Network Working Group A. Biryukov
Internet-Draft D. Dinu
Intended status: Informational D. Khovratovich
Expires: September 26, 2017 University of Luxembourg
S. Josefsson
SJD AB
March 25, 2017

The memory-hard Argon2 password hash and proof-of-work function
draft-irtf-cfrg-argon2-02

Abstract

This document describes the Argon2 memory-hard function for password hashing and proof-of-work applications. We provide an implementer oriented description together with sample code and test vectors. The purpose is to simplify adoption of Argon2 for Internet protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 26, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Notation and Conventions 3
3. Argon2 Algorithm 3
3.1. Argon2 Inputs and Outputs 4
3.2. Argon2 Operation 4
3.3. Variable-length hash function H' 6
3.4. Indexing 6
3.4.1. Getting the 32-bit values J_1 and J_2 7
3.4.2. Mapping J_1 and J_2 to reference block index 7
3.5. Compression function G 8
3.6. Permutation P 9
4. Parameter Choice 10
5. Example Code 11
6. Test Vectors 20
6.1. Argon2d Test Vectors 20
6.2. Argon2i Test Vectors 21
6.3. Argon2id Test Vectors 22
7. Acknowledgements 24
8. IANA Considerations 24
9. Security Considerations 24
9.1. Security as hash function and KDF 24
9.2. Security against time-space tradeoff attacks 24
9.3. Security for time-bounded defenders 25
9.4. Recommendations 25
10. References 25
10.1. Normative References 25
10.2. Informative References 25
Authors' Addresses 26

1. Introduction

This document describes the Argon2 memory-hard function for password hashing and proof-of-work applications. We provide an implementer oriented description together with sample code and 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 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 one primary variant: Argon2id, and two supplementary variants: Argon2d and Argon2i. Argon2d uses data-depending 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 iteration 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 tradeoffs. Argon2i makes more passes over the memory to protect from tradeoff attacks.

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

2. Notation and Conventions

x**y --- x multiplied by itself y times

a*b --- multiplication of a and b

c-d --- substraction of c with d

E_f --- variable E with subscript index f

g / h --- g divided by h

I(j) --- function I evaluated on parameter j

K || L --- string K concatenated with string L

a ^ b --- bitwise exclusive-or between a and b

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

a >>> n --- rotation of a to the right by n bits

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

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

|A| --- the number of elements in set A

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. May have any length from 0 to 2**32 - 1 bytes.

- Nonce S, which is a salt for password hashing applications. May have any length from 8 to 2**32-1 bytes. 16 bytes is recommended for password hashing. Salt must be unique for each password.

- Degree of parallelism p determines how many independent (but synchronizing) computational chains (lanes) can be run. It may take any integer value from 1 to 2**24-1.

- Tag length T may be any integer number of bytes from 4 to 2**32-1.

- Memory size m can be any 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 iterations t (used to tune the running time independently of the memory size) can be any integer number from 1 to 2**32-1.

- Version number v is one byte 0x13.

- Secret value K (serves as key if necessary, but we do not assume any key use by default) may have any length from 0 to 2**32-1 bytes.

- Associated data X may have any length from 0 to 2**32-1 bytes.

- Type y of Argon2: 0 for Argon2d, 1 for Argon2i, 2 for Argon2id.

The Argon2 output is a T-length string.

3.2. Argon2 Operation

Argon2 uses an internal compression function G with two 1024-byte inputs and a 1024-byte output, and an internal hash function H. Here H is the BLAKE2b [I-D.saarinen-blake2] hash function, and the compression function G is based on its internal permutation. A variable-length hash function H' built upon H is also used. G and H' are described in later section.

The Argon2 operation is as follows.

