Network Working Group A. Biryukov
Internet-Draft D. Dinu
Intended status: Informational D. Khovratovich
Expires: February 4, 2018 University of Luxembourg
S. Josefsson
SJD AB
August 03, 2017

The memory-hard Argon2 password hash and proof-of-work function
draft-irtf-cfrg-argon2-03

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

Copyright Notice

Table of Contents

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-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 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.

Argon2 can be viewed as 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 arbitrary function H, as long as it provides outputs up to 64 bytes, in this document it MUST be BLAKE2b.

For further background and discussion, see the Argon2 paper [ARGON2ESP] and specifications [ARGON2].

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 --- substraction of integer c with integer d

E_f --- variable E with subscript index f

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

I(j) --- function I evaluated on integer parameter 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 a 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 bytestring in little endian. Example: 123456 (decimal) is 40 E2 01 00.

LE64(a) --- 64-bit integer a converted to bytestring in little endian. Example: 123456 (decimal) is 40 E2 01 00.

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

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

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

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 string T bytes long.

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^x(). Here H^x() applied to string a is the BLAKE2b [RFC7693](a,|a|,0,x) 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 is described in Section Section 3.5 and H' is described in Section Section 3.3.

The Argon2 operation is as follows.

1. Establish H_0 as the 64-bit value as shown below.

```
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)
```

H_0 generation

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

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

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'(H_0 || LE32(0) || LE32(i))
```

Lane starting blocks

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

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

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 block indices i' and j' are determined for each i, j differently for Argon2d, Argon2i, and Argon2id (Section Section 3.4).

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

Further block generation

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

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]
```

Final block

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

3.3. Variable-length hash function H'

Recall that H^x(a) is BLAKE2b(a,|a|,0,x). Let V_i be a 64-byte block, and A_i be its first 32 bytes. Then we define:

```
if T <= 64
    H'(X) = H^T(T||X)
else
    r = ceil(T/32)-2
    V_1 = H^(64)(LE32(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}
```

Tag computation

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.

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

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 = int32(extract(B[i][j-1], 1))
J_2 = int32(extract(B[i][j-1], 2))
```

Deriving J1,J2 in Argon2d

3.4.1.2. Argon2i

Each application of the 2-round compression function G in the counter mode gives 128 64-bit values X, which are viewed as X1||X2 and converted to J_1=int32(X1) and J_2=int32(X2). The first input to G is the all zero block and the second input to G is constructed as follows:

```
( LE64(r) || LE64(l) || LE64(s) || LE64(m') || LE64(t) ||
    LE64(y) || LE64(i) || ZERO ), 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
y  -- the Argon2 type:
        - 0 for Argon2d
        - 1 for Argon2i
        - 2 for Argon2id
i  -- the counter (starts from 1 in each segment)
ZERO -- the 968-byte zero string.
```

Input to compute J1,J2 in Argon2i

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 firt pass (r=0) and the first slice (s=0) the block is taken from the current lane.

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

1. If l is the current lane, then W 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 W 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 W is excluded.

We are going to take a block from W with a non-uniform distribution over [0, |W|) using the mapping

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

Computing J1

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

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

Computing J1, part 2

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

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)
```

Blake round function P

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 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)
```

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
        |
       \ /
```

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
```

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)
```

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
```

Details of GB

The modular additions in GB 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 GiB of RAM.

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

- o Frontend server authentication, that 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. 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

5.1. argon2.h

This example code is a compact reference implementation of Argon2. Features such as input validation or secure memory wiping are not included in this code. It does not include the BLAKE2b implementation, but only the code that has to be added to the official BLAKE2b implementation in oreder to support Argon2. A complete implementation can be found in the official Argon2 repository [ARGON2GITHUB].

```
#ifndef ARGON2_H
#define ARGON2_H

#include <stdint.h>
#include <stddef.h>
#include <limits.h>

/* Number of synchronization points between lanes */
#define ARGON2_SYNC_POINTS UINT32_C(4)

/* Return codes */
#define ARGON2_OK     0
#define ARGON2_ERROR -1

/* Argon2 internal structure holds Argon2 inputs */
typedef struct Argon2_Context {
    uint8_t *out;       /* output array */
    uint32_t outlen;    /* digest length */

    uint8_t *pwd;       /* password array */
    uint32_t pwdlen;    /* password length */

    uint8_t *salt;      /* salt array */
    uint32_t saltlen;   /* salt length */

    uint8_t *secret;    /* key array */
    uint32_t secretlen; /* key length */

    uint8_t *ad;        /* associated data array */
    uint32_t adlen;     /* associated data length */

    uint32_t t_cost;    /* number of passes */
    uint32_t m_cost;    /* amount of memory requested (KB) */
    uint32_t lanes;     /* number of lanes */
    uint32_t threads;   /* maximum number of threads */
```

