Stream: Internet Research Task Force (IRTF)
RFC: 9106
Category: Informational
Published: September 2021
ISSN: 2070-1721
Authors:
A. Biryukov, *University of Luxembourg*
D. Dinu, *University of Luxembourg*
D. Khovratovich, *ABDK Consulting*
S. Josefsson, *SJD AB*

RFC 9106 Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications

Abstract

This document describes the Argon2 memory-hard function for password hashing and proof-of-work applications. We provide an implementer-oriented description with test vectors. The purpose is to simplify adoption of Argon2 for Internet protocols. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
   - 1.1. Requirements Language
2. Notation and Conventions
3. Argon2 Algorithm
   - 3.1. Argon2 Inputs and Outputs
   - 3.2. Argon2 Operation
   - 3.3. Variable-Length Hash Function H'
   - 3.4. Indexing
     - 3.4.1. Computing the 32-Bit Values J_1 and J_2
     - 3.4.2. Mapping J_1 and J_2 to Reference Block Index [l][z]
   - 3.5. Compression Function G
   - 3.6. Permutation P
4. Parameter Choice
5. Test Vectors
   - 5.1. Argon2d Test Vectors
   - 5.2. Argon2i Test Vectors
   - 5.3. Argon2id Test Vectors
6. IANA Considerations
7. Security Considerations
   - 7.1. Security as a Hash Function and KDF
   - 7.2. Security against Time-Space Trade-off Attacks
   - 7.3. Security for Time-Bounded Defenders
   - 7.4. Recommendations
8. References
   - 8.1. Normative References
   - 8.2. Informative References

Acknowledgements

Authors' Addresses

1. Introduction

This document describes the Argon2 [ARGON2ESP] memory-hard function for password hashing and proof-of-work applications. We provide an implementer-oriented description with test vectors. The purpose is to simplify adoption of Argon2 for Internet protocols. This document corresponds to version 1.3 of the Argon2 hash function.

Argon2 is a memory-hard function [HARD]. It is a streamlined design. It aims at the highest memory-filling rate and effective use of multiple computing units, while still providing defense against trade-off attacks. Argon2 is optimized for the x86 architecture and exploits the cache and memory organization of the recent Intel and AMD processors. Argon2 has one primary variant, Argon2id, and two supplementary variants, Argon2d and Argon2i. Argon2d uses data-dependent memory access, which makes it suitable for cryptocurrencies and proof-of-work 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. Argon2id works as Argon2i for the first half of the first pass over the memory and as Argon2d for the rest, thus providing both side-channel attack protection and brute-force cost savings due to time-memory trade-offs. Argon2i makes more passes over the memory to protect from trade-off attacks [AB15].

Argon2id **MUST** be supported by any implementation of this document, whereas Argon2d and Argon2i **MAY** be supported.

Argon2 is also a mode of operation over a fixed-input-length compression function G and a variable-input-length hash function H. Even though Argon2 can be potentially used with an arbitrary function H, as long as it provides outputs up to 64 bytes, the BLAKE2b function [BLAKE2] is used in this document.

For further background and discussion, see the Argon2 paper [ARGON2].

This document represents the consensus of the Crypto Forum Research Group (CFRG).

1.1. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Notation and Conventions

x^y integer x multiplied by itself integer y times a*b multiplication of integer a and integer b

c-d subtraction of integer d from integer c

E_f variable E with subscript index f

g / h integer g divided by integer h. The result is a rational number.

I(j) function I evaluated at j

K || L string K concatenated with string L

a XOR b bitwise exclusive-or between bitstrings a and b

a mod b remainder of integer a modulo integer b, always in range [0, b-1]

a >>> n rotation of 64-bit string a to the right by n bits

trunc(a) the 64-bit value, truncated to the 32 least significant bits

floor(a) the largest integer not bigger than a

ceil(a) the smallest integer not smaller than a

extract(a, i) the i-th set of 32 bits from bitstring a, starting from 0-th

|A| the number of elements in set A

LE32(a) 32-bit integer a converted to a byte string in little endian (for example, 123456 (decimal) is 40 E2 01 00)

