Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: October 27, 2015

M-J. Saarinen, Ed.
Independent Consultant
J-P. Aumasson
Kudelski Security

April 25, 2015

The BLAKE2 Cryptographic Hash and MAC
draft-saarinen-blake2-03

Abstract

This document describes the cryptographic hash function BLAKE2, making the algorithm specification and C source code conveniently available to the Internet community. BLAKE2 comes in two main flavors: BLAKE2b is optimized for 64-bit platforms, and BLAKE2s for smaller architectures. BLAKE2 can be directly keyed, making it functionally equivalent to a Message Authentication Code (MAC).

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 October 27, 2015.

Copyright Notice

Table of Contents

1. Introduction and Terminology

The [BLAKE2] cryptographic hash function was designed by Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein.

BLAKE2 comes in two basic flavors:

- BLAKE2b (or just BLAKE2) is optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes.

- BLAKE2s is optimized for 8- to 32-bit platforms, and produces digests of any size between 1 and 32 bytes.

Both BLAKE2b and BLAKE2s are believed to be highly secure and have good performance on any platform, software or hardware. BLAKE2 does not require a special "HMAC" construction for keyed message authentication as they have a built-in keying mechanism.

The BLAKE2 hash function may be used by digital signature algorithms and message authentication and integrity protection mechanisms in applications such as Public Key Infrastructure (PKI), secure communication protocols, cloud storage, intrusion detection, forensic suites, and version control systems.

The BLAKE2 suite provides a more efficient alternative to US Secure Hash Algorithms SHA and HMAC-SHA [RFC6234]. BLAKE2s-128 is especially suited as a fast and more secure drop-in replacement to MD5 and HMAC-MD5 in legacy applications [RFC6151].

A reference implementation in C programming language for BLAKE2b can be found in Appendix A and for BLAKE2s in Appendix B of this document. These implementations SHOULD be validated with the Test Module contained in Appendix C.

Due to space constraints, this document does not contain a full set of test vectors for BLAKE2. We refer to [BLAKE2] for up to date information about compliance testing and integrating BLAKE2 into various applications.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Conventions, Variables, and Constants

2.1. Parameters

The following table summarizes various parameters and their ranges:

	BLAKE2b	BLAKE2s
Bits in word	w = 64	w = 32
Rounds in F	r = 12	r = 10
Block bytes	bb = 128	bb = 64
Hash bytes	1 <= nn <= 64	1 <= nn <= 32
Key bytes	0 <= kk <= 64	0 <= kk <= 32
Input bytes	0 <= ll < 2**128	0 <= ll < 2**64
G Rotation constants =	(R1, R2, R3, R4) (32, 24, 16, 63)	(R1, R2, R3, R4) (16, 12, 8, 7)

2.2. Other Constants and Variables

IV[0..7]	Initialization Vector (constant).
SIGMA[0..9]	Message word permutations (constant).
p[0..7]	Parameter block (defines hash and key sizes).
m[0..15]	Sixteen words of a single message block.
h[0..7]	Internal state of the hash.
d[0..dd-1]	Padded input blocks. "dd" is the number of blocks.
t	Byte offset at the end of the current block.
f	Flag indicating the last block.

2.3. Arithmetic Notation

For real-valued x we define:

floor(x)	Floor, the largest integer <= x.
ceil(x)	Ceiling, the smallest integer >= x.
frac(x)	Positive fractional part of x, frac(x) = x - floor(x).

Operator notation in pseudocode:

2**n =	2 to the power "n". 2**0=1, 2**1=2, 2**2=4, 2**3=8, etc.
a ^ b =	Bitwise exclusive-or operation between "a" and "b".
a mod b =	Remainder "a" modulo "b", always in range [0, b-1].
x >> n =	floor(x / 2**n). Logical shift "x" right by "n" bits.
x << n =	(x * 2**n) mod (2**w). Logical shift "x" left by "n".
x >>> n =	(x >> n) ^ (x << (w - n)). Rotate "x" right by "n" bits.

2.4. Little-Endian Interpretation of Words as Byte Sequences

All mathematical operations are on 64-bit words in BLAKE2b and on 32-bit words on BLAKE2s.

We may also perform operations on vectors of words. Vector indexing is zero-based; the first element of an n-element vector "v" is v[0] and the last one is v[n - 1]. All elements is denoted by v[0..n-1].