1. Establish H_0 as the 64-bit value as shown in the figure below. H is BLAKE2b and the non-strings p, T, m, t, v, y, length(P), length(S), length(K), and length(X) are treated as a 32-bit little-endian encoding of the integer.

```
H_0 = H(p, T, m, t, v, y, length(P), P, length(S), S,
        length(K), K, length(X), X)
```

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

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

   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'(H_0, 0, i)
```

   Here integers are padded to 4 bytes and encoded in little endian.

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

```
B[i][1] = H'(H_0, 1, i)
```

   Here integers are padded to 4 bytes and encoded in little endian.

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 block indices i' and j' are determined differently for Argon2d, Argon2i, and Argon2id.

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

6. If the number of iterations t is larger than 1, we repeat the steps however replacing the computations with the following expression:

```
B[i][0] = G(B[i][q-1], B[i'][j']) XOR B[i][0]
B[i][j] = G(B[i][j-1], B[i'][j']) XOR B[i][j]
```

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]
```

8. The output tag is computed as H'(C).

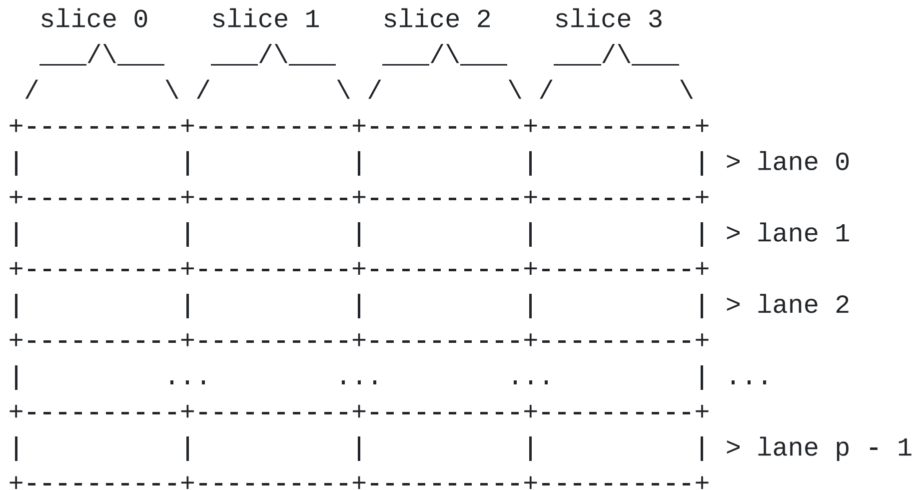
3.3. Variable-length hash function H'

Let H_x be a hash function with x-byte output (in our case H_x is BLAKE2b, which supports x between 1 and 64 inclusive). Let V_i be a 64-byte block, and A_i be its first 32 bytes, and T < 2**32 be the tag length in bytes, encoded in little-endian as 32-bit integer. Then we define:

```
if T <= 64
    H'(X) = H_T(T||X)
else
    r = ceil(T/32)-2
    V_1 = H_64(T||X)
    V_2 = H_64(V_1)
    ...
    V_r = H_64(V_{r-1})
    V_{r+1} = H_{T-32*r}(V_{r})
    H'(X) = A_1 || A_2 || ... || A_r || V_{r+1}
```

3.4. Indexing

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



Single-pass Argon2 with p lanes and 4 slices

3.4.1. Getting 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 = extract(B[i][j-1], 1)
J_2 = extract(B[i][j-1], 2)
```

3.4.1.2. Argon2i

Each application of the 2-round compression function G in the counter mode gives 128 64-bit values J_1 || J_2. The first input is the all zero block and the second input is constructed as follows:

```
( r || l || s || m' || t || x || i || 0 ), where

r  -- the pass number
l  -- the lane number
s  -- the slice number
m' -- the total number of memory blocks
t  -- the total number of passes
x  -- the Argon2 type (0 for Argon2d,  1 for Argon2i, 2 for Argon2id)
i  -- the counter (starts from 1 in each segment)
```

The values r, l, s, m', t, x, i are represented on 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

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 (s=0) the block is taken from the current lane.

The set R contains the indices that can be referenced according to the following rules:

1. If l is the current lane, then R includes the indices of all blocks in the last S - 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 R includes the indices of all blocks in the last S - 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 R is excluded.

We are going to take a block from R with a non-uniform distribution over [0, |R|):

```
J_1 in [0, 2**32) -> |R|(1 - J_1**2 / 2**64)
```

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

