Argon v1: Password Hashing Scheme

Designers: Alex Biryukov 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

8th April, 2014

Contents

Chapter 1

Introduction

Passwords remain the primary form of the authentication at various web-services. Passwords are usually stored in the hashed form in server's database. These databases are quite often leaked to adversaries, and the passwords tend to have low entropy. Protocol designers use a number of tricks to mitigate these issues. Starting from late 70's, a password is hashed with a random *salt* value to prevent detection of identical passwords across different users and services. The hash functions, 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 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 certain amount of memory to be computed. The password hashing scheme scrypt [14] 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 welcome in cryptocurrency designs [3] if a creator wants to unmotivate 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 [10], where the adversary can trade memory for time and do his job on fast hardware with low memory. In application to password-hashing scheme, 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*memory value remains almost constant. Whether it is possible to impose a harsh penalty on an adversary who wants to spend, say, a quarter or even a half of the total memory is an open question so far.

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 [1, 15, 2] 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.

Our solution. We offer a new hashing scheme called Argon which is optimized for security, clarity, and efficiency. It renders the tradeoff attacks and thus the architecture switch highly inefficient. Argon can be used for password hashing, key derivation, or any other memory-hard computation (e.g., for cryptocurrencies).

Argon is simple. It uses the AES round function [7, 13] as the only external crypto primitive, and uses only XORs and block permutations as internal operations (Chapter 2). Every operation is motivated by a certain goal (Chapter 7).

Argon is secure. It is secure as a hash function, being a pseudorandom function of the salt. It is also secure against tradeoffs. According to our cryptanalytic algorithms, saving half of memory results in the speed penalty factor of 90 and higher. The penalty grows exponentially as the available memory decreases, which effectively prohibits the adversary to use any smaller amount of memory (Chapter 6).

Argon is scalable. It may occupy any integer number of KBytes, and its performance depends strongly linearly on the memory use. Unlike some other schemes the total memory does not have to be a power of two (Chapter 8). Argon is also efficient and can be parallelized on up to 32 threads/cores that share the same memory.

Being so demanding with regard to memory, the use of Argon guarantees that an adversary would have to use exactly the same hardware as the authentication server does. This allows a protocol designer to calculate the attack and defense costs easily and compute a secure set of parameters for Argon.

Chapter 2

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 L (`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 $t = 16$.

2.1 Input

The hashing function Π takes the following inputs:

- Password P of byte length p, $0 \leq p \leq 256$.
- Salt S of byte length s, $8 \leq s \leq 32$.
- Memory size m in kilobytes, $1 \leq m < 2^{32}$.
- Iteration number L, $3 \leq L < 2^{32}$.
- Secret value K of byte length k, $0 \leq k \leq 16$;
- Tag length τ, $8 \leq \tau \leq 32$.

and outputs a tag of length τ:

$$\Pi : \ (P, S, m, L, K, \tau) \rightarrow \text{Tag} \in \{0,1\}^{\tau}.$$

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

$$\texttt{Storage} \leftarrow \langle s \rangle \ || \ \tau \ || \ m \ || \ L \ || \ S \ || \ \text{Tag},$$

with the secret K stored in another place.

Padding phase. The values p, s, k, L, m, τ are known at the start of hashing and are encoded as 4-byte strings $\langle p \rangle, \langle s \rangle, \langle k \rangle \langle L \rangle, \langle m \rangle, \langle \tau \rangle$ using the little-endian convention. Together with P, S, K they form the input I:

$$I = \langle p \rangle \ || \ \langle s \rangle \ || \ \langle k \rangle \ || \ \langle L \rangle \ || \ \langle m \rangle \ || \ \langle \tau \rangle \ || \ P \ || \ S \ || \ K \ || \ 0^*,$$

where 0^* stands a number of zero bytes such that I is $32(t-4)$ bytes long.

The input I is partitioned into $(t-4)$-byte blocks $I_0, I_1, \ldots I_{31}$. Then these blocks are repeatedly used to construct $n = 1024m/t$ blocks of t bytes each with a counter:

$$A_i = I_{i \pmod{32}} \,||\, R_i,$$

where R_i is the 4-byte encoding of the value i.