Byte (octet) streams are interpreted as words in little-endian order, with the least significant byte first. Consider this sequence of eight hexadecimal bytes:

```
x[0..7] = 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF
```

When interpreted as a 32-bit word from the beginning memory address, x[0..3] has numerical value 0x67452301 or 1732584193.

When interpreted as a 64-bit word, bytes x[0..7] have numerical value 0xEFCDAB8967452301 or 17279655951921914625.

2.5. Parameter Block

We specify the parameter block words p[0..7] as follows:

```
        byte offset:    3 2 1 0     (otherwise zero)
             p[0] = 0x0101kknn      p[1..7] = 0
```

Here the "nn" byte specifies the hash size in bytes. The second (little-endian) byte of parameter block, "kk", specifies key size in bytes. Set kk = 00 for for unkeyed hashing. Bytes 2 and 3 are set as 01. All other bytes in the parameter block are set as zero.

NOTE. [BLAKE2] defines additional variants of BLAKE2 with features such as salting, personalized hashes, and tree hashing. These OPTIONAL features use fields in the parameter block which are not defined in this document.

2.6. Initialization Vector

We define the Initialization Vector constant IV mathematically as:

```
IV[i] = floor(2**w * frac(sqrt(prime(i+1)))), where prime(i)
    is the i:th prime number ( 2, 3, 5, 7, 11, 13, 17, 19 )
    and sqrt(x) is the square root of x.
```

The numerical values of IV can be also found in implementations in Appendix A and Appendix B for BLAkE2b and BLAKE2s, respectively.

NOTE. BLAKE2b IV is the same as SHA-512 IV and BLAKE2s IV is the same as SHA-256 IV; see [RFC6234].

2.7. Message Schedule SIGMA

Message word schedule permutations for each round of both BLAKE2b and BLAKE2s are defined by SIGMA. For BLAKE2b the two extra permutations for rounds 10 and 11 are SIGMA[10..11] = SIGMA[0..1].

Round	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIGMA[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIGMA[1]	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
SIGMA[2]	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
SIGMA[3]	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
SIGMA[4]	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
SIGMA[5]	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
SIGMA[6]	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
SIGMA[7]	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
SIGMA[8]	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
SIGMA[9]	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

3. BLAKE2 Processing

3.1. Mixing Function G

The G primitive function mixes two input words "x" and "y" into four words indexed by "a", "b", "c", and "d" in the working vector v[0..15]. The full modified vector is returned. The rotation constants (R1, R2, R3, R4) are given in Section 2.1.

```
FUNCTION G( v[0..15], a, b, c, d, x, y )
|
|   v[a] := (v[a] + v[b] + x) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R1
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R2
|   v[a] := (v[a] + v[b] + y) mod 2**w
|   v[d] := (v[d] ^ v[a]) >>> R3
|   v[c] := (v[c] + v[d])     mod 2**w
|   v[b] := (v[b] ^ v[c]) >>> R4
|
|   RETURN v[0..15]
|
END FUNCTION.
```

3.2. Compression Function F

Compression function F takes as an argument the state vector "h", zero-padded message block vector "m", 2w-bit offset counter "t", and

final block indicator flag "f". Local vector v[0..15] is used in processing. F returns a new state vector.

The number of rounds "r" is 12 for BLAKE2b and 10 for BLAKE2s. Rounds are numbered from 0 to r - 1.

```
    FUNCTION F( h[0..7], m[0..15], t, f )
    |
    |   // Initialize local work vector v[0..15]
    |   v[0..7] := h[0..7]              // First half from state.
    |   v[8..15] := IV[0..7]           // Second half from IV.
    |
    |   v[12] := v[12] ^ (t mod 2**w)   // Low word of the offset.
    |   v[13] := v[13] ^ (t >> w)       // High word.
    |
    |   IF f = TRUE THEN                // last block flag?
    |   |   v[14] := v[14] ^ 0xFF..FF   // Invert all bits.
    |   END IF.
    |
    |   // Cryptographic mixing
    |   FOR i = 0 TO r - 1 DO           // Ten or twelve rounds.
    |   |
    |   |   // Message word selection permutation for this round.
    |   |   s[0..15] := SIGMA[i mod 10][0..15]
    |   |
    |   |   v := G( v, 0, 4,  8, 12, m[s[ 0]], m[s[ 1]] )
    |   |   v := G( v, 1, 5,  9, 13, m[s[ 2]], m[s[ 3]] )
    |   |   v := G( v, 2, 6, 10, 14, m[s[ 4]], m[s[ 5]] )
    |   |   v := G( v, 3, 7, 11, 15, m[s[ 6]], m[s[ 7]] )
    |   |
    |   |   v := G( v, 0, 5, 10, 15, m[s[ 8]], m[s[ 9]] )
    |   |   v := G( v, 1, 6, 11, 12, m[s[10]], m[s[11]] )
    |   |   v := G( v, 2, 7,  8, 13, m[s[12]], m[s[13]] )
    |   |   v := G( v, 3, 4,  9, 14, m[s[14]], m[s[15]] )
    |   |
    |   END FOR
    |
    |   FOR i = 0 TO 7 DO               // XOR the two halves.
    |   |   h[i] := h[i] ^ v[i] ^ v[i + 8]
    |   END FOR.
    |
    |   RETURN h[0..7]                  // New state.
    |
    END FUNCTION.
```

