
3. Store all r_i and all access complexities;
4. Store the T/q highest access complexities. If $X[i]$ refers to a vertex from this top, we store $X[i]$.

The memory reduction is a probabilistic function of q. We reimplemented the algorithm and applied to the scheme 3.1 where the addresses are randomly generated. We obtained the results in Table 3.1. Each recomputation is a tree of certain depth, also given in the table.

We conclude that for data-dependent one-pass schemes the adversary is always able to reduce the memory by the factor of 4 and still keep the time-area product the same.

3.4.3 Multi-pass schemes

If the defender has more time than needed to fill the available memory, then he can run several passes on the memory. Also some designers decided to process memory several times to get better time-space tradeoffs. Let us figure out how the adversary's costs are affected in this case.

Suppose we make K passes with T iterations each following the scheme (3.1), so that after the first pass any address in the memory may be used. Then this is equivalent to running a single pass with KT iterations such that $\phi(j) \geq j - T$. The time-space tradeoff would be the same as in a single pass with T iterations and additional condition

$$\phi(j) \geq j - \frac{T}{K}.$$

We have applied the ranking algorithm (Section 3.4.2) and obtained the results in Tables 3.2,3.3. We conclude that for the data-dependent schemes using several passes does increase the time-area product for the adversary who uses tradeoffs. Indeed, suppose we run a scheme with memory A with one pass for time T, or on $A/2$ with 2 passes. If the adversary reduces the memory to $A/6$ GB (i.e. by the factor of 6) for the first case, the time grows by the factor of 8.2, so that the time-area product is $1.35AT$. However, if in the second setting the memory is reduced to $A/6$ GB (i.e. by the factor of 3), the time grows by the factor of 14.3, so that the time-area product is $2.2AT$. For other reduction factors the ratio between the two products remains around 2.

Nevertheless, we do not immediately argue for the prevalence of multi-pass schemes, since it can be possible that new tradeoff algorithms change their relative strength.

3.4.4 Security of Argon2 to generic attacks

Now we consider preimage and collision resistance of both versions of Argon2. Variable-length inputs are prepended with their lengths, which shall ensure the absence of equal input strings. Inputs are processed by a cryptographic hash function, so no collisions should occur at this stage.

Internal collision resistance. The compression function G is not claimed to be collision resistant, so it may happen that distinct inputs produce identical outputs. Recall that G works as follows:

$$G(X, Y) = P(Z) \oplus (Z), \quad Z = X \oplus Y.$$

where P is a permutation based on the 2-round Blake2b permutation. Let us prove that all Z are different under certain assumptions.

Theorem 1. *Let Π be Argon2d or Argon2i with d lanes, s slices, and t passes over memory. Assume that*

- *$P(Z) \oplus Z$ is collision-resistant, i.e. it is hard to find a, b such that $P(a) \oplus a = P(b) \oplus b$.*
- *$P(Z) \oplus Z$ is 4-generalized-birthday-resistant, i.e. it is hard to find distinct a, b, c, d such that $P(a) \oplus P(b) \oplus P(c) \oplus P(d) = a \oplus b \oplus c \oplus d$.*

Then all the blocks $B[i]$ generated in those t passes are different.

Proof. By specification, the value of Z is different for the first two blocks of each segment in the first slice in the first pass. Consider the other blocks.

Let us enumerate the blocks according to the moment they are computed. Within a slice, where segments can be computed in parallel, we enumerate lane 0 fully first, then lane 1, etc.. Slices are then computed and enumerated sequentially. Suppose the proposition is wrong, and let $(B[a], B[b])$ be a block

Memory fraction $(1/q)$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{10}$	$\frac{1}{11}$
Computation&Read penalty $C(q)$	1.71	2.95	6.3	16.6	55	206	877	4423	$2^{14.2}$	$2^{16.5}$
Depth penalty $(D(q))$	1.7	2.5	3.8	5.7	8.2	11.5	15.7	20.8	26.6	32.8

Table 3.1: Time and computation penalties for the ranking tradeoff attack for random addresses.

Memory fraction $(1/q)$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
1 pass	1.7	3	6.3	16.6	55
2 passes	15	410	19300	2^{20}	2^{25}
3 passes	3423	2^{22}	2^{32}		

Table 3.2: Computation/read penalties for the ranking tradeoff attack.