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

$$A = \begin{pmatrix} A_0 & A_1 & \cdots & A_{31} \\ A_{32} & A_{33} & \cdots & A_{63} \\ \cdots & & & \\ A_{n-32} & A_{n-31} & \cdots & A_{n-1} \end{pmatrix}$$

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 2.2.

Main rounds. In the main rounds the transformations ShuffleSlicesand SubGroups are applied alternatively L times. ShuffleSlices operates on columns, which we call *slices* of A, and is defined in Section 2.3.

Finalization round. The algorithm of the finalization round depends on the required tag length τ. If $\tau \le t$, then all the blocks are xored together:

$$B \leftarrow \bigoplus_i A_i$$

and $\mathcal{F}$ is applied four times with the feedforward:

$$\text{Tag} \leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}((B))))) \oplus B.$$

The tag is then truncated to τ bytes.

If a longer tag is required, then we compute tags from each half of the state and concatenate them:

$$\begin{aligned} B_0 &\leftarrow \bigoplus_{0 \le i < n/2} A_i; \\ B_1 &\leftarrow \bigoplus_{n/2 \le i < n} A_i; \\ \text{Tag}_1 &\leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}((B_1))))) \oplus B_1; \\ \text{Tag}_2 &\leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}((B_2))))) \oplus B_2; \\ \text{Tag} &\leftarrow \text{Tag}_1 || \text{Tag}_2. \end{aligned}$$

The pseudocode of the entire algorithm given in Algorithm 1.

2.2 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 $A_0, A_1, \ldots, A_{n-1}$
$s \leftarrow n/32$
Initial Round:
for $0 \leq j < n$ **do**
  $A_i \leftarrow \mathcal{F}(A_i)$
end
SubGroups:
for $0 \leq j < s$ **do**
  $\mathrm{Mix}(A_{32j}, A_{32j+1}, \ldots, A_{32j+31})$
end
for $1 \leq i \leq L$ **do**
  ShuffleSlices:
  for $0 \leq j < 32$ **do**
    $\mathrm{Shuffle}(A_j, A_{j+32}, \ldots)$;
  end
  SubGroups:
  for $0 \leq j < s$ **do**
    $\mathrm{Mix}(A_{32j}, A_{32j+1}, \ldots, A_{32j+31})$
  end
end
Finalization:
if $\tau \leq t$ **then**
  $B \leftarrow \bigoplus_i A_i$
  $\mathrm{Tag} \leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}(B)))) \oplus B.$
end
else
  $B_0 \leftarrow \bigoplus_{0 \leq i < n/2} A_i$;
  $B_1 \leftarrow \bigoplus_{n/2 \leq i < n} A_i$;
  $\mathrm{Tag}_1 \leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}(B_1)))) \oplus B_1$;
  $\mathrm{Tag}_2 \leftarrow \mathcal{F}(\mathcal{F}(\mathcal{F}(\mathcal{F}(B_2)))) \oplus B_2$;
  $\mathrm{Tag} \leftarrow \mathrm{Tag}_1 || \mathrm{Tag}_2.$
end

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

The X_i are computed as linear functions of A_i (motivation given in Section 7.1):

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

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

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

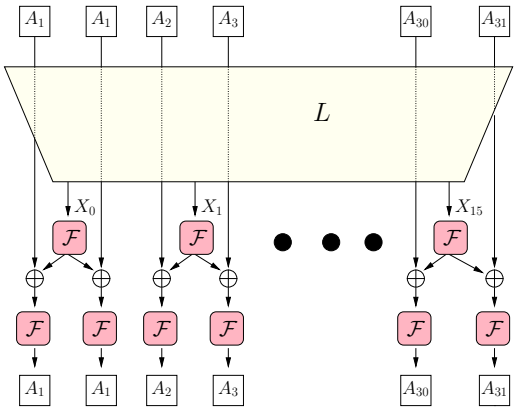

Figure 2.1: Mix transformation.