3.3. Padding data and Computing a BLAKE2 Digest

We refer the reader to Appendix A and Appendix B for reference C language implementations of BLAkE2b and BLAKE2s, respectively.

Key and data input is split and padded into "dd" message blocks d[0..dd-1], each consisting of 16 words (or "bb" bytes).

If a secret key is used (kk > 0), it is padded with zero bytes and set as d[0]. Otherwise d[0] is the first data block. The final data block d[dd-1] is also padded with zero to "bb" bytes (16 words).

Number of blocks is therefore dd = ceil(kk / bb) + ceil(ll / bb). However in special case of unkeyed empty message (kk = 0 and ll = 0), we still set dd = 1 and d[0] consists of all zeros.

The following procedure for processes the padded data blocks into an "nn"-byte final hash value. See Section 2 for description of various variables and constants used.

```
FUNCTION BLAKE2( d[0..dd-1], ll, kk, nn )
|
|   h[0..7] := IV[0..7]             // Initialization Vector.
|
|   // Parameter block p[0]
|   h[0] := h[0] ^ 0x01010000 ^ (kk << 8) ^ nn
|
|   // Process padded key and data blocks
|   IF dd > 1 THEN
|   |   FOR i = 0 TO dd - 2 DO
|   |   |   h := COMPRESS( h, d[i], (i + 1) * bb, FALSE )
|   |   END FOR.
|   END IF.
|
|   // Final block.
|   IF kk = 0 THEN
|   |   h := COMPRESS( h, d[dd - 1], ll, TRUE )
|   ELSE
|   |   h := COMPRESS( h, d[dd - 1], ll + bb, TRUE )
|   END IF.
|
|   RETURN first "nn" bytes from little-endian word array h[].
|
END FUNCTION.
```

4. Standard Parameter Sets and Algorithm Identifiers

An implementation of BLAKE2b and / or BLAKE2s SHOULD support the following digest size parameters for interoperability (e.g. digital signatures), as long as sufficient level of security is attained by the parameter selections. These parameters and identifiers are intended to be suitable as drop-in replacements to corresponding SHA algorithms.

For unkeyed hashing, developers adapting BLAKE2 to ASN.1 - based message formats SHOULD use the OID tree at x = 1.3.6.1.4.1.1722.12.2.

Algorithm Identifier	Target Arch	Collision Security	Hash nn	Hash ASN.1 OID Suffix
id-blake2b160	64-bit	2**80	20	x.1.20
id-blake2b256	64-bit	2**128	32	x.1.32
id-blake2b384	64-bit	2**192	48	x.1.48
id-blake2b512	64-bit	2**256	64	x.1.64
id-blake2s128	32-bit	2**64	16	x.2.16
id-blake2s160	32-bit	2**80	20	x.2.20
id-blake2s224	32-bit	2**112	28	x.2.28
id-blake2s256	32-bit	2**128	32	x.2.32

```
hashAlgs OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1)
    private(4) enterprise(1) kudelski(1722) cryptography(12) 2
}

-- the two BLAKE2 variants --
blake2b OBJECT IDENTIFIER ::= { hashAlgs 1 }
blake2s OBJECT IDENTIFIER ::= { hashAlgs 2 }

-- BLAKE2b Identifiers --
id-blake2b160 OBJECT IDENTIFIER ::= { blake2b 20 }
id-blake2b256 OBJECT IDENTIFIER ::= { blake2b 32 }
id-blake2b384 OBJECT IDENTIFIER ::= { blake2b 48 }
id-blake2b512 OBJECT IDENTIFIER ::= { blake2b 64 }

-- BLAKE2s Identifiers --
id-blake2s128 OBJECT IDENTIFIER ::= { blake2s 16 }
id-blake2s160 OBJECT IDENTIFIER ::= { blake2s 20 }
id-blake2s224 OBJECT IDENTIFIER ::= { blake2s 28 }
id-blake2s256 OBJECT IDENTIFIER ::= { blake2s 32 }
```