```
x = J_1**2 / 2**32
y = (|R| * x) / 2**32
z = |R| - 1 - y
```

The value of z gives the reference block index in R.

3.5. Compression function G

Compression function G is built upon the BLAKE2b round function P. P operates on the 128-byte input, which can be viewed as 8 16-byte registers:

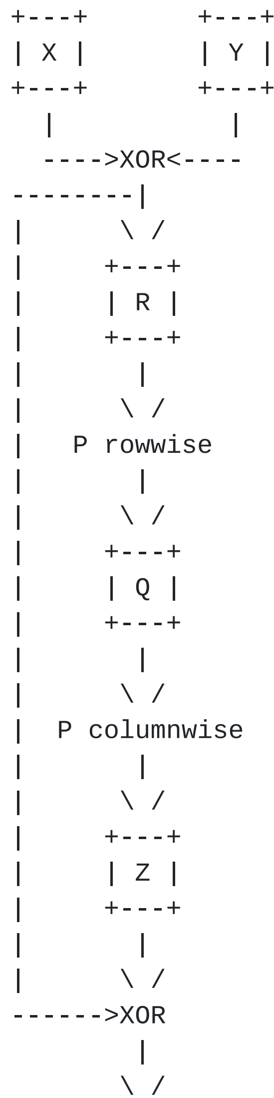
```
P(A_0, A_1, ... ,A_7) = (B_0, B_1, ... ,B_7)
```

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 a 8x8 matrix of 16-byte registers R_0, R_1, ... , R_63. Then P is first applied rowwise, and then columnwise 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)
```

Finally, G outputs Z XOR R:

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



Argon2 compression function G.

3.6. Permutation P

Permutation P is based on the round function of BLAKE2b. The 8 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
```

It works as follows:

```
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)
```

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

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

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

The modular additions in G are combined with 64-bit multiplications. Multiplications are the only difference to 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. he can not run side-channel attacks based on the timing information, nor he can 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:

- o Cryptocurrency mining, that 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 possibly can access the same machine, use its CPU or mount cold-boot attacks. We suggest the following settings:

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

- Key derivation for hard-drive encryption, that takes 3 seconds on a 2 GHz CPU using 2 cores - Argon2id with 4 lanes and 6 GB of RAM.

- Frontend server authentication, that takes 0.5 seconds on a 2 GHz CPU using 2 cores - Argon2id with 4 lanes and 1 GB of RAM.

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

1. Select the type y. If you do not know the difference between them or you consider side-channel attacks as viable threat, choose Argon2id.

2. Figure out the maximum number h of threads that can be initiated by each call to Argon2.

3. Figure out the maximum amount m of memory that each call can afford.

4. Figure out the maximum amount x of time (in seconds) that each call can afford.

5. Select the salt length. 128 bits is sufficient for all applications, but can be reduced to 64 bits in the case of space constraints.

6. Select the tag length. 128 bits is sufficient for most applications, including key derivation. If longer keys are needed, select longer tags.

7. If side-channel attacks is a viable threat, enable the memory wiping option in the library call.

8. Run the scheme of type y, memory m and h lanes and threads, using different number of passes t. Figure out the maximum t such that the running time does not exceed x. If it exceeds x even for t = 1, reduce m accordingly.

9. Hash all the passwords with the just determined values m, h, and t.

5. Example Code

```
void fill_block(const block *prev_block,
                const block *ref_block,
                block *next_block) {
  block blockR, block_tmp;
  unsigned i;

  copy_block(&blockR, ref_block);
  xor_block(&blockR, prev_block);
  copy_block(&block_tmp, &blockR);

  /* Now blockR = ref_block + prev_block and bloc_tmp = ref_block +
     prev_block */