```
    uint32_t version;   /* version number */
} argon2_context;

/* Argon2 primitive type */
typedef enum Argon2_type {
  Argon2_d  = 0,
  Argon2_i  = 1,
  Argon2_id = 2
} argon2_type;

/* Version of the algorithm */
typedef enum Argon2_version {
    ARGON2_VERSION_10     = 0x10,
    ARGON2_VERSION_13     = 0x13,
    ARGON2_VERSION_NUMBER = ARGON2_VERSION_13
} argon2_version;

/*
 * Function that performs memory-hard hashing with certain degree of
 * parallelism
 * @param context Pointer to the Argon2 internal structure
 * @param type Argon2 primitive type
 * @return ARGON2_OK if successful, ARGON2_ERROR otherwise
 */
int argon2_ctx(argon2_context *context, argon2_type type);

/**
 * Hashes a password with Argon2, producing a raw hash at @hash
 * @param t_cost Number of iterations
 * @param m_cost Sets memory usage to m_cost kibibytes
 * @param parallelism Number of threads and compute lanes
 * @param pwd Pointer to password
 * @param pwdlen Password size in bytes
 * @param salt Pointer to salt
 * @param saltlen Salt size in bytes
 * @param hash Buffer where to write the raw hash - updated by the
 *        function
 * @param hashlen Desired length of the hash in bytes
 * @param type Argon2 primitive type
 * @param version Version of the algorithm
 * @pre   Different parallelism levels will give different results
 * @return ARGON2_OK if successful, ARGON2_ERROR otherwise
 */
int argon2_hash(const uint32_t t_cost, const uint32_t m_cost,
                const uint32_t parallelism, const void *pwd,
                const size_t pwdlen, const void *salt,
                const size_t saltlen, void *hash, const size_t hashlen,
                argon2_type type, const uint32_t version);
```

```
#endif
```

5.2. argon2.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "argon2.h"
#include "core.h"

int argon2_ctx(argon2_context *context, argon2_type type) {
    int result;
    uint32_t memory_blocks, segment_length;
    argon2_instance_t instance;

    /* Minimum memory_blocks = 8L blocks, where L is the number of
       lanes */
    memory_blocks = context->m_cost;
    if (memory_blocks < 2 * ARGON2_SYNC_POINTS * context->lanes) {
        memory_blocks = 2 * ARGON2_SYNC_POINTS * context->lanes;
    }

    /* Ensure that all segments have equal length */
    segment_length = memory_blocks / (context->lanes *
                                      ARGON2_SYNC_POINTS);
    memory_blocks = segment_length * (context->lanes *
                                      ARGON2_SYNC_POINTS);

    instance.version = context->version;
    instance.memory = NULL;
    instance.passes = context->t_cost;
    instance.memory_blocks = memory_blocks;
    instance.segment_length = segment_length;
    instance.lane_length = segment_length * ARGON2_SYNC_POINTS;
    instance.lanes = context->lanes;
    instance.threads = context->threads;
    instance.type = type;

    if (instance.threads > instance.lanes) {
        instance.threads = instance.lanes;
    }

    /* Initialization: hash inputs, allocate memory, fill first
       blocks */
    result = initialize(&instance, context);
    if (ARGON2_OK != result) {
        return result;
```

```
     }

     /* Fill memory */
     result = fill_memory_blocks(&instance);
     if (ARGON2_OK != result) {
         return result;
     }

     /* Finalization */
     finalize(context, &instance);

     return ARGON2_OK;
 }

 int argon2_hash(const uint32_t t_cost, const uint32_t m_cost,
                 const uint32_t parallelism, const void *pwd,
                 const size_t pwdlen, const void *salt,
                 const size_t saltlen, void *hash, const size_t hashlen,
                 argon2_type type, const uint32_t version) {
     argon2_context context;
     int result;
     uint8_t *out;

     out = malloc(hashlen);
     if (!out) {
         return ARGON2_ERROR;
     }

     context.out = (uint8_t *)out;
     context.outlen = (uint32_t)hashlen;
     context.pwd = CONST_CAST(uint8_t *)pwd;
     context.pwdlen = (uint32_t)pwdlen;
     context.salt = CONST_CAST(uint8_t *)salt;
     context.saltlen = (uint32_t)saltlen;
     context.secret = NULL;
     context.secretlen = 0;
     context.ad = NULL;
     context.adlen = 0;
     context.t_cost = t_cost;
     context.m_cost = m_cost;
     context.lanes = parallelism;
     context.threads = parallelism;
     context.version = version;

     result = argon2_ctx(&context, type);
     if (result != ARGON2_OK) {
         free(out);
         return result;
```

```
     }

     memcpy(hash, out, hashlen);
     free(out);

     return ARGON2_OK;
 }
```