5. Acknowledgements

The editor wishes to thank the [BLAKE2] team for their encouragement: Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. We have borrowed passages from [BLAKE] and [BLAKE2] with permission.

BLAKE2 is based on the SHA-3 proposal [BLAKE], designed by Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. BLAKE2, like BLAKE, relies on a core algorithm borrowed from the ChaCha stream cipher, designed by Daniel J. Bernstein.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

This document is intended to provide convenient open source access by the Internet community to the BLAKE2 cryptographic hash algorithm. We wish to make no independent assertion to its security in this document. We refer the reader to [BLAKE] and [BLAKE2] for detailed cryptanalytic rationale behind its design.

In order to avoid bloat, the reference implementations in Appendix A and Appendix B may not erase all sensitive data (such as secret keys) immediately from process memory after use. Such cleanups can be added if needed.

8. References

8.1. Normative References

[BLAKE2] Aumasson, J-P., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5", January 2013, <https://blake2.net/>.

8.2. Informative References

[BLAKE] Aumasson, J-P., Meier, W., Phan, R., and L. Henzen, "The Hash Function BLAKE", February 2015, <https://131002.net/blake/>.

[FIPS140-2IG] NIST, US., "Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program", January 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, BCP 14, March 1997.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, March 2011.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

Appendix A. BLAKE2b Implementation C Source

A.1. blake2b.h

<CODE BEGINS>

```
// blake2b.h
// BLAKE2b Hashing Context and API Prototypes

#ifndef BLAKE2B_H
#define BLAKE2B_H

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

// state context
typedef struct {
    uint8_t b[128];                     // input buffer
    uint64_t h[8];                      // chained state
    uint64_t t[2];                      // total number of bytes
    size_t c;                           // pointer for b[]
    size_t outlen;                      // digest size
} blake2b_ctx;

// Initialize the hashing context "ctx" with optional key "key".
//      1 <= outlen <= 64 gives the digest size in bytes.
//      Secret key (also <= 64 bytes) is optional (keylen = 0).
int blake2b_init(blake2b_ctx *ctx, size_t outlen,
    const void *key, size_t keylen);    // secret key

// Add "inlen" bytes from "in" into the hash.
void blake2b_update(blake2b_ctx *ctx,   // context
    const void *in, size_t inlen);      // data to be hashed

// Generate the message digest (size given in init).
//      Result placed in "out"
void blake2b_final(blake2b_ctx *ctx, void *out);
```

```
   // All-in-one convenience function.
   int blake2b(void *out, size_t outlen,   // return buffer for digest
       const void *key, size_t keylen,     // optional secret key
       const void *in, size_t inlen);      // data to be hashed

   #endif

   <CODE ENDS>
```