2.3 ShuffleSlices

The ShuffleSlices transformation operates on *slices*. Blocks in a slice $\mathcal{S}_i$ $(1 \leq i \leq 32)$ stay 32 positions apart from each other:

$$\mathcal{S}_i = \langle A_i,\, A_{i+32}, \ldots, A_{n-32+i} \rangle.$$

Each slice is shuffled independently of the others with the same algorithm Shuffle. Let us denote the 32-bit subwords of state A by A^0, A^1, A^2, A^3. The Shuffle algorithm is derived from the RC4 permutation algorithm and operates according to Algorithm 2.

Input: Slice $\langle B_0, B_1, \ldots, B_{s-1} \rangle$.
$s \leftarrow n/32$
$j \leftarrow 0$
for $0 \leq i \leq s-1$ **do**
  $j \leftarrow j + B_i^0 \pmod{s}$;
  Swap(B_i, B_j);
end

Algorithm 2: The Shuffle algorithm.

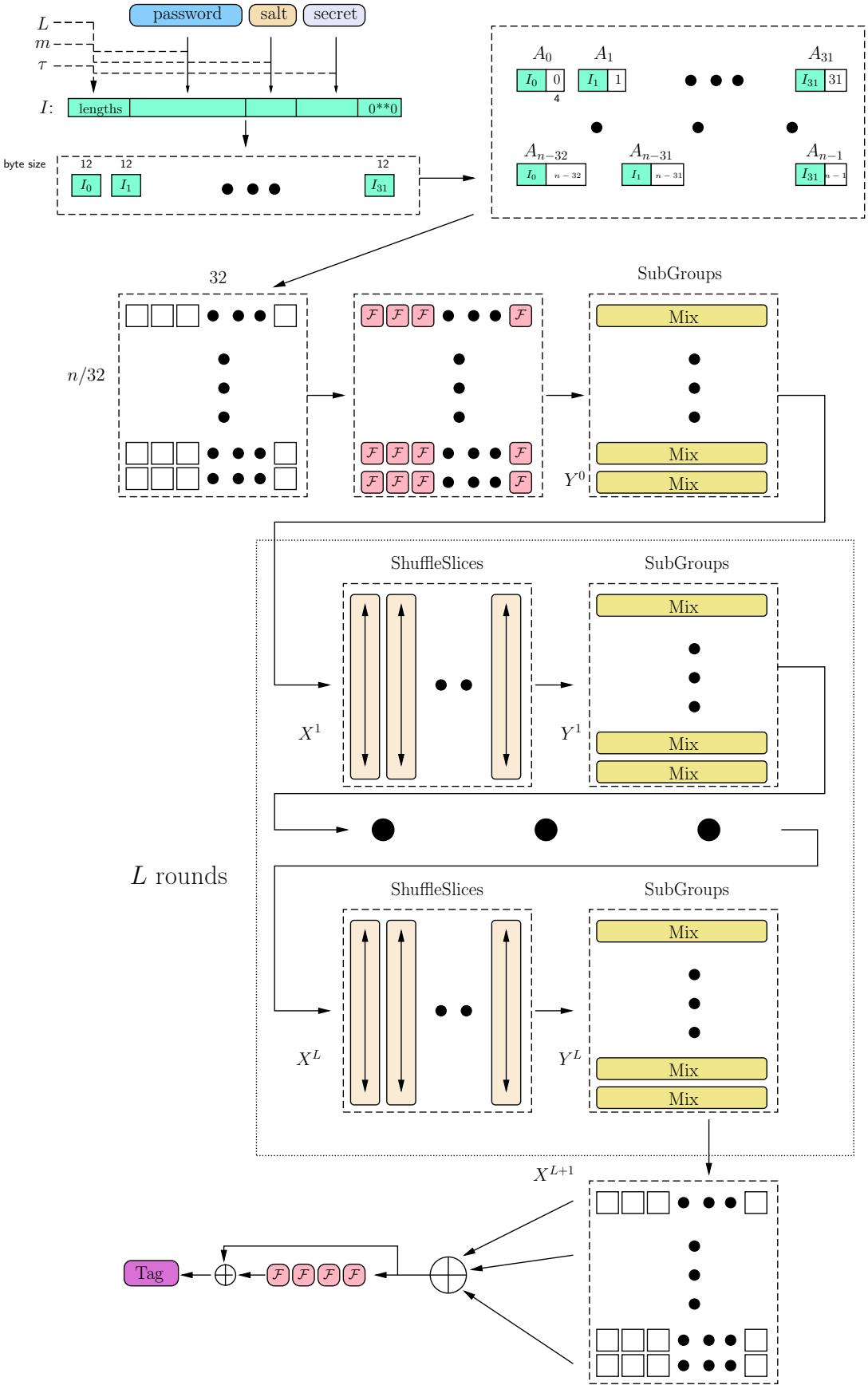

Figure 2.2: Overview of Argon. Transformation $\mathcal{F}$ is the 5-round AES-128.

Chapter 3

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 6. Future cryptanalysis may reduce the minimal `t_cost` values.

`m_cost`	1	10	100	10^3	10^4	10^5	10^6
Memory used	1 KB	10 KB	100 KB	1 MB	10 MB	100 MB	1 GB
Minimal `t_cost`	254	236	56	3	3	3	3

Table 3.1: 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.

Chapter 4

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 $L = 3$ (Chapter 6):

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}	236	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. However, cache-timing attacks are extremely hard to mount on a remote machine, so we presume their application on Argon being purely theoretical.

Chapter 5

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.

5.1 Main features

Now we provide an extensive list of features of Argon.

Clarity	Except for the 5-round fixed-key AES, which used as a black-box, Argonuses only XORs and block swaps to process the state. It is easy to implement and, except for AES, requires no external crypto primitive.
Design rationality	Argon follows a simple design strategy, which has 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 5.4.
CPU-friendly	Implementation of Argon benefits from modern CPUs that are equipped with dedicated AES instructions [9] 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* [8].