5.3. core.h

```
#ifndef ARGON2_CORE_H
#define ARGON2_CORE_H

#include "argon2.h"

#define CONST_CAST(x) (x)(uintptr_t)

/* Argon2 internal constants */

enum argon2_core_constants {
    /* Memory block size in bytes */
    ARGON2_BLOCK_SIZE            = 1024,
    ARGON2_QWORDS_IN_BLOCK       = ARGON2_BLOCK_SIZE / 8,
    ARGON2_OWORDS_IN_BLOCK       = ARGON2_BLOCK_SIZE / 16,
    ARGON2_HWORDS_IN_BLOCK       = ARGON2_BLOCK_SIZE / 32,

    /* Number of pseudo-random values generated by one call to Blake in
       Argon2i to generate reference block positions */
    ARGON2_ADDRESSES_IN_BLOCK    = 128,

    /* Pre-hashing digest length and its extension*/
    ARGON2_PREHASH_DIGEST_LENGTH = 64,
    ARGON2_PREHASH_SEED_LENGTH   = 72
};

/* Argon2 internal data types */
/* Structure for the (1KB) memory block implemented as 128 64-bit
   words */
typedef struct block_ { uint64_t v[ARGON2_QWORDS_IN_BLOCK]; } block;

/* Functions that work with the block */
/* Initialize each byte of the block with @in */
void init_block_value(block *b, uint8_t in);

/* Copy block @src to block @dst */
void copy_block(block *dst, const block *src);

/* XOR @src onto @dst bytewise */
```

```
void xor_block(block *dst, const block *src);

/*
 * Argon2 instance: memory pointer, number of passes, amount of memory,
 * type, and derived values. Used to evaluate the number and location of
 * blocks to construct in each thread.
 */
typedef struct Argon2_instance_t {
    block *memory;              /* Memory pointer */
    uint32_t version;
    uint32_t passes;            /* Number of passes */
    uint32_t memory_blocks;     /* Number of blocks in memory */
    uint32_t segment_length;
    uint32_t lane_length;
    uint32_t lanes;
    uint32_t threads;
    argon2_type type;
    argon2_context *context_ptr; /* Points back to original context */
} argon2_instance_t;

/*
 * Argon2 position: where we construct the block right now. Used to
 * distribute work between threads.
 */
typedef struct Argon2_position_t {
    uint32_t pass;
    uint32_t lane;
    uint8_t slice;
    uint32_t index;
} argon2_position_t;

/* Struct that holds the inputs for thread handling FillSegment */
typedef struct Argon2_thread_data {
    argon2_instance_t *instance_ptr;
    argon2_position_t pos;
} argon2_thread_data;

/* Argon2 core functions */

/* Allocates memory to the given pointer. Total allocated memory is
 * @num * @size
 * @param memory pointer to the pointer to the memory
 * @param size the size in bytes for each element to be allocated
 * @param num the number of elements to be allocated
 * @return ARGON2_OK if @memory is a valid pointer and memory is
 * allocated
 */
int allocate_memory(uint8_t **memory, size_t num, size_t size);
```

```
/*
 * Frees memory at the given pointer.
 * @param memory pointer to buffer to be freed
 */
void free_memory(uint8_t *memory);

/*
 * Computes absolute position of reference block in the lane following a
 * skewed distribution and using a pseudo-random value as input
 * @param instance Pointer to the current instance
 * @param position Pointer to the current position
 * @param pseudo_rand 32-bit pseudo-random value used to determine the
 * position
 * @param same_lane Indicates if the block will be taken from the
 * current lane.
 * If so we can reference the current segment
 * @pre All pointers must be valid
 */
uint32_t index_alpha(const argon2_instance_t *instance,
                     const argon2_position_t *position,
                     uint32_t pseudo_rand, int same_lane);

/*
 * Hashes all the inputs into @blockhash[PREHASH_DIGEST_LENGTH]
 * @param blockhash Buffer for pre-hashing digest
 * @param context Pointer to the Argon2 internal structure containing
 * memory pointer, and parameters for time and space requirements
 * @param type Argon2 type
 * @pre @blockhash must have at least @PREHASH_DIGEST_LENGTH bytes
 * allocated
 */
void initial_hash(uint8_t *blockhash, argon2_context *context,
                  argon2_type type);

/*
 * Creates first 2 blocks per lane
 * @param blockhash Pointer to the pre-hashing digest
 * @param instance Pointer to the current instance
 * @pre @blockhash must point to @PREHASH_SEED_LENGTH allocated values
 */
void fill_first_blocks(uint8_t *blockhash,
                       const argon2_instance_t *instance);

/*
 * Function allocates memory, hashes the inputs with Blake, and creates
 * first two blocks. Returns the pointer to the main memory with
 * 2 blocks per lane initialized
 * @param instance Current Argon2 instance
```

```
 * @param context Pointer to the Argon2 internal structure containing
 * memory pointer, and parameters for time and space requirements
 * @return ARGON2_OK if successful, ARGON2_ERROR if memory failed to
 * allocate.
 * @context->state will be modified if successful
 */
int initialize(argon2_instance_t *instance, argon2_context *context);

/*
 * XORing the last block of each lane, hashing it, making the tag.
 * Deallocates the memory.
 * @param context Pointer to current Argon2 context (use only the out
 * parameters from it)
 * @param instance Pointer to current instance of Argon2
 * @pre instance->state must point to necessary amount of memory
 * @pre context->out must point to outlen bytes of memory
 */
void finalize(const argon2_context *context,
              argon2_instance_t *instance);

/*
 * Function that fills the segment using previous segments also from
 * other threads
 * @param instance Pointer to the current instance
 * @param position Current position
 * @pre all block pointers must be valid
 */
void fill_segment(const argon2_instance_t *instance,
                  argon2_position_t position);

/*
 * Function that fills the entire memory t_cost times based on the first
 * 2 blocks in each lane
 * @param instance Pointer to the current instance
 * @return ARGON2_OK if successful, ARGON2_ERROR otherwise
 */
int fill_memory_blocks(argon2_instance_t *instance);

#endif
```