A.2. blake2b.c

```
   <CODE BEGINS>

   // blake2b.c
   // A simple BLAKE2b Reference Implementation

   #include "blake2b.h"

   // cyclic right rotation

   #ifndef ROTR64
   #define ROTR64(x, y)  (((x) >> (y)) ^ ((x) << (64 - (y))))
   #endif

   // little-endian byte access

   #define B2B_GET64(p)                            \
       (((uint64_t) ((uint8_t *) (p))[0]) ^        \
       (((uint64_t) ((uint8_t *) (p))[1]) << 8) ^  \
       (((uint64_t) ((uint8_t *) (p))[2]) << 16) ^ \
       (((uint64_t) ((uint8_t *) (p))[3]) << 24) ^ \
       (((uint64_t) ((uint8_t *) (p))[4]) << 32) ^ \
       (((uint64_t) ((uint8_t *) (p))[5]) << 40) ^ \
       (((uint64_t) ((uint8_t *) (p))[6]) << 48) ^ \
       (((uint64_t) ((uint8_t *) (p))[7]) << 56))

   // G Mixing function

   #define B2B_G(a, b, c, d, x, y) {   \
       v[a] = v[a] + v[b] + x;         \
       v[d] = ROTR64(v[d] ^ v[a], 32); \
       v[c] = v[c] + v[d];             \
       v[b] = ROTR64(v[b] ^ v[c], 24); \
       v[a] = v[a] + v[b] + y;         \
       v[d] = ROTR64(v[d] ^ v[a], 16); \
       v[c] = v[c] + v[d];             \
       v[b] = ROTR64(v[b] ^ v[c], 63); }
```

```
   // Initialization Vector

   static const uint64_t blake2b_iv[8] = {
       0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,
       0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,
       0x510E527FADE682D1, 0x9B05688C2B3E6C1F,
       0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179
   };

   // Compression function. "last" flag indicates last block.

   static void blake2b_compress(blake2b_ctx *ctx, int last)
   {
       const uint8_t sigma[12][16] = {
           { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
           { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
           { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
           { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
           { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
           { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
           { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
           { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
           { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
           { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 },
           { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
           { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 }
       };
       int i;
       uint64_t v[16], m[16];

       for (i = 0; i < 8; i++) {           // init work variables
           v[i] = ctx->h[i];
           v[i + 8] = blake2b_iv[i];
       }

       v[12] ^= ctx->t[0];                 // low 64 bits of offset
       v[13] ^= ctx->t[1];                 // high 64 bits
       if (last)                           // last block flag set ?
           v[14] = ~v[14];

       for (i = 0; i < 16; i++)            // get little-endian words
           m[i] = B2B_GET64(&ctx->b[8 * i]);

       for (i = 0; i < 12; i++) {          // twelve rounds
           B2B_G( 0, 4,  8, 12, m[sigma[i][ 0]], m[sigma[i][ 1]]);
           B2B_G( 1, 5,  9, 13, m[sigma[i][ 2]], m[sigma[i][ 3]]);
           B2B_G( 2, 6, 10, 14, m[sigma[i][ 4]], m[sigma[i][ 5]]);
           B2B_G( 3, 7, 11, 15, m[sigma[i][ 6]], m[sigma[i][ 7]]);
```

```
            B2B_G( 0, 5, 10, 15, m[sigma[i][ 8]], m[sigma[i][ 9]]);
            B2B_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
            B2B_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
            B2B_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
        }

        for( i = 0; i < 8; ++i )
            ctx->h[i] ^= v[i] ^ v[i + 8];
    }

    // Initialize the state. key is optional

    int blake2b_init(blake2b_ctx *ctx, size_t outlen,
        const void *key, size_t keylen)     // (keylen=0: no key)
    {
        size_t i;

        if (outlen == 0 || outlen > 64 || keylen > 64)
            return -1;                      // illegal parameters

        for (i = 0; i < 8; i++)             // state, "param block"
            ctx->h[i] = blake2b_iv[i];
        ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;

        ctx->t[0] = 0;                      // input count low word
        ctx->t[1] = 0;                      // input count high word
        ctx->c = 0;                         // pointer within buffer
        ctx->outlen = outlen;

        for (i = keylen; i < 128; i++)      // zero input block
            ctx->b[i] = 0;
        if (keylen > 0) {
            blake2b_update(ctx, key, keylen);
            ctx->c = 128;                   // at the end
        }

        return 0;
    }

    // update with new data

    void blake2b_update(blake2b_ctx *ctx,
        const void *in, size_t inlen)       // data bytes
    {
        size_t i;

        for (i = 0; i < inlen; i++) {
            if (ctx->c == 128) {            // buffer full ?
```

```
            ctx->t[0] += ctx->c;        // add counters
            if (ctx->t[0] < ctx->c)     // carry overflow ?
                ctx->t[1]++;            // high word
            blake2b_compress(ctx, 0);   // compress (not last)
            ctx->c = 0;                 // counter to zero
        }
        ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
    }
}

// finalize

void blake2b_final(blake2b_ctx *ctx, void *out)
{
    size_t i;

    ctx->t[0] += ctx->c;                // mark last block offset
    if (ctx->t[0] < ctx->c)             // carry overflow
        ctx->t[1]++;                    // high word

    while (ctx->c < 128)                // fill up with zeros
        ctx->b[ctx->c++] = 0;
    blake2b_compress(ctx, 1);           // final block flag = 1

    // little endian convert and store
    for (i = 0; i < ctx->outlen; i++) {
        ((uint8_t *) out)[i] =
            (ctx->h[i >> 3] >> (8 * (i & 7))) & 0xFF;
    }
}

// convenience function for all-in-one computation

int blake2b(void *out, size_t outlen,
    const void *key, size_t keylen,
    const void *in, size_t inlen)
{
    blake2b_ctx ctx;

    if (blake2b_init(&ctx, outlen, key, keylen))
        return -1;
    blake2b_update(&ctx, in, inlen);
    blake2b_final(&ctx, out);

    return 0;
}
<CODE ENDS>
```