Memory fraction $(1/q)$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$
1 pass	1.7	2.5	3.8	5.7	8.2
2 passes	5.7	14.3	28.8	49	75
3 passes	20.7	56	103	–	–

Table 3.3: Depth penalties for the ranking tradeoff attack.

collision such that $x < y$ and y is the smallest among all such collisions. As $F(Z) \oplus Z$ is collision resistant, the collision occurs in Z, i.e.

$$Z_x = Z_y.$$

Let r_x, r_y be reference block indices for $B[x]$ and $B[y]$, respectively, and let p_x, p_y be previous block indices for $B[x], B[y]$. Then we get

$$B[r_x] \oplus B[p_x] = B[r_y] \oplus B[p_y].$$

As we assume 4-generalized-birthday-resistance, some arguments are equal. Consider three cases:

- $r_x = p_x$. This is forbidden by the rule 3 in Section 3.1.3.
- $r_x = r_y$. We get $B[p_x] = B[p_y]$. As $p_x, p_y < y$, and y is the smallest yielding such a collision, we get $p_x = p_y$. However, by construction $p_x \neq p_y$ for $x \neq y$.
- $r_x = p_y$. Then we get $B[r_y] = B[p_x]$. As $r_y < y$ and $p_x < x < y$, we obtain $r_y = p_x$. Since $p_y = r_x < x < y$, we get that x and y are in the same , we have two options:
  - p_y is the last block of a segment. Then y is the first block of a segment in the next slice. Since r_x is the last block of a segment, and $x < y$, x must be in the same slice as y, and x can not be the first block in a segment by the rule 4 in Section 3.1.3. Therefore, $r_y = p_x = x - 1$. However, this is impossible, as r_y can not belong to the same slice as y.
  - p_y is not the last block of a segment. Then $r_x = p_y = y - 1$, which implies that $r_x \geq x$. The latter is forbidden.

Thus we get a contradiction in all cases. This ends the proof. □

The compression function G is not claimed to be collision resistant nor preimage-resistant. However, as the attacker has no control over its input, the collisions are highly unlikely. We only take care that the starting blocks are not identical by producing the first two blocks with a counter and forbidding to reference from the memory the last block as (pseudo)random.

Argon2d does not overwrite the memory, hence it is vulnerable to garbage-collector attacks and similar ones, and is not recommended to use in the setting where these threats are possible. Argon2i with 3 passes overwrites the memory twice, thus thwarting the memory-leak attacks. Even if the entire working memory of Argon2i is leaked after the hash is computed, the adversary would have to compute two passes over the memory to try the password.

3.4.5 Security of Argon2 to tradeoff attacks

Time and computational penalties for 1-pass Argon2d are given in Table 3.1. It suggests that the adversary can reduce memory by the factor of 4 while keeping the time-area product the same.

Argon2i is more vulnerable to tradeoff attacks due to its data-independent addressing scheme. We apply the ranking algorithm to 3-pass Argon2i to calculate time and computational penalties. Table 3.2

Processor	Threads	Argon2d (1 pass)		Argon2i (3 passes)	
		Cycles/Byte	Bandwidth (GB/s)	Cycles/Byte	Bandwidth (GB/s)
Core i7-4600U	2	1	6.36	1.4	9.30
Core i7-4600U	4	0.8	8.31	1.6	11.22
Core i7-4600U	8	0.8	8.07	1.2	11.19

Table 3.4: Cycles/byte and bandwidth of Argon2.

demonstrates that the memory reduction by the factor of 3 already gives the computational penalty of around 2^{14}. The 2^{14} Blake2b cores would take more area than 1 GB of RAM (Section 3.3.1), thus prohibiting the adversary to further reduce the time-area product. We conclude that the time-area product cost for Argon2d can be reduced by 3 at best.

3.5 Performance

3.5.1 x86 architecture

To optimize the data load and store from/to memory, the memory that will be processed has to be alligned on 16-byte boundary when loaded/stored into/from 128-bit registers and on 32-byte boundary when loaded/stored into/from 256-bit registers. If the memory is not aligned on the specified boundaries, then each memory operation may take one extra CPU cycle, which may cause consistent penalties for many memory accesses.

The results presented are obtained on an Intel i7-4600U (Haswell) clocked at 2.1 GHz with 8GHz of memory and the clock speed of 1600 MHz, running 64-bit operating system `Linux Ubuntu 14.04.1 LTS`. The `gcc 4.8.2` compiler was used with the following options: `-O3 -funroll-loops -unroll-aggressive -Wall -pedantic -march=native`. The cycle count value was measured using the `__rdtscp` Intel intrinsics C function which inlines the `RDTSCP` assembly instruction that returns the 64-bit Time Stamp Counter (TSC) value. The instruction waits for prevoius instruction to finish and then is executed, but meanwhile the next instructions may begin before the value is read [19]. Although this shortcoming, we used this method because it is the most realiable handy method to measure the execution time and also it is widely used in other cryptographic operations benchmarking.