LE64(a) 64-bit integer a converted to a byte string in little endian (for example, 123456 (decimal) is 40 E2 01 00 00 00 00 00)

int32(s) 32-bit string s is converted to a non-negative integer in little endian

int64(s) 64-bit string s is converted to a non-negative integer in little endian

length(P) the byte length of string P expressed as 32-bit integer

ZERO(P) the P-byte zero string

3. Argon2 Algorithm

3.1. Argon2 Inputs and Outputs

Argon2 has the following input parameters:

- Message string P, which is a password for password hashing applications. It **MUST** have a length not greater than 2^(32)-1 bytes.
- Nonce S, which is a salt for password hashing applications. It **MUST** have a length not greater than 2^(32)-1 bytes. 16 bytes is **RECOMMENDED** for password hashing. The salt **SHOULD** be unique for each password.

- Degree of parallelism p determines how many independent (but synchronizing) computational chains (lanes) can be run. It **MUST** be an integer value from 1 to 2^(24)-1.
- Tag length T **MUST** be an integer number of bytes from 4 to 2^(32)-1.
- Memory size m **MUST** be an integer number of kibibytes from 8*p to 2^(32)-1. The actual number of blocks is m', which is m rounded down to the nearest multiple of 4*p.
- Number of passes t (used to tune the running time independently of the memory size) **MUST** be an integer number from 1 to 2^(32)-1.
- Version number v **MUST** be one byte 0x13.
- Secret value K is **OPTIONAL**. If used, it **MUST** have a length not greater than 2^(32)-1 bytes.
- Associated data X is **OPTIONAL**. If used, it **MUST** have a length not greater than 2^(32)-1 bytes.
- Type y **MUST** be 0 for Argon2d, 1 for Argon2i, or 2 for Argon2id.

The Argon2 output, or "tag", is a string T bytes long.

3.2. Argon2 Operation

Argon2 uses an internal compression function G with two 1024-byte inputs, a 1024-byte output, and an internal hash function H^x(), with x being its output length in bytes. Here, H^x() applied to string A is the BLAKE2b ([BLAKE2], Section 3.3) function, which takes (d,ll,kk=0,nn=x) as parameters, where d is A padded to a multiple of 128 bytes and ll is the length of d in bytes. The compression function G is based on its internal permutation. A variable-length hash function H' built upon H is also used. G is described in Section 3.5, and H' is described in Section 3.3.

The Argon2 operation is as follows.

1. Establish H_0 as the 64-byte value as shown below. If K, X, or S has zero length, it is just absent, but its length field remains.

   ```
   H_0 = H^(64)(LE32(p) || LE32(T) || LE32(m) || LE32(t) ||
           LE32(v) || LE32(y) || LE32(length(P)) || P ||
           LE32(length(S)) || S ||  LE32(length(K)) || K ||
           LE32(length(X)) || X)
   ```

   Figure 1: H_0 Generation

2. Allocate the memory as m' 1024-byte blocks, where m' is derived as:

   ```
   m' = 4 * p * floor (m / 4p)
   ```

   Figure 2: Memory Allocation

   For p lanes, the memory is organized in a matrix B[i][j] of blocks with p rows (lanes) and q = m' / p columns.

3. Compute B[i][0] for all i ranging from (and including) 0 to (not including) p.

```
B[i][0] = H'^(1024)(H_0 || LE32(0) || LE32(i))
```

Figure 3: Lane Starting Blocks

4. Compute B[i][1] for all i ranging from (and including) 0 to (not including) p.

```
B[i][1] = H'^(1024)(H_0 || LE32(1) || LE32(i))
```

Figure 4: Second Lane Blocks

5. Compute B[i][j] for all i ranging from (and including) 0 to (not including) p and for all j ranging from (and including) 2 to (not including) q. The computation **MUST** proceed slicewise (Section 3.4): first, blocks from slice 0 are computed for all lanes (in an arbitrary order of lanes), then blocks from slice 1 are computed, etc. The block indices l and z are determined for each i, j differently for Argon2d, Argon2i, and Argon2id.

```
B[i][j] = G(B[i][j-1], B[l][z])
```