Appendix B. BLAKE2s Implementation C Source

B.1. blake2s.h

```
<CODE BEGINS>

// blake2s.h
// BLAKE2s Hashing Context and API Prototypes

#ifndef BLAKE2S_H
#define BLAKE2S_H

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

// state context
typedef struct {
    uint8_t b[64];                      // input buffer
    uint32_t h[8];                      // chained state
    uint32_t t[2];                      // total number of bytes
    size_t c;                           // pointer for b[]
    size_t outlen;                      // digest size
} blake2s_ctx;

// Initialize the hashing context "ctx" with optional key "key".
//      1 <= outlen <= 32 gives the digest size in bytes.
//      Secret key (also <= 32 bytes) is optional (keylen = 0).
int blake2s_init(blake2s_ctx *ctx, size_t outlen,
    const void *key, size_t keylen);    // secret key

// Add "inlen" bytes from "in" into the hash.
void blake2s_update(blake2s_ctx *ctx,   // context
    const void *in, size_t inlen);      // data to be hashed

// Generate the message digest (size given in init).
//      Result placed in "out"
void blake2s_final(blake2s_ctx *ctx, void *out);

// All-in-one convenience function.
int blake2s(void *out, size_t outlen,   // return buffer for digest
    const void *key, size_t keylen,     // optional secret key
    const void *in, size_t inlen);      // data to be hashed

#endif

<CODE ENDS>
```