5.4. core.c

```
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "core.h"
```

```
#include "blake2/blake2.h"
#include "blake2/blake2-impl.h"

/* Instance and position constructors */
void init_block_value(block *b, uint8_t in) {
    memset(b->v, in, sizeof(b->v));
}

void copy_block(block *dst, const block *src) {
    memcpy(dst->v, src->v, sizeof(uint64_t) * ARGON2_QWORDS_IN_BLOCK);
}

void xor_block(block *dst, const block *src) {
    int i;
    for (i = 0; i < ARGON2_QWORDS_IN_BLOCK; ++i) {
        dst->v[i] ^= src->v[i];
    }
}

static void load_block(block *dst, const void *input) {
    unsigned i;
    for (i = 0; i < ARGON2_QWORDS_IN_BLOCK; ++i) {
        dst->v[i] = load64((const uint8_t *)input +
                            i * sizeof(dst->v[i]));
    }
}

static void store_block(void *output, const block *src) {
    unsigned i;
    for (i = 0; i < ARGON2_QWORDS_IN_BLOCK; ++i) {
        store64((uint8_t *)output + i * sizeof(src->v[i]), src->v[i]);
    }
}

/* Memory functions */
int allocate_memory(uint8_t **memory, size_t num, size_t size) {
    size_t memory_size = num * size;
    if (memory == NULL) {
        return ARGON2_ERROR;
    }

    /* Check for multiplication overflow */
    if (size != 0 && memory_size / size != num) {
        return ARGON2_ERROR;
    }

    /* Try to allocate with appropriate allocator */
    *memory = malloc(memory_size);
```

```
    if (*memory == NULL) {
        return ARGON2_ERROR;
    }

    return ARGON2_OK;
}

void free_memory(uint8_t *memory) {
    free(memory);
}

void finalize(const argon2_context *context,
              argon2_instance_t *instance) {
    if (context != NULL && instance != NULL) {
        block blockhash;
        uint32_t l;

        copy_block(&blockhash,
                   instance->memory + instance->lane_length - 1);

        /* XOR the last blocks */
        for (l = 1; l < instance->lanes; ++l) {
            uint32_t last_block_in_lane =
                l * instance->lane_length + (instance->lane_length - 1);
            xor_block(&blockhash,
                      instance->memory + last_block_in_lane);
        }

        /* Hash the result */
        {
            uint8_t blockhash_bytes[ARGON2_BLOCK_SIZE];
            store_block(blockhash_bytes, &blockhash);
            blake2b_long(context->out, context->outlen, blockhash_bytes,
                         ARGON2_BLOCK_SIZE);
            /* clear blockhash and blockhash_bytes */
            memset(blockhash.v, 0, ARGON2_BLOCK_SIZE);
            memset(blockhash_bytes, 0, ARGON2_BLOCK_SIZE);
        }

        free_memory((uint8_t *)instance->memory);
    }
}

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;
}

/* Single-threaded version for p=1 case */
static int fill_memory_blocks_st(argon2_instance_t *instance) {
    uint32_t r, s, l;

    for (r = 0; r < instance->passes; ++r) {
        for (s = 0; s < ARGON2_SYNC_POINTS; ++s) {
            for (l = 0; l < instance->lanes; ++l) {
                argon2_position_t position = {r, l, (uint8_t)s, 0};
                fill_segment(instance, position);
            }
        }
    }
    return ARGON2_OK;
}

int fill_memory_blocks(argon2_instance_t *instance) {
    if (instance == NULL || instance->lanes == 0) {
        return ARGON2_ERROR;
    }

    return fill_memory_blocks_st(instance);
}

void fill_first_blocks(uint8_t *blockhash,
                       const argon2_instance_t *instance) {
    uint32_t l;
    /* Make the first and second block in each lane as G(H0||0||i) or
       G(H0||1||i) */
    uint8_t blockhash_bytes[ARGON2_BLOCK_SIZE];
    for (l = 0; l < instance->lanes; ++l) {
```

```
        store32(blockhash + ARGON2_PREHASH_DIGEST_LENGTH, 0);
        store32(blockhash + ARGON2_PREHASH_DIGEST_LENGTH + 4, l);
        blake2b_long(blockhash_bytes, ARGON2_BLOCK_SIZE, blockhash,
                     ARGON2_PREHASH_SEED_LENGTH);
        load_block(&instance->memory[l * instance->lane_length + 0],
                   blockhash_bytes);

        store32(blockhash + ARGON2_PREHASH_DIGEST_LENGTH, 1);
        blake2b_long(blockhash_bytes, ARGON2_BLOCK_SIZE, blockhash,
                     ARGON2_PREHASH_SEED_LENGTH);
        load_block(&instance->memory[l * instance->lane_length + 1],
                   blockhash_bytes);
    }
    memset(blockhash_bytes, 0, ARGON2_BLOCK_SIZE);
}

void initial_hash(uint8_t *blockhash, argon2_context *context,
                  argon2_type type) {
    blake2b_state BlakeHash;
    uint8_t value[sizeof(uint32_t)];

    if (NULL == context || NULL == blockhash) {
        return;
    }

    blake2b_init(&BlakeHash, ARGON2_PREHASH_DIGEST_LENGTH);

    store32(&value, context->lanes);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, context->outlen);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, context->m_cost);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, context->t_cost);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, context->version);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, (uint32_t)type);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    store32(&value, context->pwdlen);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));
```

```
    if (context->pwd != NULL) {
        blake2b_update(&BlakeHash, (const uint8_t *)context->pwd,
                       context->pwdlen);
    }

    store32(&value, context->saltlen);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    if (context->salt != NULL) {
        blake2b_update(&BlakeHash, (const uint8_t *)context->salt,
                       context->saltlen);
    }

    store32(&value, context->secretlen);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    if (context->secret != NULL) {
        blake2b_update(&BlakeHash, (const uint8_t *)context->secret,
                       context->secretlen);
    }

    store32(&value, context->adlen);
    blake2b_update(&BlakeHash, (const uint8_t *)&value, sizeof(value));

    if (context->ad != NULL) {
        blake2b_update(&BlakeHash, (const uint8_t *)context->ad,
                       context->adlen);
    }

    blake2b_final(&BlakeHash, blockhash, ARGON2_PREHASH_DIGEST_LENGTH);
}

int initialize(argon2_instance_t *instance, argon2_context *context) {
    uint8_t blockhash[ARGON2_PREHASH_SEED_LENGTH];
    int result = ARGON2_OK;

    if (instance == NULL || context == NULL)
        return ARGON2_ERROR;
    instance->context_ptr = context;

    /* 1. Memory allocation */
    result = allocate_memory((uint8_t **)&(instance->memory),
                             instance->memory_blocks, sizeof(block));
    if (result != ARGON2_OK) {
        return result;
    }

    /* 2. Initial hashing */
```

```
    /* H_0 + 8 extra bytes to produce the first blocks */
    /* Hashing all inputs */
    initial_hash(blockhash, context, instance->type);
    /* Zeroing 8 extra bytes */
    memset(blockhash + ARGON2_PREHASH_DIGEST_LENGTH, 0,
                    ARGON2_PREHASH_SEED_LENGTH -
                    ARGON2_PREHASH_DIGEST_LENGTH);

    /* 3. Creating first blocks, always at least two blocks in a
     * slice */
    fill_first_blocks(blockhash, instance);
    /* Clearing the hash */
    memset(blockhash, 0, ARGON2_PREHASH_SEED_LENGTH);

    return ARGON2_OK;
}
```