3.6 Applications

Argon2d is optimized for settings where the adversary does not get regular access to system memory or CPU, i.e. he can not run side-channel attacks based on the timing information, nor he can recover the password much faster using garbage collection [12]. These settings are more typical for backend servers and cryptocurrency minings. For practice we suggest the following settings:

- Cryptocurrency mining, that takes 0.1 seconds on a 2 Ghz CPU using 1 core — Argon2d with 2 lanes and 250 MB of RAM;
- Backend server authentication, that takes 0.5 seconds on a 2 GHz CPU using 4 cores — Argon2d with 8 lanes and 4 GB of RAM.

Argon2i is optimized for more dangerous settings, where the adversary possibly can access the same machine, use its CPU or mount cold-boot attacks. We use three passes to get rid entirely of the password in the memory. We suggest the following settings:

- Key derivation for hard-drive encryption, that takes 3 seconds on a 2 GHz CPU using 2 cores — Argon2dwith 4 lanes and 6 GB of RAM;
- Frontend server authentication, that takes 0.5 seconds on a 2 GHz CPU using 2 cores — Argon2d with 4 lanes and 1 GB of RAM.

3.7 Other details

3.7.1 Permutation $\mathcal{P}$

Permutation $\mathcal{P}$ is based on the round function of Blake2b and works as follows. Its 8 16-byte inputs $S_0, S_1, \ldots, S_7$ are viewed as a 4×4-matrix of 64-bit words, where $S_i = (v_{2i+1}||v_{2i})$:

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

Then we do

$$\begin{array}{cccc} G(v_0, v_4, v_8, v_{12}) & G(v_1, v_5, v_9, v_{13}) & G(v_2, v_6, v_{10}, v_{14}) & G(v_3, v_7, v_{11}, v_{15}) \\ G(v_0, v_5, v_{10}, v_{15}) & G(v_1, v_6, v_{11}, v_{12}) & G(v_2, v_7, v_8, v_{13}) & G(v_3, v_4, v_9, v_{14}), \end{array}$$

where G applies to (a, b, c, d) as follows:

$$\begin{aligned} &a \leftarrow a + b; \\ &d \leftarrow (d \oplus a) \ggg 32; \\ &c \leftarrow c + d; \\ &b \leftarrow (b \oplus c) \ggg 24; \\ &a \leftarrow a + b; \\ &d \leftarrow (d \oplus a) \ggg 16; \\ &c \leftarrow c + d; \\ &b \leftarrow (b \oplus c) \ggg 63; \end{aligned}$$

Here $+$ are additions modulo 2^{64} and $\ggg$ are 64-bit rotations to the right.

3.7.2 Proof of Lemma 1

Proof. Let A_j be the computational complexity of recomputing $M[j]$. If $M[j]$ is stored, then $A_j = 0$. When we have to compute a new block $M[i]$, then the computational complexity C_i of computing $M[i]$ (measured in calls to F) is calculated as

$$C_i = A_{\phi_2(i)} + 1.$$

and the total computational penalty is calculated as

$$C(q) = \frac{\sum_{i<T}(A_{\phi_2(i)} + 1)}{T}.$$

Let R_j be the total number of blocks to be read from the memory in order to recompute $M[j]$. The total bandwidth penalty is calculated as

$$R(q) = \frac{\sum_{i<T} R_{\phi_2(i)}}{T}.$$

Let us prove that

$$R_j = A_j + 1. \tag{3.3}$$

by induction.

- We store $M[0]$, so for $j = 0$ we have $R_0 = 1$ and $A_0 = 0$.
- If $M[j]$ is stored, then we read it and make no call to F, i.e.

$$A_j = 0; \quad R_j = 1.$$

- If $M[j]$ is not stored, we have to recompute $M[j-1]$ and $M[\phi_2(j)]$:

  $$A_j = A_{j-1} + A_{\phi_2(j)} + 1 = R_{j-1} - 1 + R_{\phi_2(j)} - 1 + 1 = (R_{j-1} + R_{\phi_2(j)}) - 1 = R_j - 1.$$

  The last equation follows from the fact that the total amount of reads for computing $M[j]$ is the sum of necessary reads for $M[j-1]$ and $M[\phi_2(j)]$.