  /* Apply Blake2 on columns of 64-bit words: (0,1,...,15),
     then (16,17,..31)... finally (112,113,...127) */
  for (i = 0; i < 8; ++i) {
    BLAKE2_ROUND_NOMSG(
      blockR.v[16 * i], blockR.v[16 * i + 1],
      blockR.v[16 * i + 2], blockR.v[16 * i + 3],
      blockR.v[16 * i + 4], blockR.v[16 * i + 5],
      blockR.v[16 * i + 6], blockR.v[16 * i + 7],
      blockR.v[16 * i + 8], blockR.v[16 * i + 9],
      blockR.v[16 * i + 10], blockR.v[16 * i + 11],
      blockR.v[16 * i + 12], blockR.v[16 * i + 13],
      blockR.v[16 * i + 14], blockR.v[16 * i + 15]);
  }

  /* Apply Blake2 on rows of 64-bit words: (0,1,16,17,...112,113),
     then (2,3,18,19,...,114,115), ... and finally
     (14,15,30,31,...,126,127) */
  for (i = 0; i < 8; i++) {
    BLAKE2_ROUND_NOMSG(
      blockR.v[2 * i], blockR.v[2 * i + 1],
      blockR.v[2 * i + 16], blockR.v[2 * i + 17],
      blockR.v[2 * i + 32], blockR.v[2 * i + 33],
      blockR.v[2 * i + 48], blockR.v[2 * i + 49],
      blockR.v[2 * i + 64], blockR.v[2 * i + 65],
      blockR.v[2 * i + 80], blockR.v[2 * i + 81],
      blockR.v[2 * i + 96], blockR.v[2 * i + 97],
      blockR.v[2 * i + 112], blockR.v[2 * i + 113]);
  }

  copy_block(next_block, &block_tmp);
  xor_block(next_block, &blockR);
}
```

```
void fill_block_with_xor(const block *prev_block,
                         const block *ref_block,
                         block *next_block) {
  block blockR, block_tmp;
  unsigned i;

  copy_block(&blockR, ref_block);
  xor_block(&blockR, prev_block);
  copy_block(&block_tmp, &blockR);

  /* Saving the next block contents for XOR over */
  xor_block(&block_tmp, next_block);

  /* Now blockR = ref_block + prev_block and bloc_tmp = ref_block +
     prev_block + next_block*/
  /* Apply Blake2 on columns of 64-bit words: (0,1,...,15) , then
     (16,17,..31),... and finally (112,113,...127) */
  for (i = 0; i < 8; ++i) {
    BLAKE2_ROUND_NOMSG(
      blockR.v[16 * i], blockR.v[16 * i + 1],
      blockR.v[16 * i + 2], blockR.v[16 * i + 3],
      blockR.v[16 * i + 4], blockR.v[16 * i + 5],
      blockR.v[16 * i + 6], blockR.v[16 * i + 7],
      blockR.v[16 * i + 8], blockR.v[16 * i + 9],
      blockR.v[16 * i + 10], blockR.v[16 * i + 11],
      blockR.v[16 * i + 12], blockR.v[16 * i + 13],
      blockR.v[16 * i + 14], blockR.v[16 * i + 15]);
    }

  /* Apply Blake2 on rows of 64-bit words:
     (0,1,16,17,...112,113), then
     (2,3,18,19,...,114,115), ... and finally
     (14,15,30,31,...,126,127) */
  for (i = 0; i < 8; i++) {
    BLAKE2_ROUND_NOMSG(
      blockR.v[2 * i], blockR.v[2 * i + 1],
      blockR.v[2 * i + 16], blockR.v[2 * i + 17],
      blockR.v[2 * i + 32], blockR.v[2 * i + 33],
      blockR.v[2 * i + 48], blockR.v[2 * i + 49],
      blockR.v[2 * i + 64], blockR.v[2 * i + 65],
      blockR.v[2 * i + 80], blockR.v[2 * i + 81],
      blockR.v[2 * i + 96], blockR.v[2 * i + 97],
      blockR.v[2 * i + 112], blockR.v[2 * i + 113]);
  }

  copy_block(next_block, &block_tmp);
  xor_block(next_block, &blockR);
}
```

```
void generate_addresses(const argon2_instance_t *instance,
                        const argon2_position_t *position,
                        uint64_t *pseudo_rands) {
  block zero_block, input_block, address_block,tmp_block;
  uint32_t i;

  init_block_value(&zero_block, 0);
  init_block_value(&input_block, 0);