5.2 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:

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

The tag computation function is preimage-resistant, as it corresponds to the 10-round AES permutation. 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.

5.3 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

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

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

5.4 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:

Unlimited password/salt length	Password and salt may have arbitrary length, which would mean increasing the length of I. In terms of security, this results in a smaller fraction of groups affected by a single byte change in a password and hence in a weaker avalanche effect. The minimally secure number L of iterations must be updated accordingly.
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 [5], Spongent [6], or Quark [4] may be used.

Chapter 6

Security analysis

6.1 Avalanche properties

Consider the SubGroups transformation. A difference in a single block activates at least 3 middle $\mathcal{F}$ calls, and at least 6 output blocks. To have all middle $\mathcal{F}$ inactive one would need at least 8 input blocks active (this comes from the fact that the Reed-Muller code (2,5) is self-dual).

A difference in a single byte of input I activates all groups with at least 6 active blocks each in X^1. Therefore, 6 slices become fully active, and all the groups remain active in the next round.

The RC4 permutation within ShuffleSlices has not been previously applied to very small slices (2, 4, 6, etc.), so we expect some artifacts appearing for too short slices. To mitigate further attacks, we propose to increase the number of iterations for small n, for instance to make

$$L = \max(3, 256 - \frac{n}{32}) = \max(3, 256 - 2 \cdot \texttt{m\_cost}).$$

6.2 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.

6.3 Collision and preimage attacks

Since the 4-round AES has the expected differential probability bounded by 2^{-113} [11], 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.