5.5. ref.c

```
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include "argon2.h"
#include "core.h"

#include "blake2/blamka-round-ref.h"
#include "blake2/blake2-impl.h"
#include "blake2/blake2.h"


/*
 * Fills a new memory block and optionally XORs the old block over the
 * new one
 * @param prev_block Pointer to the previous block
 * @param ref_block Pointer to the reference block
 * @param next_block Pointer to the block to be constructed
 * @param with_xor Whether to XOR into the new block (1) or just
 * overwrite (0)
 * @pre all block pointers must be valid
 * @pre @next_block must be initialized
 */
static void fill_block(const block *prev_block, const block *ref_block,
                       block *next_block, int with_xor) {
    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 block_tmp = ref_block +
     * prev_block */
    if (with_xor) {
        /* Saving the next block contents for XOR over: */
        xor_block(&block_tmp, next_block);
        /* Now blockR = ref_block + prev_block and
           block_tmp = ref_block + prev_block + next_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) ..., 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);
}

static void next_addresses(block *address_block, block *input_block,
                           const block *zero_block) {
    input_block->v[6]++;
    fill_block(zero_block, input_block, address_block, 0);
```

```
    fill_block(zero_block, address_block, address_block, 0);
}

void fill_segment(const argon2_instance_t *instance,
                  argon2_position_t position) {
    block *ref_block = NULL, *curr_block = NULL;
    block address_block, input_block, zero_block;
    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;

    if (instance == NULL) {
        return;
    }

    data_independent_addressing =
        (instance->type == Argon2_i) ||
        (instance->type == Argon2_id && (position.pass == 0) &&
         (position.slice < ARGON2_SYNC_POINTS / 2));

    if (data_independent_addressing) {
        init_block_value(&zero_block, 0);
        init_block_value(&input_block, 0);

        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;
    }

    starting_index = 0;

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

        /* Don't forget to generate the first block of addresses: */
        if (data_independent_addressing) {
            next_addresses(&address_block, &input_block, &zero_block);
        }
    }

    /* 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) {
            if (i % ARGON2_ADDRESSES_IN_BLOCK == 0) {
                next_addresses(&address_block, &input_block,
                               &zero_block);
            }
            pseudo_rand = address_block.v[i %
                                          ARGON2_ADDRESSES_IN_BLOCK];
        } 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 (ARGON2_VERSION_10 == instance->version) {
            /* version 1.2.1 and earlier: overwrite, not XOR */
            fill_block(instance->memory + prev_offset, ref_block,
                       curr_block, 0);
        } else {
            if(0 == position.pass) {
                fill_block(instance->memory + prev_offset, ref_block,
                           curr_block, 0);
            } else {
                fill_block(instance->memory + prev_offset, ref_block,
                           curr_block, 1);
            }
        }
    }
}
```