Figure 5: Further Block Generation

6. If the number of passes t is larger than 1, we repeat step 5. We compute B[i][0] and B[i][j] for all i raging from (and including) 0 to (not including) p and for all j ranging from (and including) 1 to (not including) q. However, blocks are computed differently as the old value is XORed with the new one:

```
B[i][0] = G(B[i][q-1], B[l][z]) XOR B[i][0];
B[i][j] = G(B[i][j-1], B[l][z]) XOR B[i][j].
```

Figure 6: Further Passes

7. After t steps have been iterated, the final block C is computed as the XOR of the last column:

```
C = B[0][q-1] XOR B[1][q-1] XOR ... XOR B[p-1][q-1]
```

Figure 7: Final Block

8. The output tag is computed as H'^T(C).

3.3. Variable-Length Hash Function H'

Let V_i be a 64-byte block and W_i be its first 32 bytes. Then we define function H' as follows:

```
        if T <= 64
            H'^T(A) = H^T(LE32(T)||A)
        else
            r = ceil(T/32)-2
            V_1 = H^(64)(LE32(T)||A)
            V_2 = H^(64)(V_1)
            ...
            V_r = H^(64)(V_{r-1})
            V_{r+1} = H^(T-32*r)(V_{r})
            H'^T(X) = W_1 || W_2 || ... || W_r || V_{r+1}
```

Figure 8: Function H' for Tag and Initial Block Computations

3.4. Indexing

To enable parallel block computation, we further partition the memory matrix into SL = 4 vertical slices. The intersection of a slice and a lane is called a segment, which has a length of q/SL. Segments of the same slice can be computed in parallel and do not reference blocks from each other. All other blocks can be referenced.

```
    slice 0    slice 1    slice 2    slice 3
    ___/\___   ___/\___   ___/\___   ___/\___
   /        \ /        \ /        \ /        \
  +----------+----------+----------+----------+
  |          |          |          |          | > lane 0
  +----------+----------+----------+----------+
  |          |          |          |          | > lane 1
  +----------+----------+----------+----------+
  |          |          |          |          | > lane 2
  +----------+----------+----------+----------+
  |         ...        ...        ...         | ...
  +----------+----------+----------+----------+
  |          |          |          |          | > lane p - 1
  +----------+----------+----------+----------+
```

Figure 9: Single-Pass Argon2 with p Lanes and 4 Slices

3.4.1. Computing the 32-Bit Values J_1 and J_2

3.4.1.1. Argon2d

J_1 is given by the first 32 bits of block B[i][j-1], while J_2 is given by the next 32 bits of block B[i][j-1]:

```
J_1 = int32(extract(B[i][j-1], 0))
J_2 = int32(extract(B[i][j-1], 1))
```

Figure 10: Deriving J1,J2 in Argon2d

3.4.1.2. Argon2i

For each segment, we do the following. First, we compute the value Z as:

```
Z= ( LE64(r) || LE64(l) || LE64(sl) || LE64(m') ||
     LE64(t) || LE64(y) )
```

Figure 11: Input to Compute J1,J2 in Argon2i

where

r: the pass number
l: the lane number
sl: the slice number
m': the total number of memory blocks
t: the total number of passes
y: the Argon2 type (0 for Argon2d, 1 for Argon2i, 2 for Argon2id)

Then we compute:

```
q/(128*SL) 1024-byte values
G(ZERO(1024),G(ZERO(1024),
Z || LE64(1) || ZERO(968) )),
G(ZERO(1024),G(ZERO(1024),
Z || LE64(2) || ZERO(968) )),... ,
G(ZERO(1024),G(ZERO(1024),
Z || LE64(q/(128*SL)) || ZERO(968) )),
```

which are partitioned into q/(SL) 8-byte values X, which are viewed as X1||X2 and converted to J_1=int32(X1) and J_2=int32(X2).

The values r, l, sl, m', t, y, and i are represented as 8 bytes in little endian.

3.4.1.3. Argon2id

If the pass number is 0 and the slice number is 0 or 1, then compute J_1 and J_2 as for Argon2i, else compute J_1 and J_2 as for Argon2d.

3.4.2. Mapping J_1 and J_2 to Reference Block Index [l][z]

The value of l = J_2 mod p gives the index of the lane from which the block will be taken. For the first pass (r=0) and the first slice (sl=0), the block is taken from the current lane.