B.2. blake2s.c

```
<CODE BEGINS>

// blake2s.c
// A simple BLAKE2 Reference Implementation

#include "blake2s.h"

// cyclic right rotation

#ifndef ROTR32
#define ROTR32(x, y)  (((x) >> (y)) ^ ((x) << (32 - (y))))
#endif

// little-endian byte access
#define B2S_GET32(p)                            \
    (((uint32_t) ((uint8_t *) (p))[0]) ^        \
    (((uint32_t) ((uint8_t *) (p))[1]) << 8) ^  \
    (((uint32_t) ((uint8_t *) (p))[2]) << 16) ^ \
    (((uint32_t) ((uint8_t *) (p))[3]) << 24))

// Mixing function G
#define B2S_G(a, b, c, d, x, y) {   \
    v[a] = v[a] + v[b] + x;         \
    v[d] = ROTR32(v[d] ^ v[a], 16); \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR32(v[b] ^ v[c], 12); \
    v[a] = v[a] + v[b] + y;         \
    v[d] = ROTR32(v[d] ^ v[a], 8);  \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR32(v[b] ^ v[c], 7); }

// Initialization Vector

static const uint32_t blake2s_iv[8] =
{
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
};

// Compression function. "last" flag indicates last block.

static void blake2s_compress(blake2s_ctx *ctx, int last)
{
    const uint8_t sigma[10][16] = {
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
        { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
```

```
            { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
            { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
            { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
            { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
            { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
            { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
            { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
            { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 }
        };
        int i;
        uint32_t v[16], m[16];

        for (i = 0; i < 8; i++) {           // init work variables
            v[i] = ctx->h[i];
            v[i + 8] = blake2s_iv[i];
        }

        v[12] ^= ctx->t[0];                 // low 32 bits of offset
        v[13] ^= ctx->t[1];                 // high 32 bits
        if (last)                           // last block flag set ?
            v[14] = ~v[14];

        for (i = 0; i < 16; i++)            // get little-endian words
            m[i] = B2S_GET32(&ctx->b[4 * i]);

        for (i = 0; i < 10; i++) {          // ten rounds
            B2S_G( 0, 4,  8, 12, m[sigma[i][ 0]], m[sigma[i][ 1]]);
            B2S_G( 1, 5,  9, 13, m[sigma[i][ 2]], m[sigma[i][ 3]]);
            B2S_G( 2, 6, 10, 14, m[sigma[i][ 4]], m[sigma[i][ 5]]);
            B2S_G( 3, 7, 11, 15, m[sigma[i][ 6]], m[sigma[i][ 7]]);
            B2S_G( 0, 5, 10, 15, m[sigma[i][ 8]], m[sigma[i][ 9]]);
            B2S_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
            B2S_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
            B2S_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
        }

        for( i = 0; i < 8; ++i )
            ctx->h[i] ^= v[i] ^ v[i + 8];
    }

    // Initialize the state. key is optional

    int blake2s_init(blake2s_ctx *ctx, size_t outlen,
        const void *key, size_t keylen)     // (keylen=0: no key)
    {
        size_t i;

        if (outlen == 0 || outlen > 32 || keylen > 32)
```

```
        return -1;                      // illegal parameters

    for (i = 0; i < 8; i++)             // state, "param block"
        ctx->h[i] = blake2s_iv[i];
    ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;

    ctx->t[0] = 0;                      // input count low word
    ctx->t[1] = 0;                      // input count high word
    ctx->c = 0;                         // pointer within buffer
    ctx->outlen = outlen;

    for (i = keylen; i < 64; i++)       // zero input block
        ctx->b[i] = 0;
    if (keylen > 0) {
        blake2s_update(ctx, key, keylen);
        ctx->c = 64;                    // at the end
    }

    return 0;
}

// update with new data

void blake2s_update(blake2s_ctx *ctx,
    const void *in, size_t inlen)       // data bytes
{
    size_t i;

    for (i = 0; i < inlen; i++) {
        if (ctx->c == 64) {             // buffer full ?
            ctx->t[0] += ctx->c;        // add counters
            if (ctx->t[0] < ctx->c)     // carry overflow ?
                ctx->t[1]++;            // high word
            blake2s_compress(ctx, 0);   // compress (not last)
            ctx->c = 0;                 // counter to zero
        }
        ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
    }
}

// finalize

void blake2s_final(blake2s_ctx *ctx, void *out)
{
    size_t i;

    ctx->t[0] += ctx->c;                // mark last block offset
    if (ctx->t[0] < ctx->c)             // carry overflow
```

```
            ctx->t[1]++;                    // high word

        while (ctx->c < 64)                 // fill up with zeros
            ctx->b[ctx->c++] = 0;
        blake2s_compress(ctx, 1);           // final block flag = 1

        // little endian convert and store
        for (i = 0; i < ctx->outlen; i++) {
            ((uint8_t *) out)[i] =
                (ctx->h[i >> 2] >> (8 * (i & 3))) & 0xFF;
        }
    }

    // convenience function for all-in-one computation

    int blake2s(void *out, size_t outlen,
        const void *key, size_t keylen,
        const void *in, size_t inlen)
    {
        blake2s_ctx ctx;

        if (blake2s_init(&ctx, outlen, key, keylen))
            return -1;
        blake2s_update(&ctx, in, inlen);
        blake2s_final(&ctx, out);

        return 0;
    }

    <CODE ENDS>
```

Appendix C. BLAKE2b and BLAKE2s Self Test Module C Source

This module computes a series of keyed and unkeyed hashes from deterministically generated pseudo-random data, and computes a hash over those results. This is fairly a exhaustive, yet compact and fast method for verifying that the hashing module is functioning correctly.