5.6. main.c

```
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "argon2.h"
#include "core.h"

#define T_COST 3
#define M_COST 1 << 12
#define THREADS 1
#define OUTLEN 32

char pwd[] = "pasword";
char salt[] = "somesalt";

static void print_hex(uint8_t *bytes, size_t bytes_len) {
    size_t i;
    for (i = 0; i < bytes_len; ++i) {
        printf("%02x", bytes[i]);
    }
    printf("\n");
}

static int run(uint32_t outlen, char *pwd, char *salt, uint32_t t_cost,
               uint32_t m_cost, uint32_t threads, argon2_type type,
               uint32_t version, unsigned char *expected_hash) {
```

```
    size_t pwdlen, saltlen;
    uint32_t i;
    int result;
    unsigned char *out = NULL;

    pwdlen = strlen(pwd);
    saltlen = strlen(salt);

    out = malloc(outlen + 1);
    if (!out) {
        memset(pwd, 0, strlen(pwd));
        printf("Could not allocate memory for output!");
    }

    result = argon2_hash(t_cost, m_cost, threads, pwd, pwdlen, salt,
                         saltlen, out, outlen, type, version);
    if (ARGON2_OK != result) {
        printf("Error while hasing!");
    }

    printf("  Actual Hash:\t\t");
    print_hex(out, outlen);
    printf("Expected Hash:\t\t");
    print_hex(expected_hash, outlen);

    if (ARGON2_OK == result) {
        for (i = 0; i < outlen; i++) {
            if (expected_hash[i] != out[i]) {
                return ARGON2_ERROR;
            }
        }
    }

    free(out);
    return result;
}

int argon2_i_selftest()
{
    unsigned char expected_hash[] = {
        0x95, 0x7f, 0xc0, 0x72, 0x7d, 0x83, 0xf4, 0x06,
        0x0b, 0xb0, 0xf1, 0x07, 0x1e, 0xb5, 0x90, 0xa1,
        0x9a, 0x8c, 0x44, 0x8f, 0xc0, 0x20, 0x94, 0x97,
        0xee, 0x4f, 0x54, 0xca, 0x24, 0x1f, 0x3c, 0x90};
    return run(OUTLEN, pwd, salt, T_COST, M_COST, THREADS, Argon2_i,
               ARGON2_VERSION_NUMBER, expected_hash);
}
```

```
int argon2_d_selftest()
{
    unsigned char expected_hash[] = {
        0x0b, 0x3f, 0x09, 0xe7, 0xb8, 0xd0, 0x36, 0xe5,
        0x8c, 0xcd, 0x08, 0xf0, 0x8c, 0xb6, 0xba, 0xbf,
        0x7e, 0x5e, 0x24, 0x63, 0xc2, 0x6b, 0xcf, 0x2a,
        0x9e, 0x4e, 0xa7, 0x0d, 0x74, 0x7c, 0x40, 0x98};
    return run(OUTLEN, pwd, salt, T_COST, M_COST, THREADS, Argon2_d,
               ARGON2_VERSION_NUMBER, expected_hash);
}

int argon2_id_selftest()
{
    unsigned char expected_hash[] = {
        0xf5, 0x55, 0x35, 0xbf, 0xe9, 0x48, 0x71, 0x00,
        0x51, 0x42, 0x4c, 0x74, 0x24, 0xb1, 0x1b, 0xa9,
        0xa1, 0x3a, 0x50, 0x23, 0x9b, 0x04, 0x59, 0xf5,
        0x6c, 0xa6, 0x95, 0xea, 0x14, 0xbc, 0x19, 0x5e};
    return run(OUTLEN, pwd, salt, T_COST, M_COST, THREADS, Argon2_id,
               ARGON2_VERSION_NUMBER, expected_hash);
}

int main(int argc, char *argv[]) {
    printf("Argon2i - %s\n", ARGON2_OK == argon2_i_selftest() ?
           "OK" : "FAIL");
    printf("Argon2d - %s\n", ARGON2_OK == argon2_d_selftest() ?
           "OK" : "FAIL");
    printf("Argon2id - %s\n", ARGON2_OK == argon2_id_selftest() ?
           "OK" : "FAIL");

    return ARGON2_OK;
}
```

