Independent Submission
Request for Comments: 7693
Category: Informational
ISSN: 2070-1721

M-J. Saarinen, Ed.
Queen's University Belfast
J-P. Aumasson
Kudelski Security

November 2015

The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)

Abstract

This document describes the cryptographic hash function BLAKE2 and makes 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 document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7693.

Table of Contents

1. Introduction and Terminology

The BLAKE2 cryptographic hash function [BLAKE2] 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 perform well on any platform, software, or hardware. BLAKE2 does not require a special "HMAC" (Hashed Message Authentication Code) construction for keyed message authentication as it has 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].

To aid implementation, we provide a trace of BLAKE2b-512 hash computation in Appendix A and a trace of BLAKE2s-256 hash computation in Appendix B. Due to space constraints, this document does not contain a full set of test vectors for BLAKE2.

A reference implementation in C programming language for BLAKE2b can be found in Appendix C and for BLAKE2s in Appendix D of this document. These implementations MAY be validated with the more exhaustive Test Module contained in Appendix E.

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

These variables are used in the algorithm description:

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. Each has "bb" bytes.

t Message 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 the following functions:

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".

2.4. Little-Endian Interpretation of Words as Bytes

All mathematical operations are on 64-bit words in BLAKE2b and on 32-bit words in 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 are 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 a numerical value of 0x67452301 or 1732584193.

When interpreted as a 64-bit word, bytes x[0..7] have a numerical value of 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 the parameter block, "kk", specifies the key size in bytes. Set kk = 00 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 that 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 also be found in implementations in Appendices C and D 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", message block vector "m" (last block is padded with zeros to full block size, if required), 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 Appendices C and D for reference C language implementations of BLAKE2b and BLAKE2s, respectively.

Key and data input are 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).

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

The following procedure processes the padded data blocks into an "nn"-byte final hash value. See Section 2 for a 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 := F( h, d[i], (i + 1) * bb, FALSE )
|     |       END FOR.
|     END IF.
|
|     // Final block.
|     IF kk = 0 THEN
|     |       h := F( h, d[dd - 1], ll, TRUE )
|     ELSE
|     |       h := F( 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 MAY support the following digest size parameters for interoperability (e.g., digital signatures), as long as a sufficient level of security is attained by the parameter selections. These parameters and identifiers are intended to be suitable as drop-in replacements to MD5 and corresponding SHA algorithms.

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. The same OID can be used for both keyed and unkeyed hashing since in the latter case the key simply has zero length.

Algorithm Identifier	Target Arch	Collision Security	Hash nn	Hash ASN.1 OID Suffix
id-blake2b160	64-bit	2**80	20	x.1.5
id-blake2b256	64-bit	2**128	32	x.1.8
id-blake2b384	64-bit	2**192	48	x.1.12
id-blake2b512	64-bit	2**256	64	x.1.16
id-blake2s128	32-bit	2**64	16	x.2.4
id-blake2s160	32-bit	2**80	20	x.2.5
id-blake2s224	32-bit	2**112	28	x.2.7
id-blake2s256	32-bit	2**128	32	x.2.8

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

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

-- BLAKE2b Identifiers --
id-blake2b160 OBJECT IDENTIFIER ::= { blake2b 5 }
id-blake2b256 OBJECT IDENTIFIER ::= { blake2b 8 }
id-blake2b384 OBJECT IDENTIFIER ::= { blake2b 12 }
id-blake2b512 OBJECT IDENTIFIER ::= { blake2b 16 }
```

```
        -- BLAKE2s Identifiers --
        id-blake2s128 OBJECT IDENTIFIER ::= { blake2s 4 }
        id-blake2s160 OBJECT IDENTIFIER ::= { blake2s 5 }
        id-blake2s224 OBJECT IDENTIFIER ::= { blake2s 7 }
        id-blake2s256 OBJECT IDENTIFIER ::= { blake2s 8 }
```

5. 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 Appendices C and D may not erase all sensitive data (such as secret keys) immediately from process memory after use. Such cleanup can be added if needed.

6. References

6.1. Normative References

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

6.2. Informative References

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

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

[FIPS140-2IG] NIST, "Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program", September 2015, <http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf/>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

Appendix A. Example of BLAKE2b Computation

We compute the unkeyed hash of three ASCII bytes "abc" with BLAKE2b-512 and show internal values during computation.