Such testing is recommended especially when compiling the implementation for a new a target platform configuration. Furthermore, some security standards such as FIPS-140 may require a Power-On Self Test (POST) to be performed every time the cryptographic module is loaded [FIPS140-2IG].

```
    <CODE BEGINS>

    // test_main.c
```

```
   // Self test Modules for BLAKE2b and BLAKE2s -- and a stub main().

   #include <stdio.h>

   #include "blake2b.h"
   #include "blake2s.h"

   // Deterministic sequences (Fibonacci generator)

   static void selftest_seq(uint8_t *out, size_t len, uint32_t seed)
   {
       size_t i;
       uint32_t t, a , b;

       a = 0xDEAD4BAD * seed;              // prime
       b = 1;

       for (i = 0; i < len; i++) {         // fill the buf
           t = a + b;
           a = b;
           b = t;
           out[i] = (t >> 24) & 0xFF;
       }
   }

   // BLAKE2b self-test validation. Return 0 when OK.

   int blake2b_selftest()
   {
       // grand hash of hash results
       const uint8_t blake2b_res[32] = {
           0xC2, 0x3A, 0x78, 0x00, 0xD9, 0x81, 0x23, 0xBD,
           0x10, 0xF5, 0x06, 0xC6, 0x1E, 0x29, 0xDA, 0x56,
           0x03, 0xD7, 0x63, 0xB8, 0xBB, 0xAD, 0x2E, 0x73,
           0x7F, 0x5E, 0x76, 0x5A, 0x7B, 0xCC, 0xD4, 0x75
       };
       // parameter sets
       const size_t b2b_md_len[4] = { 20, 32, 48, 64 };
       const size_t b2b_in_len[6] = { 0, 3, 128, 129, 255, 1024 };

       size_t i, j, outlen, inlen;
       uint8_t in[1024], md[64], key[64];
       blake2b_ctx ctx;

       // 256-bit hash for testing
       if (blake2b_init(&ctx, 32, NULL, 0))
           return -1;
```

```
        for (i = 0; i < 4; i++) {
            outlen = b2b_md_len[i];
            for (j = 0; j < 6; j++) {
                inlen = b2b_in_len[j];

                selftest_seq(in, inlen, inlen);     // unkeyed hash
                blake2b(md, outlen, NULL, 0, in, inlen);
                blake2b_update(&ctx, md, outlen);   // hash the hash

                selftest_seq(key, outlen, outlen);  // keyed hash
                blake2b(md, outlen, key, outlen, in, inlen);
                blake2b_update(&ctx, md, outlen);   // hash the hash
            }
        }

        // compute and compare the hash of hashes
        blake2b_final(&ctx, md);
        for (i = 0; i < 32; i++) {
            if (md[i] != blake2b_res[i])
                return -1;
        }

        return 0;
    }

    // BLAKE2s self-test validation. Return 0 when OK.

    int blake2s_selftest()
    {
        // grand hash of hash results
        const uint8_t blake2s_res[32] = {
            0x6A, 0x41, 0x1F, 0x08, 0xCE, 0x25, 0xAD, 0xCD,
            0xFB, 0x02, 0xAB, 0xA6, 0x41, 0x45, 0x1C, 0xEC,
            0x53, 0xC5, 0x98, 0xB2, 0x4F, 0x4F, 0xC7, 0x87,
            0xFB, 0xDC, 0x88, 0x79, 0x7F, 0x4C, 0x1D, 0xFE
        };
        // parameter sets
        const size_t b2s_md_len[4] = { 16, 20, 28, 32 };
        const size_t b2s_in_len[6] = { 0,  3,  64, 65, 255, 1024 };

        size_t i, j, outlen, inlen;
        uint8_t in[1024], md[32], key[32];
        blake2s_ctx ctx;

        // 256-bit hash for testing
        if (blake2s_init(&ctx, 32, NULL, 0))
            return -1;
```

```
        for (i = 0; i < 4; i++) {
            outlen = b2s_md_len[i];
            for (j = 0; j < 6; j++) {
                inlen = b2s_in_len[j];

                selftest_seq(in, inlen, inlen);     // unkeyed hash
                blake2s(md, outlen, NULL, 0, in, inlen);
                blake2s_update(&ctx, md, outlen);   // hash the hash

                selftest_seq(key, outlen, outlen);  // keyed hash
                blake2s(md, outlen, key, outlen, in, inlen);
                blake2s_update(&ctx, md, outlen);   // hash the hash
            }
        }

        // compute and compare the hash of hashes
        blake2s_final(&ctx, md);
        for (i = 0; i < 32; i++) {
            if (md[i] != blake2s_res[i])
                return -1;
        }

        return 0;
    }

    // test driver

    int main(int argc, char **argv)
    {
        printf("blake2b_selftest() = %s\n",
             blake2b_selftest() ? "FAIL" : "OK");
        printf("blake2s_selftest() = %s\n",
             blake2s_selftest() ? "FAIL" : "OK");

        return 0;
    }

    <CODE ENDS>
```

Authors' Addresses

Markku-Juhani O. Saarinen (editor)
Independent Consultant

Email: mjos@iki.fi
URI: https://mjos.fi

Jean-Philippe Aumasson
Kudelski Security
22-24, Route de Geneve
Case Postale 134
Cheseaux 1033
Switzerland

Email: jean-philippe.aumasson@nagra.com
URI: https://www.kudelskisecurity.com