  if (instance != NULL && position != NULL) {
    input_block.v[0] = position->pass;
    input_block.v[1] = position->lane;
    input_block.v[2] = position->slice;
    input_block.v[3] = instance->memory_blocks;
    input_block.v[4] = instance->passes;
    input_block.v[5] = instance->type;

    for (i = 0; i < instance->segment_length; ++i) {
      if (i % ARGON2_ADDRESSES_IN_BLOCK == 0) {
        input_block.v[6]++;
        init_block_value(&tmp_block, 0);
        init_block_value(&address_block, 0);
        fill_block_with_xor(&zero_block, &input_block, &tmp_block);
        fill_block_with_xor(&zero_block, &tmp_block, &address_block);
    }

    pseudo_rands[i] = address_block.v[i % ARGON2_ADDRESSES_IN_BLOCK];
  }
}

void fill_segment(const argon2_instance_t *instance,
                  argon2_position_t position) {
  block *ref_block = NULL, *curr_block = NULL;
  uint64_t pseudo_rand, ref_index, ref_lane;
  uint32_t prev_offset, curr_offset;
  uint32_t starting_index;
  uint32_t i;
  int data_independent_addressing;

  /* Pseudo-random values that determine the reference block
     position */
  uint64_t *pseudo_rands = NULL;

  if (instance == NULL) {
    return;
  }

  data_independent_addressing = (instance->type == Argon2_i);
```

```
     pseudo_rands = (uint64_t *)malloc(sizeof(uint64_t) *
                                   (instance->segment_length));

     if (pseudo_rands == NULL) {
       return;
     }

     if (data_independent_addressing) {
       generate_addresses(instance, &position, pseudo_rands);
     }

     starting_index = 0;

     if ((0 == position.pass) && (0 == position.slice)) {
       /* we have already generated the first two blocks */
       starting_index = 2;
     }

     /* Offset of the current block */
     curr_offset = position.lane * instance->lane_length +
                   position.slice * instance->segment_length +
                   starting_index;

     if (0 == curr_offset % instance->lane_length) {
       /* Last block in this lane */
       prev_offset = curr_offset + instance->lane_length - 1;
     } else {
       /* Previous block */
       prev_offset = curr_offset - 1;
     }

     for (i = starting_index; i < instance->segment_length;
          ++i, ++curr_offset, ++prev_offset) {
       /*1.1 Rotating prev_offset if needed */
       if (curr_offset % instance->lane_length == 1) {
         prev_offset = curr_offset - 1;
       }

       /* 1.2 Computing the index of the reference block */
       /* 1.2.1 Taking pseudo-random value from the previous block */
       if (data_independent_addressing) {
         pseudo_rand = pseudo_rands[i];
        } else {
          pseudo_rand = instance->memory[prev_offset].v[0];
       }

       /* 1.2.2 Computing the lane of the reference block */
       ref_lane = ((pseudo_rand >> 32)) % instance->lanes;
```

```
        if ((position.pass == 0) && (position.slice == 0)) {
           /* Can not reference other lanes yet */
           ref_lane = position.lane;
        }

        /* 1.2.3 Computing the number of possible reference block
           within the lane. */
        position.index = i;
        ref_index = index_alpha(instance, &position,
                                pseudo_rand & 0xFFFFFFFF,
                                ref_lane == position.lane);

        /* 2 Creating a new block */
        ref_block = instance->memory +
                    instance->lane_length * ref_lane + ref_index;
        curr_block = instance->memory + curr_offset;
        if (instance->version == ARGON2_OLD_VERSION_NUMBER) {
          /* version 1.2.1 and earlier: overwrite, not XOR */
          fill_block(instance->memory + prev_offset, ref_block,
                     curr_block);
        } else {
          if(0 == position.pass) {
            fill_block(instance->memory + prev_offset, ref_block,
                       curr_block);
          } else {
            fill_block_with_xor(instance->memory + prev_offset,
                                ref_block, curr_block);
          }
        }
      }

      free(pseudo_rands);
    }

  uint32_t index_alpha(const argon2_instance_t *instance,
                       const argon2_position_t *position,
                       uint32_t pseudo_rand,
                       int same_lane) {
    /*
     * Pass 0:
     *  This lane : all already finished segments plus already
     *    constructed blocks in this segment
     *      Other lanes : all already finished segments
     * Pass 1+:
     *      This lane : (SYNC_POINTS - 1) last segments plus
     *        already constructed blocks in this segment
     *      Other lanes : (SYNC_POINTS - 1) last segments
     */
```