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

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

6.4.1 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 7.2). Let us denote by σ_i the permutation over X^i realized by ShuffleSlices. The attack algorithm is as follows:

- Sequentially compute $\sigma_1, \sigma_2, \ldots, \sigma_L$. To compute σ_l, use the knowledge of $\sigma_1, \sigma_2, \ldots, \sigma_{l-1}$.
- For each group in Y^L, compute its output blocks in X^{L+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,

$$T_{\mathcal{F}}(X^l) = 8T_{\mathcal{F}}(Y^{l-1}) + 2; \tag{6.1}$$

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

$$\begin{aligned} &T_{\mathcal{F}}(X^{L+1}) = T_{\mathcal{F}}(X^L) + 1.5; \\ &\frac{T_{\mathcal{F}}(\mathrm{Tag})}{n} \approx 1.25 \cdot 8^L + 1.5 + \sum_{i=1}^{L} T_{\mathcal{F}}(\sigma_l) = 1.25 \cdot 8^L + 1.5 + \sum_{i=1}^{L} 1.25 \cdot 8^i \approx 2.6 \cdot 8^L. \end{aligned}$$

and the penalty is

$$\mathcal{P} = \frac{2.6 \cdot 8^L}{1.5L + 2.5} \approx 190 \text{ for } L = 3 \quad \text{and } 1253 \text{ for } L = 4. \tag{6.2}$$

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

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

where M is counted in bytes.

Some examples for various n are given in Table 6.1.

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

Table 6.1: Computational penalty when storing permutations only.

6.4.2 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 $\sigma_1, \sigma_2, \ldots, \sigma_L$. To compute σ_l, use the knowledge of $\mathcal{F}(X_i)$ and $\sigma_1, \sigma_2, \ldots, \sigma_{l-1}$.
- For each group in Y^L, compute its output blocks in X^{L+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 (6.1) holds. Therefore, we have

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

and the penalty is

$$\mathcal{P} = \frac{(20 - 32\alpha)8^{L-1}}{1.5L + 2.5}. \tag{6.3}$$

For $L = 3$ we have the penalties depending on α in Table 6.2.

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

Table 6.2: Computational penalty on an adversary who stores some $\mathcal{F}(X_i)$ in αM memory and additionally all the permutations.

6.4.3 Summary

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

- If
$$\beta \le L\frac{\lg M - 9}{128},$$
then the adversary is recommended to spend the memory entirely to store the permutations produced by ShuffleSlices. For $\beta = l\frac{\lg M-9}{128}$, $0 \le l \le L$, he gets the penalty about (Eq. (6.2))
$$\mathcal{P}(l) = \frac{2.6 \cdot 8^l (n/32)^{L-l}}{1.5L + 2.5},$$
where we assume that to compute a permutation online he has to make $n/32$ queries to the previous layer.

- If
$$\beta > L\frac{\lg M - 9}{128},$$
then the adversary stores all the permutations in $LM\frac{\lg M-9}{128}$ memory, and spend the rest $\alpha M = (\beta - L\frac{\lg M-9}{128})$ on storing $\mathcal{F}(X_i)$ in SubGroups. If $\alpha \le 1/2$, we have the penalty (Eq. (6.3)).
$$\mathcal{P}(\alpha) = \frac{(20 - 32\alpha)8^{L-1}}{1.5L + 2.5}$$
Some examples for different α and $L = 3$ are given in Table 6.2.

Chapter 7

Design rationale

7.1 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:

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

$f_0 = y_1y_2$:	$(0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1)$ $A_3 \oplus A_7 \oplus A_{11} \oplus A_{15} \oplus A_{19} \oplus A_{23} \oplus A_{27} \oplus A_{31}$
$f_1 = y_1(y_3+1)$:	$(0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0)$ $A_1 \oplus A_3 \oplus A_9 \oplus A_{11} \oplus A_{17} \oplus A_{19} \oplus A_{25} \oplus A_{27}$
$f_2 = (y_1+1)(y_4+1)$:	$(1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0)$ $A_0 \oplus A_2 \oplus A_4 \oplus A_6 \oplus A_{16} \oplus A_{18} \oplus A_{20} \oplus A_{22}$
$f_3 = y_1y_5$:	$(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_1 \oplus A_3 \oplus A_5 \oplus A_7 \oplus A_9 \oplus A_{11} \oplus A_{13} \oplus A_{15}$
$f_4 = y_2y_3$:	$(0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1)$ $A_6 \oplus A_7 \oplus A_{14} \oplus A_{15} \oplus A_{22} \oplus A_{23} \oplus A_{30} \oplus A_{31}$
$f_5 = y_2y_4$:	$(0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1)$ $A_{10} \oplus A_{11} \oplus A_{14} \oplus A_{15} \oplus A_{26} \oplus A_{27} \oplus A_{30} \oplus A_{31}$
$f_6 = (y_2+1)y_5$:	$(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0)$ $A_{16} \oplus A_{17} \oplus A_{20} \oplus A_{21} \oplus A_{24} \oplus A_{25} \oplus A_{28} \oplus A_{29}$
$f_7 = y_3y_4$:	$(0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1)$ $A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15} \oplus A_{28} \oplus A_{29} \oplus A_{30} \oplus A_{31}$
$f_8 = y_3y_5$:	$(0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_4 \oplus A_5 \oplus A_6 \oplus A_7 \oplus A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15}$
$f_9 = (y_4+1)y_5$:	$(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0)$ $A_{16} \oplus A_{17} \oplus A_{18} \oplus A_{19} \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23}$
$f_{10} = y_1(y_2+1)$:	$(0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0)$ $A_1 \oplus A_5 \oplus A_9 \oplus A_{13} \oplus A_{17} \oplus A_{21} \oplus A_{25} \oplus A_{29}$
$f_{11} = (y_1+1)y_2$:	$(0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0)$ $A_2 \oplus A_6 \oplus A_{10} \oplus A_{14} \oplus A_{18} \oplus A_{22} \oplus A_{26} \oplus A_{30}$
$f_{12} = y_3(y_4+1)$:	$(0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0)$ $A_4 \oplus A_5 \oplus A_6 \oplus A_7 \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23}$
$f_{13} = (y_3+1)y_4$:	$(0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0)$ $A_8 \oplus A_9 \oplus A_{10} \oplus A_{11} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{27}$
$f_{14} = (y_3+1)(y_5+1)$:	$(1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$ $A_0 \oplus A_1 \oplus A_2 \oplus A_3 \oplus A_8 \oplus A_9 \oplus A_{10} \oplus A_{11}$
$f_{15} = (y_1+1)(y_2+1)$:	$(1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0)$ $A_0 \oplus A_4 \oplus A_8 \oplus A_{12} \oplus A_{16} \oplus A_{20} \oplus A_{24} \oplus A_{28}$

Table 7.1: Linearly independent combinations of 32 variables given by 16 functions of degree 2.

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 7.1. 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.

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	$\frac{7}{8}$	$\frac{11}{16}$	$\frac{1}{2}$	$\frac{11}{32}$

7.2 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. Since there are 32 slices, the entire ShuffleSlices transformation may avoid cache misses even if the total memory exceeds the cache size by a factor of 32.

The exact transformation was taken from the RC4 algorithm [12], 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. We apply a similar permutation one time to each slice of the internal state.

To summarize, the ShuffleSlices transformation has the following properties:

- It is parallelizable;
- It is non-invertible;
- It employs one of simplest data-dependent permutations;
- 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.

7.3 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 [9]. 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 [11]. We decided to take 5 rounds.

7.4 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.

Chapter 8

Efficiency analysis

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.
- The slices can be stored in the memory sequentially[1], which increases the speed when the entire slice can be loaded into the cache. We report the speed of ShuffleSlices from 4 to 8 cycles per byte.

Our implementation for $L = 3$, 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.

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.

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.

[1] The reference implementation stores groups sequentially.

Chapter 9

Change log

9.1 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 [8] to the secret part of the input, whereas it must have been called *pepper*. V1 uses the correct naming.

Bibliography

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Joan Daemen and Vincent Rijmen. *The Design of Rijndael. AES — the Advanced Encryption Standard*. Springer, 2002.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] National Institute of Standards and Technology (NIST), available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. *FIPS-197: Advanced Encryption Standard*, November 2001.

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

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