5.7. blake2.h

```
int blake2b_long(void *out, size_t outlen, const void *in,
                 size_t inlen);
```

5.8. blake2.c

```
int blake2b_long(void *pout, size_t outlen, const void *in,
                 size_t inlen) {
    uint8_t *out = (uint8_t *)pout;
    blake2b_state blake_state;
    uint8_t outlen_bytes[sizeof(uint32_t)] = {0};
    int ret = -1;
```

```
     if (outlen > UINT32_MAX) {
         goto fail;
     }

     /* Ensure little-endian byte order! */
     store32(outlen_bytes, (uint32_t)outlen);

 #define TRY(statement)                      \
     do {                                    \
         ret = statement;                    \
         if (ret < 0) {                      \
             goto fail;                      \
         }                                   \
     } while ((void)0, 0)

     if (outlen <= BLAKE2B_OUTBYTES) {
         TRY(blake2b_init(&blake_state, outlen));
         TRY(blake2b_update(&blake_state, outlen_bytes,
             sizeof(outlen_bytes)));
         TRY(blake2b_update(&blake_state, in, inlen));
         TRY(blake2b_final(&blake_state, out, outlen));
     } else {
         uint32_t toproduce;
         uint8_t out_buffer[BLAKE2B_OUTBYTES];
         uint8_t in_buffer[BLAKE2B_OUTBYTES];
         TRY(blake2b_init(&blake_state, BLAKE2B_OUTBYTES));
         TRY(blake2b_update(&blake_state, outlen_bytes,
             sizeof(outlen_bytes)));
         TRY(blake2b_update(&blake_state, in, inlen));
         TRY(blake2b_final(&blake_state, out_buffer, BLAKE2B_OUTBYTES));
         memcpy(out, out_buffer, BLAKE2B_OUTBYTES / 2);
         out += BLAKE2B_OUTBYTES / 2;
         toproduce = (uint32_t)outlen - BLAKE2B_OUTBYTES / 2;

         while (toproduce > BLAKE2B_OUTBYTES) {
             memcpy(in_buffer, out_buffer, BLAKE2B_OUTBYTES);
             TRY(blake2b(out_buffer, BLAKE2B_OUTBYTES, in_buffer,
                         BLAKE2B_OUTBYTES, NULL, 0));
             memcpy(out, out_buffer, BLAKE2B_OUTBYTES / 2);
             out += BLAKE2B_OUTBYTES / 2;
             toproduce -= BLAKE2B_OUTBYTES / 2;
         }

         memcpy(in_buffer, out_buffer, BLAKE2B_OUTBYTES);
         TRY(blake2b(out_buffer, toproduce, in_buffer, BLAKE2B_OUTBYTES,
                     NULL, 0));
         memcpy(out, out_buffer, toproduce);
     }
```

```
 fail:
     memset(&blake_state, 0, sizeof(blake_state));
     return ret;
 #undef TRY
```