```
       m[16] = 0000000000636261 0000000000000000 0000000000000000
               0000000000000000 0000000000000000 0000000000000000
               0000000000000000 0000000000000000 0000000000000000
               0000000000000000 0000000000000000 0000000000000000
               0000000000000000 0000000000000000 0000000000000000
               0000000000000000

(i= 0) v[16] = 6A09E667F2BDC948 BB67AE8584CAA73B 3C6EF372FE94F82B
               A54FF53A5F1D36F1 510E527FADE682D1 9B05688C2B3E6C1F
               1F83D9ABFB41BD6B 5BE0CD19137E2179 6A09E667F3BCC908
               BB67AE8584CAA73B 3C6EF372FE94F82B A54FF53A5F1D36F1
               510E527FADE682D2 9B05688C2B3E6C1F E07C265404BE4294
               5BE0CD19137E2179

(i= 1) v[16] = 86B7C1568029BB79 C12CBCC809FF59F3 C6A5214CC0EACA8E
               0C87CD524C14CC5D 44EE6039BD86A9F7 A447C850AA694A7E
               DE080F1BB1C0F84B 595CB8A9A1ACA66C BEC3AE837EAC4887
               6267FC79DF9D6AD1 FA87B01273FA6DBE 521A715C63E08D8A
               E02D0975B8D37A83 1C7B754F08B7D193 8F885A76B6E578FE
               2318A24E2140FC64

(i= 2) v[16] = 53281E83806010F2 3594B403F81B4393 8CD63C7462DE0DFF
               85F693F3DA53F974 BAABDBB2F386D9AE CA5425AEC65A10A8
               C6A22E2FF0F7AA48 C6A56A51CB89C595 224E6A3369224F96
               500E125E58A92923 E9E4AD0D0E1A0D48 85DF9DC143C59A74
               92A3AAAA6D952B7F C5FDF71090FAE853 2A8A40F15A462DD0
               572D17EFFDD37358

(i= 3) v[16] = 60ED96AA7AD41725 E46A743C71800B9D 1A04B543A01F156B
               A2F8716E775C4877 DA0A61BCDE4267EA B1DD230754D7BDEE
               25A1422779E06D14 E6823AE4C3FF58A5 A1677E19F37FD5DA
               22BDCE6976B08C51 F1DE8696BEC11BF1 A0EBD586A4A1D2C8
               C804EBAB11C99FA9 8E0CEC959C715793 7C45557FAE0D4D89
               716343F52FDD265E

(i= 4) v[16] = BB2A77D3A8382351 45EB47971F23B103 98BE297F6E45C684
               A36077DEE3370B89 8A03C4CB7E97590A 24192E49EBF54EA0
               4F82C9401CB32D7A 8CCD013726420DC4 A9C9A8F17B1FC614
               55908187977514A0 5B44273E66B19D27 B6D5C9FCA2579327
               086092CFB858437E 5C4BE2156DBEECF9 2EFEDE99ED4EFF16
               3E7B5F234CD1F804
```

```
(i= 5) v[16] = C79C15B3D423B099 2DA2224E8DA97556 77D2B26DF1C45C55
               8934EB09A3456052 0F6D9EEED157DA2A 6FE66467AF88C0A9
               4EB0B76284C7AAFB 299C8E725D954697 B2240B59E6D567D3
               2643C2370E49EBFD 79E02EEF20CDB1AE 64B3EED7BB602F39
               B97D2D439E4DF63D C718E755294C9111 1F0893F2772BB373
               1205EA4A7859807D

(i= 6) v[16] = E58F97D6385BAEE4 7640AA9764DA137A DEB4C7C23EFE287E
               70F6F41C8783C9F6 7127CD48C76A7708 9E472AF0BE3DB3F6
               0F244C62DDF71788 219828AA83880842 41CCA9073C8C4D0D
               5C7912BC10DF3B4B A2C3ABBD37510EE2 CB5668CC2A9F7859
               8733794F07AC1500 C67A6BE42335AA6F ACB22B28681E4C82
               DB2161604CBC9828