The set W contains the indices that are referenced according to the following rules:

1. If l is the current lane, then W includes the indices of all blocks in the last SL - 1 = 3 segments computed and finished, as well as the blocks computed in the current segment in the current pass excluding B[i][j-1].

2. If l is not the current lane, then W includes the indices of all blocks in the last SL - 1 = 3 segments computed and finished in lane l. If B[i][j] is the first block of a segment, then the very last index from W is excluded.

Then take a block from W with a nonuniform distribution over [0, |W|) using the following mapping:

```
J_1 -> |W|(1 - J_1^2 / 2^(64))
```

Figure 12: Computing J1

To avoid floating point computation, the following approximation is used:

```
x = J_1^2 / 2^(32)
y = (|W| * x) / 2^(32)
zz = |W| - 1 - y
```

Figure 13: Computing J1, Part 2

Then take the zz-th index from W; it will be the z value for the reference block index [l][z].

3.5. Compression Function G

The compression function G is built upon the BLAKE2b-based transformation P. P operates on the 128-byte input, which can be viewed as eight 16-byte registers:

```
P(A_0, A_1, ... ,A_7) = (B_0, B_1, ... ,B_7)
```

Figure 14: Blake Round Function P

The compression function G(X, Y) operates on two 1024-byte blocks X and Y. It first computes R = X XOR Y. Then R is viewed as an 8x8 matrix of 16-byte registers R_0, R_1, ... , R_63. Then P is first applied to each row, and then to each column to get Z:

```
( Q_0,  Q_1,  Q_2, ... ,  Q_7) <- P( R_0,  R_1,  R_2, ... ,  R_7)
( Q_8,  Q_9, Q_10, ... , Q_15) <- P( R_8,  R_9, R_10, ... , R_15)
                               ...
(Q_56, Q_57, Q_58, ... , Q_63) <- P(R_56, R_57, R_58, ... , R_63)
( Z_0,  Z_8, Z_16, ... , Z_56) <- P( Q_0,  Q_8, Q_16, ... , Q_56)
( Z_1,  Z_9, Z_17, ... , Z_57) <- P( Q_1,  Q_9, Q_17, ... , Q_57)
                               ...
( Z_7, Z_15, Z 23, ... , Z_63) <- P( Q_7, Q_15, Q_23, ... , Q_63)
```

Figure 15: Core of Compression Function G

Finally, G outputs Z XOR R:

```
G: (X, Y) -> R -> Q -> Z -> Z XOR R
```

```
                            +---+       +---+
                            | X |       | Y |
                            +---+       +---+
                              |           |
                              ---->XOR<----
                            --------|
                            |      \ /
                            |     +---+
                            |     | R |
                            |     +---+
                            |       |
                            |      \ /
                            |   P rowwise
                            |       |
                            |      \ /
                            |     +---+
                            |     | Q |
                            |     +---+
                            |       |
                            |      \ /
                            |  P columnwise
                            |       |
                            |      \ /
                            |     +---+
                            |     | Z |
                            |     +---+
                            |       |
                            |      \ /
                            ------>XOR
                                    |
                                   \ /
```

Figure 16: Argon2 Compression Function G

3.6. Permutation P

Permutation P is based on the round function of BLAKE2b. The eight 16-byte inputs S_0, S_1, ... , S_7 are viewed as a 4x4 matrix of 64-bit words, where S_i = (v_{2*i+1} || v_{2*i}):

```
          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
```

Figure 17: Matrix Element Labeling

It works as follows:

```
        GB(v_0, v_4,  v_8, v_12)
        GB(v_1, v_5,  v_9, v_13)
        GB(v_2, v_6, v_10, v_14)
        GB(v_3, v_7, v_11, v_15)

        GB(v_0, v_5, v_10, v_15)
        GB(v_1, v_6, v_11, v_12)
        GB(v_2, v_7,  v_8, v_13)
        GB(v_3, v_4,  v_9, v_14)
```

Figure 18: Feeding Matrix Elements to GB

GB(a, b, c, d) is defined as follows:

```
        a = (a + b + 2 * trunc(a) * trunc(b)) mod 2^(64)
        d = (d XOR a) >>> 32
        c = (c + d + 2 * trunc(c) * trunc(d)) mod 2^(64)
        b = (b XOR c) >>> 24

        a = (a + b + 2 * trunc(a) * trunc(b)) mod 2^(64)
        d = (d XOR a) >>> 16
        c = (c + d + 2 * trunc(c) * trunc(d)) mod 2^(64)
        b = (b XOR c) >>> 63
```

Figure 19: Details of GB

The modular additions in GB are combined with 64-bit multiplications. Multiplications are the only difference from the original BLAKE2b design. This choice is done to increase the circuit depth and thus the running time of ASIC implementations, while having roughly the same running time on CPUs thanks to parallelism and pipelining.