5.9. blake2-impl.h

```
#if (defined(__BYTE_ORDER__) &&                          \
     (__BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__)) ||     \
    defined(__LITTLE_ENDIAN__) || defined(__ARMEL__) ||  \
    defined(__MIPSEL__) || defined(__AARCH64EL__) ||     \
    defined(__amd64__) || defined(__i386__) ||           \
    defined(_M_IX86) || defined(_M_X64) ||               \
    defined(_M_AMD64) || defined(_M_ARM)
#define NATIVE_LITTLE_ENDIAN
#endif
```

5.10. blamka-round-ref.h

```
#ifndef BLAKE_ROUND_MKA_H
#define BLAKE_ROUND_MKA_H

#include "blake2.h"
#include "blake2-impl.h"

/*designed by the Lyra PHC team */
static BLAKE2_INLINE uint64_t fBlaMka(uint64_t x, uint64_t y) {
    const uint64_t m = UINT64_C(0xFFFFFFFF);
    const uint64_t xy = (x & m) * (y & m);
    return x + y + 2 * xy;
}

#define G(a, b, c, d)                 \
    do {                              \
        a = fBlaMka(a, b);            \
        d = rotr64(d ^ a, 32);        \
        c = fBlaMka(c, d);            \
        b = rotr64(b ^ c, 24);        \
        a = fBlaMka(a, b);            \
        d = rotr64(d ^ a, 16);        \
        c = fBlaMka(c, d);            \
        b = rotr64(b ^ c, 63);        \
    } while ((void)0, 0)

#define BLAKE2_ROUND_NOMSG(v0, v1, v2, v3, v4, v5, v6, v7,
                           v8, v9, v10, v11, v12, v13, v14, v15) \
    do {                                                          \
        G(v0, v4, v8, v12);                                       \
        G(v1, v5, v9, v13);                                       \
        G(v2, v6, v10, v14);                                      \
        G(v3, v7, v11, v15);                                      \
        G(v0, v5, v10, v15);                                      \
        G(v1, v6, v11, v12);                                      \
        G(v2, v7, v8, v13);                                       \
        G(v3, v4, v9, v14);                                       \
    } while ((void)0, 0)

#endif
```

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 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 are 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 gibibyte 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], the number of passes must exceed binary logarithm of memory minus 26. Asymptotically, the best attack on 1-pass Argon2i is given in [BZ17] with reduction factor O(m^(0.233)) where m is the number of blocks. This attack is also asymptotically optimal as [BZ17] also prove the upper bound of O(m^(0.25)).

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 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 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 Argon2id 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

[RFC7693] 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, <http://www.rfc-editor.org/info/rfc7693>.

10.2. Informative References

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

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

[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.

[ARGON2ESP]
Biryukov, A., Dinu, D., and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications", Euro S&P 2016 <https://www.cryptolux.org/images/0/0d/Argon2ESP.pdf>, March 2016.

[ARGON2GITHUB]
"The password hash Argon2, winner of PHC", WWW <https://github.com/P-H-C/phc-winner-argon2>, July 2017.

[BZ17] Blocki, J. and S. Zhou, "On the Depth-Robustness and Cumulative Pebbling Cost of Argon2i", WWW <https://eprint.iacr.org/2017/442.pdf>, May 2017.

[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.

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/