Therefore, we get

$$C(q) = \frac{\sum_{i<T}(A_{\phi_2(i)} + 1)}{T} = \frac{\sum_{i<T} R_{\phi_2(i)}}{T} = R(q).$$

$\square$

Chapter 4

Change log

4.1 v1.1 of Argon2 – 6th February, 2015

- New indexing rule added to avoid collision with a proof.
- New rule to generate first two blocks in each lane.
- Non-zero constant added to the input block used to generate addresses in Argon2i.

4.2 v2 and Argon2 – 31th January, 2015

- New scheme – Argon2 – is introduced;
- Argon is tweaked. Tweaks are detailed in Section 2.7.

4.3 v1 – 15th April, 2014

Typos corrected.

4.4 v1 – 8th April, 2014

The version v1 has the following differences with v0:

- New matrix L (Equation (2.1)). The software that generated the matrix in the previous version had a bug, which resulted in two duplicate rows. As a result, an adversary could use 12.5% less memory in the listed tradeoff attacks, whereas the diffusion and collision/preimage resistance properties remained as claimed. The current matrix has been checked by SAGE for the maximal rank.
- Version v0 mistakenly attributed *garlic* in [14] to the secret part of the input, whereas it must have been called *pepper*. V1 uses the correct naming.

Bibliography

[1] Micron power calculator. `http://www.micron.com/products/support/power-calc`.

[2] *NIST: AES competition*, 1998. `http://csrc.nist.gov/archive/aes/index.html`.

[3] *NIST: SHA-3 competition*, 2007. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[4] *Litecoin - Open source P2P digital currency*, 2011. `https://litecoin.org/`.

[5] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *CHES'10*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010. `https://131002.net/quark/quark_full.pdf`.

[6] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2 – fast secure hashing. `https://blake2.net/`.

[7] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *ACNS'13*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[8] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT'13*, volume 8270 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2013.

[9] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. The Keccak reference, version 3.0, 2011. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[10] Alex Biryukov, Dmitry Khovratovich, and Johann Groszschädl. Tradeoff cryptanalysis of password hashing schemes. Technical report, 2014. available at `https://www.cryptolux.org/images/5/57/Tradeoffs.pdf`.

[11] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In *CHES'11*, volume 6917 of *Lecture Notes in Computer Science*, pages 312–325. Springer, 2011.

[12] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel. Overview of the candidates for the password hashing competition – and their resistance against garbage-collector attacks. Cryptology ePrint Archive, Report 2014/881, 2014. `http://eprint.iacr.org/`.

[13] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive, Report 2013/525.* to appear in Asiacrypt'14.

[14] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. Cryptology ePrint Archive, Report 2013/525, 2013. `http://eprint.iacr.org/`.

[15] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald G. Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor N. Mudge, and David Blaauw. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2013)*, pages 23–35. ACM, 2013.

[16] Shay Gueron. Aes-gcm software performance on the current high end cpus as a performance baseline for caesar competition. 2013. `http://2013.diac.cr.yp.to/slides/gueron.pdf`.

[17] Frank Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens-Peter Kaps. Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates. In *Third SHA-3 Candidate Conference*, March 2012.

[18] Martin E Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-052US. September 2014.

[20] Marcos A. Simplicio Jr, Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. The Lyra2 reference guide, version 2.3.2 (april 4th, 2013). Technical report, 2014. available at `https://password-hashing.net/submissions/specs/Lyra2-v1.pdf`.

[21] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for 2-round Advanced Encryption Standard (AES). *IACR Cryptology ePrint Archive*, 2005:321, 2005.

[22] Lars R Knudsen, Willi Meier, Bart Preneel, Vincent Rijmen, and Sven Verdoolaege. Analysis methods for (alleged) RC4. In *Advances in CryptologyASIACRYPT98*, pages 327–341. Springer, 1998.

[23] Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009. `http://www.tarsnap.com/scrypt/scrypt.pdf`.

[24] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212, 2009.

[25] Matthew Robshaw and Olivier Billet. *New stream cipher designs: the eSTREAM finalists*, volume 4986. Springer, 2008.

[26] Bram Rooseleer, Stefan Cosemans, and Wim Dehaene. A 65 nm, 850 mhz, 256 kbit, 4.3 pj/access, ultra low leakage power memory using dynamic cell stability and a dual swing data link. *J. Solid-State Circuits*, 47(7):1784–1796, 2012.

[27] Clark D. Thompson. Area-time complexity for VLSI. In *STOC'79*, pages 81–88. ACM, 1979.