4. Parameter Choice

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

- Cryptocurrency mining, which takes 0.1 seconds on a 2 GHz CPU using 1 core -- Argon2d with 2 lanes and 250 MB of RAM.

Argon2id is optimized for more realistic settings, where the adversary can possibly access the same machine, use its CPU, or mount cold-boot attacks. We suggest the following settings:

- Backend server authentication, which takes 0.5 seconds on a 2 GHz CPU using 4 cores -- Argon2id with 8 lanes and 4 GiB of RAM.

- Key derivation for hard-drive encryption, which takes 3 seconds on a 2 GHz CPU using 2 cores -- Argon2id with 4 lanes and 6 GiB of RAM.
- Frontend server authentication, which takes 0.5 seconds on a 2 GHz CPU using 2 cores -- Argon2id with 4 lanes and 1 GiB of RAM.

We recommend the following procedure to select the type and the parameters for practical use of Argon2.

1. If a uniformly safe option that is not tailored to your application or hardware is acceptable, select Argon2id with t=1 iteration, p=4 lanes, m=2^(21) (2 GiB of RAM), 128-bit salt, and 256-bit tag size. This is the FIRST **RECOMMENDED** option.
2. If much less memory is available, a uniformly safe option is Argon2id with t=3 iterations, p=4 lanes, m=2^(16) (64 MiB of RAM), 128-bit salt, and 256-bit tag size. This is the SECOND **RECOMMENDED** option.
3. Otherwise, start with selecting the type y. If you do not know the difference between the types or you consider side-channel attacks to be a viable threat, choose Argon2id.
4. Select p=4 lanes.
5. Figure out the maximum amount of memory that each call can afford and translate it to the parameter m.
6. Figure out the maximum amount of time (in seconds) that each call can afford.
7. Select the salt length. A length of 128 bits is sufficient for all applications but can be reduced to 64 bits in the case of space constraints.
8. Select the tag length. A length of 128 bits is sufficient for most applications, including key derivation. If longer keys are needed, select longer tags.
9. If side-channel attacks are a viable threat or if you're uncertain, enable the memory-wiping option in the library call.
10. Run the scheme of type y, memory m, and p lanes using a different number of passes t. Figure out the maximum t such that the running time does not exceed the affordable time. If it even exceeds for t = 1, reduce m accordingly.
11. Use Argon2 with determined values m, p, and t.

5. Test Vectors

This section contains test vectors for Argon2.