```
    uint32_t reference_area_size;
    uint64_t relative_position;
    uint32_t start_position, absolute_position;

    if (0 == position->pass) {
      /* First pass */
      if (0 == position->slice) {
        /* First slice */
        reference_area_size =
        position->index - 1; /* all but the previous */
      } else {
        if (same_lane) {
          /* The same lane => add current segment */
          reference_area_size = position->slice *
                                instance->segment_length +
                                position->index - 1;
        } else {
          reference_area_size = position->slice *
                                instance->segment_length +
                                ((position->index == 0) ? (-1) : 0);
        }
      }
    } else {
      /* Second pass */
      if (same_lane) {
        reference_area_size = instance->lane_length -
                              instance->segment_length +
                              position->index - 1;
      } else {
        reference_area_size = instance->lane_length -
                              instance->segment_length +
                              ((position->index == 0) ? (-1) : 0);
      }
    }

    /* 1.2.4. Mapping pseudo_rand to 0..<reference_area_size-1>
       and produce relative position */
    relative_position = pseudo_rand;
    relative_position = relative_position * relative_position >> 32;
    relative_position = reference_area_size - 1 -
                        (reference_area_size * relative_position >> 32);

    /* 1.2.5 Computing starting position */
    start_position = 0;

    if (0 != position->pass) {
      start_position = (position->slice == ARGON2_SYNC_POINTS - 1)
                        ? 0
```

```
                        : (position->slice + 1) *
                        instance->segment_length;
    }

    /* 1.2.6. Computing absolute position */
    absolute_position = (start_position + relative_position) %
                         instance->lane_length; /* absolute position */
    return absolute_position;
  }

int fill_memory_blocks(argon2_instance_t *instance) {
  uint32_t r, s;
  argon2_thread_handle_t *thread = NULL;
  argon2_thread_data *thr_data = NULL;

  if (instance == NULL || instance->lanes == 0) {
    return ARGON2_THREAD_FAIL;
  }

  /* 1. Allocating space for threads */
  thread = calloc(instance->lanes, sizeof(argon2_thread_handle_t));
  if (thread == NULL) {
    return ARGON2_MEMORY_ALLOCATION_ERROR;
  }

  thr_data = calloc(instance->lanes, sizeof(argon2_thread_data));
  if (thr_data == NULL) {
    free(thread);
    return ARGON2_MEMORY_ALLOCATION_ERROR;
  }

  for (r = 0; r < instance->passes; ++r) {
    for (s = 0; s < ARGON2_SYNC_POINTS; ++s) {
      int rc;
      uint32_t l;

      /* 2. Calling threads */
      for (l = 0; l < instance->lanes; ++l) {
        argon2_position_t position;

        /* 2.1 Join a thread if limit is exceeded */
        if (l >= instance->threads) {
          rc = argon2_thread_join(thread[l - instance->threads]);
          if (rc) {
            free(thr_data);
            free(thread);
            return ARGON2_THREAD_FAIL;
          }
```

```
        }

        /* 2.2 Create thread */
        position.pass = r;
        position.lane = l;
        position.slice = (uint8_t)s;
        position.index = 0;
        /* preparing the thread input */
        thr_data[l].instance_ptr = instance;
        memcpy(&(thr_data[l].pos), &position,
               sizeof(argon2_position_t));
        rc = argon2_thread_create(&thread[l], &fill_segment_thr,
                                  (void *)&thr_data[l]);
        if (rc) {
          free(thr_data);
          free(thread);
          return ARGON2_THREAD_FAIL;
        }