(i= 7) v[16] = 6E2D286EEADEDC81 BCF02C0787E86358 57D56A56DD015EDF
               55D899D40A5D0D0A 819415B56220C459 B63C479A6A769F02
               258E55E0EC1F362A 3A3B4EC60E19DFDC 04D769B3FCB048DB
               B78A9A33E9BFF4DD 5777272AE1E930C0 5A387849E578DBF6
               92AAC307CF2C0AFC 30AACCC4F06DAFAA 483893CC094F8863
               E03C6CC89C26BF92

(i= 8) v[16] = FFC83ECE76024D01 1BE7BFFB8C5CC5F9 A35A18CBAC4C65B7
               B7C2C7E6D88C285F 81937DA314A50838 E1179523A2541963
               3A1FAD7106232B8F 1C7EDE92AB8B9C46 A3C2D35E4F685C10
               A53D3F73AA619624 30BBCC0285A22F65 BCEFBB6A81539E5D
               3841DEF6F4C9848A 98662C85FBA726D4 7762439BD5A851BD
               B0B9F0D443D1A889

(i= 9) v[16] = 753A70A1E8FAEADD 6B0D43CA2C25D629 F8343BA8B94F8C0B
               BC7D062B0DB5CF35 58540EE1B1AEBC47 63C5B9B80D294CB9
               490870ECAD27DEBD B2A90DDF667287FE 316CC9EBEEFAD8FC
               4A466BCD021526A4 5DA7F7638CEC5669 D9C8826727D306FC
               88ED6C4F3BD7A537 19AE688DDF67F026 4D8707AAB40F7E6D
               FD3F572687FEA4F1

(i=10) v[16] = E630C747CCD59C4F BC713D41127571CA 46DB183025025078
               6727E81260610140 2D04185EAC2A8CBA 5F311B88904056EC
               40BD313009201AAB 0099D4F82A2A1EAB 6DD4FBC1DE60165D
               B3B0B51DE3C86270 900AEE2F233B08E5 A07199D87AD058D8
               2C6B25593D717852 37E8CA471BEAA5F8 2CFC1BAC10EF4457
               01369EC18746E775

(i=11) v[16] = E801F73B9768C760 35C6D22320BE511D 306F27584F65495E
               B51776ADF569A77B F4F1BE86690B3C34 3CC88735D1475E4B
               5DAC67921FF76949 1CDB9D31AD70CC4E 35BA354A9C7DF448
               4929CBE45679D73E 733D1A17248F39DB 92D57B736F5F170A
               61B5C0A41D491399 B5C333457E12844A BD696BE010D0D889
               02231E1A917FE0BD
```

```
(i=12) v[16] = 12EF8A641EC4F6D6 BCED5DE977C9FAF5 733CA476C5148639
               97DF596B0610F6FC F42C16519AD5AFA7 AA5AC1888E10467E
               217D930AA51787F3 906A6FF19E573942 75AB709BD3DCBF24
               EE7CE1F345947AA4 F8960D6C2FAF5F5E E332538A36B6D246
               885BEF040EF6AA0B A4939A417BFB78A3 646CBB7AF6DCE980
               E813A23C60AF3B82

        h[8] = 0D4D1C983FA580BA E9F6129FB697276A B7C45A68142F214C
               D1A2FFDB6FBB124B 2D79AB2A39C5877D 95CC3345DED552C2
               5A92F1DBA88AD318 239900D4ED8623B9

BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

Appendix B. Example of BLAKE2s Computation

We compute the unkeyed hash of three ASCII bytes "abc" with BLAKE2s-256 and show internal values during computation.

```
       m[16] = 00636261 00000000 00000000 00000000 00000000 00000000
               00000000 00000000 00000000 00000000 00000000 00000000
               00000000 00000000 00000000 00000000

(i=0)  v[16] = 6B08E647 BB67AE85 3C6EF372 A54FF53A 510E527F 9B05688C
               1F83D9AB 5BE0CD19 6A09E667 BB67AE85 3C6EF372 A54FF53A
               510E527C 9B05688C E07C2654 5BE0CD19

(i=1)  v[16] = 16A3242E D7B5E238 CE8CE24B 927AEDE1 A7B430D9 93A4A14E
               A44E7C31 41D4759B 95BF33D3 9A99C181 608A3A6B B666383E
               7A8DD50F BE378ED7 353D1EE6 3BB44C6B

(i=2)  v[16] = 3AE30FE3 0982A96B E88185B4 3E339B16 F24338CD 0E66D326
               E005ED0C D591A277 180B1F3A FCF43914 30DB62D6 4847831C
               7F00C58E FB847886 C544E836 524AB0E2

(i=3)  v[16] = 7A3BE783 997546C1 D45246DF EDB5F821 7F98A742 10E864E2
               D4AB70D0 C63CB1AB 6038DA9E 414594B0 F2C218B5 8DA0DCB7
               D7CD7AF5 AB4909DF 85031A52 C4EDFC98

(i=4)  v[16] = 2A8B8CB7 1ACA82B2 14045D7F CC7258ED 383CF67C E090E7F9
               3025D276 57D04DE4 994BACF0 F0982759 F17EE300 D48FC2D5
               DC854C10 523898A9 C03A0F89 47D6CD88

(i=5)  v[16] = C4AA2DDB 111343A3 D54A700A 574A00A9 857D5A48 B1E11989
               6F5C52DF DD2C53A3 678E5F8E 9718D4E9 622CB684 92976076
               0E41A517 359DC2BE 87A87DDD 643F9CEC
```

```
   (i=6)  v[16] = 3453921C D7595EE1 592E776D 3ED6A974 4D997CB3 DE9212C3
                  35ADF5C9 9916FD65 96562E89 4EAD0792 EBFC2712 2385F5B2
                  F34600FB D7BC20FB EB452A7B ECE1AA40

   (i=7)  v[16] = BE851B2D A85F6358 81E6FC3B 0BB28000 FA55A33A 87BE1FAD
                  4119370F 1E2261AA A1318FD3 F4329816 071783C2 6E536A8D
                  9A81A601 E7EC80F1 ACC09948 F849A584

   (i=8)  v[16] = 07E5B85A 069CC164 F9DE3141 A56F4680 9E440AD2 9AB659EA
                  3C84B971 21DBD9CF 46699F8C 765257EC AF1D998C 75E4C3B6
                  523878DC 30715015 397FEE81 4F1FA799

   (i=9)  v[16] = 435148C4 A5AA2D11 4B354173 D543BC9E BDA2591C BF1D2569
                  4FCB3120 707ADA48 565B3FDE 32C9C916 EAF4A1AB B1018F28
                  8078D978 68ADE4B5 9778FDA3 2863B92E

   (i=10) v[16] = D9C994AA CFEC3AA6 700D0AB2 2C38670E AF6A1F66 1D023EF3
                  1D9EC27D 945357A5 3E9FFEBD 969FE811 EF485E21 A632797A
                  DEEF082E AF3D80E1 4E86829B 4DEAFD3A

          h[8] = 8C5E8C50 E2147C32 A32BA7E1 2F45EB4E 208B4537 293AD69E
                  4C9B994D 82596786

   BLAKE2s-256("abc") = 50 8C 5E 8C 32 7C 14 E2 E1 A7 2B A3 4E EB 45 2F
                        37 45 8B 20 9E D6 3A 29 4D 99 9B 4C 86 67 59 82
```

Appendix C. BLAKE2b Implementation C Source

C.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>
```

C.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 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)        // (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;
    }

    // Add "inlen" bytes from "in" into the hash.

    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];
        }
    }

    // Generate the message digest (size given in init).
    //      Result placed in "out".

    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 D. BLAKE2s Implementation C Source

D.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>
```

D.2. blake2s.c

```
   <CODE BEGINS>
   // blake2s.c
   // A simple blake2s 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 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)     // (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;
    }
```

```
   // Add "inlen" bytes from "in" into the hash.

   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];
       }
   }

   // Generate the message digest (size given in init).
   //      Result placed in "out".

   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 E. BLAKE2b and BLAKE2s Self-Test Module C Source

This module computes a series of keyed and unkeyed hashes from deterministically generated pseudorandom data and computes a hash over those results. This is a fairly 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>
```

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.

Authors' Addresses

Markku-Juhani O. Saarinen (editor)
Queen's University Belfast
Centre for Secure Information Technologies, ECIT
Northern Ireland Science Park
Queen's Road, Queen's Island
Belfast BT3 9DT
United Kingdom

Email: m.saarinen@qub.ac.uk
URI: http://www.csit.qub.ac.uk

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