5.1. Argon2d Test Vectors

We provide test vectors with complete outputs (tags). For the convenience of developers, we also provide some interim variables -- concretely, the first and last memory blocks of each pass.

```
=======================================
Argon2d version number 19
=======================================
Memory: 32 KiB
Passes: 3
Parallelism: 4 lanes
Tag length: 32 bytes
Password[32]: 01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
Salt[16]: 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02
Secret[8]: 03 03 03 03 03 03 03 03
Associated data[12]: 04 04 04 04 04 04 04 04 04 04 04 04
Pre-hashing digest: b8 81 97 91 a0 35 96 60
                    bb 77 09 c8 5f a4 8f 04
                    d5 d8 2c 05 c5 f2 15 cc
                    db 88 54 91 71 7c f7 57
                    08 2c 28 b9 51 be 38 14
                    10 b5 fc 2e b7 27 40 33
                    b9 fd c7 ae 67 2b ca ac
                    5d 17 90 97 a4 af 31 09

 After pass 0:
Block 0000 [  0]: db2fea6b2c6f5c8a
Block 0000 [  1]: 719413be00f82634
Block 0000 [  2]: a1e3f6dd42aa25cc
Block 0000 [  3]: 3ea8efd4d55ac0d1
...
Block 0031 [124]: 28d17914aea9734c
Block 0031 [125]: 6a4622176522e398
Block 0031 [126]: 951aa08aeecb2c05
Block 0031 [127]: 6a6c49d2cb75d5b6

 After pass 1:
Block 0000 [  0]: d3801200410f8c0d
Block 0000 [  1]: 0bf9e8a6e442ba6d
Block 0000 [  2]: e2ca92fe9c541fcc
Block 0000 [  3]: 6269fe6db177a388
...
Block 0031 [124]: 9eacfcfbdb3ce0fc
Block 0031 [125]: 07dedaeb0aee71ac
Block 0031 [126]: 074435fad91548f4
Block 0031 [127]: 2dbfff23f31b5883

 After pass 2:
Block 0000 [  0]: 5f047b575c5ff4d2
Block 0000 [  1]: f06985dbf11c91a8
Block 0000 [  2]: 89efb2759f9a8964
Block 0000 [  3]: 7486a73f62f9b142
...
Block 0031 [124]: 57cfb9d20479da49
Block 0031 [125]: 4099654bc6607f69
Block 0031 [126]: f142a1126075a5c8
Block 0031 [127]: c341b3ca45c10da5
Tag: 51 2b 39 1b 6f 11 62 97
     53 71 d3 09 19 73 42 94
```

```
       f8 68 e3 be 39 84 f3 c1
       a1 3a 4d b9 fa be 4a cb
```

5.2. Argon2i Test Vectors

```
=======================================
Argon2i version number 19
=======================================
Memory: 32 KiB
Passes: 3
Parallelism: 4 lanes
Tag length: 32 bytes
Password[32]: 01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
              01 01 01 01 01 01 01 01
Salt[16]: 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02
Secret[8]: 03 03 03 03 03 03 03 03
Associated data[12]: 04 04 04 04 04 04 04 04 04 04 04 04
Pre-hashing digest: c4 60 65 81 52 76 a0 b3
                    e7 31 73 1c 90 2f 1f d8
                    0c f7 76 90 7f bb 7b 6a
                    5c a7 2e 7b 56 01 1f ee
                    ca 44 6c 86 dd 75 b9 46
                    9a 5e 68 79 de c4 b7 2d
                    08 63 fb 93 9b 98 2e 5f
                    39 7c c7 d1 64 fd da a9

 After pass 0:
Block 0000 [  0]: f8f9e84545db08f6
Block 0000 [  1]: 9b073a5c87aa2d97
Block 0000 [  2]: d1e868d75ca8d8e4
Block 0000 [  3]: 349634174e1aebcc
...
Block 0031 [124]: 975f596583745e30
Block 0031 [125]: e349bdd7edeb3092
Block 0031 [126]: b751a689b7a83659
Block 0031 [127]: c570f2ab2a86cf00

 After pass 1:
Block 0000 [  0]: b2e4ddfcf76dc85a
Block 0000 [  1]: 4ffd0626c89a2327
Block 0000 [  2]: 4af1440fff212980
Block 0000 [  3]: 1e77299c7408505b
...
Block 0031 [124]: e4274fd675d1e1d6
Block 0031 [125]: 903fffb7c4a14c98
Block 0031 [126]: 7e5db55def471966
Block 0031 [127]: 421b3c6e9555b79d

 After pass 2:
Block 0000 [  0]: af2a8bd8482c2f11
Block 0000 [  1]: 785442294fa55e6d
Block 0000 [  2]: 9256a768529a7f96
Block 0000 [  3]: 25a1c1f5bb953766
...
Block 0031 [124]: 68cf72fccc7112b9
Block 0031 [125]: 91e8c6f8bb0ad70d
Block 0031 [126]: 4f59c8bd65cbb765
Block 0031 [127]: 71e436f035f30ed0
Tag: c8 14 d9 d1 dc 7f 37 aa
     13 f0 d7 7f 24 94 bd a1
```

```
     c8 de 6b 01 6d d3 88 d2
     99 52 a4 c4 67 2b 6c e8
```