        /* fill_segment(instance, position); */
        /*Non-thread equivalent of the lines above */
      }

      /* 3. Joining remaining threads */
      for (l = instance->lanes - instance->threads; l < instance->lanes;
           ++l) {
        rc = argon2_thread_join(thread[l]);
        if (rc) {
          return ARGON2_THREAD_FAIL;
        }
      }
    }
  }

  if (thread != NULL) {
    free(thread);
  }
  if (thr_data != NULL) {
    free(thr_data);
  }

  return ARGON2_OK;
}
```

6. Test Vectors

This section contains test vectors for Argon2.

6.1. Argon2d Test Vectors

```
=======================================
Argon2d version number 19
=======================================
Memory: 32 KiB
Iterations: 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
```

6.2. Argon2i Test Vectors

```
=======================================
Argon2i version number 19
=======================================
Memory: 32 KiB
Iterations: 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
```

6.3. Argon2id Test Vectors

```
          =======================================
Argon2id version number 19
=======================================
Memory: 32 KiB, Iterations: 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
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
```

7. Acknowledgements

TBA

8. IANA Considerations

None.

9. Security Considerations

9.1. Security as 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 keyed Argon2 from random, minimum of (2**128,2**length(K)) calls to Blake2b is needed.

9.2. Security against time-space tradeoff attacks

Time-space tradeoffs allow computing a memory-hard function storing fewer memory blocks at the cost of more calls to the internal comression function. The advantage of tradeoff 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 accomodate the recomputation of missed blocks. A high reduction factor may potentially speed up preimage search.

The best attacks 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 3-pass and more Argon2i is [AB16] with reduction factor being a function of memory size and the number of passes. For 1 GiB of memory: 3 for 3 passes, 2.5 for 4 passes, 2 for 6 passes. The reduction factor grows by about 0.5 with every doubling the memory size. To completely prevent time-space tradeoffs from [AB16], number t of passes must exceed binary logarithm of memory minus 26.

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

The best tradeoff 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 bring together the factor of

about 2.1. The best tradeoff attack on t-pass Argon2d is the ranking tradeoff attack, which reduces the time-area product by the factor of 1.33.

9.3. 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 bruteforce 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 the most of reasonable memory sizes, and that for Argon2d and Argon2id 1 pass maximizes the attack costs for the constant defender time.

9.4. Recommendations

The Argon2id variant with t=1 and maximum available memory is recommended as a default setting for all environments. This setting is secure against side-channel attacks and maximizes adversarial costs on dedicated bruteforce hardware.

10. References

10.1. Normative References

[I-D.saarinen-blake2]
Saarinen, M. and J. Aumasson, "The BLAKE2 Cryptographic Hash and MAC", draft-saarinen-blake2-06 (work in progress), August 2015.

10.2. Informative References

[ARGON2] Biryukov, A., Dinu, D., and D. Khovratovich, "Argon2: the memory-hard function for password hashing and other applications", WWW <https://www.cryptolux.org/images/0/0d/Argon2.pdf>, October 2015.

[CBS16] Corrigan-Gibbs, H., Boneh, D., and S. Schechter, "Balloon Hashing: Provably Space-Hard Hash Functions with Data-Independent Access Patterns", WWW <https://eprint.iacr.org/2016/027.pdf>, January 2016.

[AB16] Alwen, J. and J. Blocki, "Efficiently Computing Data-Independent Memory-Hard Functions", WWW <https://eprint.iacr.org/2016/115.pdf>, December 2015.

[AB15] Biryukov, A. and D. Khovratovich, "Tradeoff Cryptanalysis of Memory-Hard Functions", Asiacrypt'15 <https://eprint.iacr.org/2015/227.pdf>, December 2015.

Authors' Addresses

Alex Biryukov
University of Luxembourg

Email: alex.biryukov@uni.lu

Daniel Dinu
University of Luxembourg

Email: dumitru-daniel.dinu@uni.lu

Dmitry Khovratovich
University of Luxembourg

Email: dmitry.khovratovich@uni.lu

Simon Josefsson
SJD AB

Email: simon@josefsson.org
URI: http://josefsson.org/