5.3. Argon2id Test Vectors

```
=======================================
Argon2id version number 19
=======================================
Memory: 32 KiB, Passes: 3,
Parallelism: 4 lanes, Tag length: 32 bytes
Password[32]: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
Salt[16]: 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02
Secret[8]: 03 03 03 03 03 03 03 03
Associated data[12]: 04 04 04 04 04 04 04 04 04 04 04 04
Pre-hashing digest: 28 89 de 48 7e b4 2a e5 00 c0 00 7e d9 25 2f
 10 69 ea de c4 0d 57 65 b4 85 de 6d c2 43 7a 67 b8 54 6a 2f 0a
 cc 1a 08 82 db 8f cf 74 71 4b 47 2e 94 df 42 1a 5d a1 11 2f fa
 11 43 43 70 a1 e9 97

 After pass 0:
Block 0000 [  0]: 6b2e09f10671bd43
Block 0000 [  1]: f69f5c27918a21be
Block 0000 [  2]: dea7810ea41290e1
Block 0000 [  3]: 6787f7171870f893
...
Block 0031 [124]: 377fa81666dc7f2b
Block 0031 [125]: 50e586398a9c39c8
Block 0031 [126]: 6f732732a550924a
Block 0031 [127]: 81f88b28683ea8e5

 After pass 1:
Block 0000 [  0]: 3653ec9d01583df9
Block 0000 [  1]: 69ef53a72d1e1fd3
Block 0000 [  2]: 35635631744ab54f
Block 0000 [  3]: 599512e96a37ab6e
...
Block 0031 [124]: 4d4b435cea35caa6
Block 0031 [125]: c582210d99ad1359
Block 0031 [126]: d087971b36fd6d77
Block 0031 [127]: a55222a93754c692

 After pass 2:
Block 0000 [  0]: 942363968ce597a4
Block 0000 [  1]: a22448c0bdad5760
Block 0000 [  2]: a5f80662b6fa8748
Block 0000 [  3]: a0f9b9ce392f719f
...
Block 0031 [124]: d723359b485f509b
Block 0031 [125]: cb78824f42375111
Block 0031 [126]: 35bc8cc6e83b1875
Block 0031 [127]: 0b012846a40f346a
Tag: 0d 64 0d f5 8d 78 76 6c 08 c0 37 a3 4a 8b 53 c9 d0
 1e f0 45 2d 75 b6 5e b5 25 20 e9 6b 01 e6 59
```

6. IANA Considerations

This document has no IANA actions.

7. Security Considerations

7.1. Security as a Hash Function and KDF

The collision and preimage resistance levels of Argon2 are equivalent to those of the underlying BLAKE2b hash function. To produce a collision, 2^(256) inputs are needed. To find a preimage, 2^(512) inputs must be tried.

The KDF security is determined by the key length and the size of the internal state of hash function H'. To distinguish the output of the keyed Argon2 from random, a minimum of (2^(128),2^length(K)) calls to BLAKE2b are needed.

7.2. Security against Time-Space Trade-off Attacks

Time-space trade-offs allow computing a memory-hard function storing fewer memory blocks at the cost of more calls to the internal compression function. The advantage of trade-off attacks is measured in the reduction factor to the time-area product, where memory and extra compression function cores contribute to the area and time is increased to accommodate the recomputation of missed blocks. A high reduction factor may potentially speed up the preimage search.

The best-known attack on the 1-pass and 2-pass Argon2i is the low-storage attack described in [CBS16], which reduces the time-area product (using the peak memory value) by the factor of 5. The best attack on Argon2i with 3 passes or more is described in [AB16], with the reduction factor being a function of memory size and the number of passes (e.g., for 1 gibibyte of memory, a reduction factor of 3 for 3 passes, 2.5 for 4 passes, 2 for 6 passes). The reduction factor grows by about 0.5 with every doubling of the memory size. To completely prevent time-space trade-offs from [AB16], the number of passes **MUST** exceed the binary logarithm of memory minus 26. Asymptotically, the best attack on 1-pass Argon2i is given in [BZ17], with maximal advantage of the adversary upper bounded by O(m^(0.233)), where m is the number of blocks. This attack is also asymptotically optimal as [BZ17] also proves the upper bound on any attack is O(m^(0.25)).

The best trade-off attack on t-pass Argon2d is the ranking trade-off attack, which reduces the time-area product by the factor of 1.33.

The best attack on Argon2id can be obtained by complementing the best attack on the 1-pass Argon2i with the best attack on a multi-pass Argon2d. Thus, the best trade-off attack on 1-pass Argon2id is the combined low-storage attack (for the first half of the memory) and the ranking attack (for the second half), which generate the factor of about 2.1. The best trade-off attack on t-pass Argon2id is the ranking trade-off attack, which reduces the time-area product by the factor of 1.33.

7.3. Security for Time-Bounded Defenders

A bottleneck in a system employing the password hashing function is often the function latency rather than memory costs. A rational defender would then maximize the brute-force costs for the attacker equipped with a list of hashes, salts, and timing information for fixed computing time on the defender's machine. The attack cost estimates from [AB16] imply that for Argon2i, 3 passes is almost optimal for most reasonable memory sizes; for Argon2d and Argon2id, 1 pass maximizes the attack costs for the constant defender time.

7.4. Recommendations

The Argon2id variant with t=1 and 2 GiB memory is the FIRST **RECOMMENDED** option and is suggested as a default setting for all environments. This setting is secure against side-channel attacks and maximizes adversarial costs on dedicated brute-force hardware. The Argon2id variant with t=3 and 64 MiB memory is the SECOND **RECOMMENDED** option and is suggested as a default setting for memory-constrained environments.

8. References

8.1. Normative References

[BLAKE2] Saarinen, M-J., Ed. and J-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC 7693, DOI 10.17487/RFC7693, November 2015, <https://www.rfc-editor.org/info/rfc7693>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[AB15] Biryukov, A. and D. Khovratovich, "Tradeoff Cryptanalysis of Memory-Hard Functions", ASIACRYPT 2015, DOI 10.1007/978-3-662-48800-3_26, December 2015, <https://eprint.iacr.org/2015/227.pdf>.

[AB16] Alwen, J. and J. Blocki, "Efficiently Computing Data-Independent Memory-Hard Functions", CRYPTO 2016, DOI 10.1007/978-3-662-53008-5_9, March 2016, <https://eprint.iacr.org/2016/115.pdf>.

[ARGON2] Biryukov, A., Dinu, D., and D. Khovratovich, "Argon2: the memory-hard function for password hashing and other applications", March 2017, <https://www.cryptolux.org/images/0/0d/Argon2.pdf>.

[ARGON2ESP] Biryukov, A., Dinu, D., and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications", Euro SnP 2016, DOI 10.1109/EuroSP.2016.31, March 2016, <https://www.cryptolux.org/images/d/d0/Argon2ESP.pdf>.

[BZ17] Blocki, J. and S. Zhou, "On the Depth-Robustness and Cumulative Pebbling Cost of Argon2i", TCC 2017, DOI 10.1007/978-3-319-70500-2_15, May 2017, <https://eprint.iacr.org/2017/442.pdf>.

[CBS16] Boneh, D., Corrigan-Gibbs, H., and S. Schechter, "Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks", ASIACRYPT 2016, DOI 10.1007/978-3-662-53887-6_8, May 2017, <https://eprint.iacr.org/2016/027.pdf>.

[HARD] Alwen, J. and V. Serbinenko, "High Parallel Complexity Graphs and Memory-Hard Functions", STOC '15, DOI 10.1145/2746539.2746622, June 2015, <https://eprint.iacr.org/2014/238.pdf>.

Acknowledgements

We greatly thank the following individuals who helped in preparing and reviewing this document: Jean-Philippe Aumasson, Samuel Neves, Joel Alwen, Jeremiah Blocki, Bill Cox, Arnold Reinhold, Solar Designer, Russ Housley, Stanislav Smyshlyaev, Kenny Paterson, Alexey Melnikov, and Gwynne Raskind.

The work described in this document was done before Daniel Dinu joined Intel, while he was at the University of Luxembourg.

Authors' Addresses

Alex Biryukov
University of Luxembourg
Email: alex.biryukov@uni.lu

Daniel Dinu
University of Luxembourg
Email: daniel.dinu@intel.com

Dmitry Khovratovich
ABDK Consulting
Email: khovratovich@gmail.com

Simon Josefsson
SJD AB
Email: simon@josefsson.org
URI: http://josefsson.org/