BLAKE2: simpler, smaller, fast as MD5

2012.12.21

https://blake2.net

Jean-Philippe Aumasson (@aumasson)
Samuel Neves (@sevenps)
Zooko Wilcox-O'Hearn (@zooko)
Christian Winnerlein (@codesinchaos)

Abstract. We present the cryptographic hash function BLAKE2, an improved version of the SHA-3 finalist BLAKE optimized for speed in software. Target applications include cloud storage, intrusion detection, or version control systems. BLAKE2 comes in two main flavors: BLAKE2b is optimized for 64-bit platforms, and BLAKE2s for smaller architectures. On 64-bit platforms, BLAKE2 is often faster than MD5, yet provides security similar to that of SHA-3. We specify parallel versions BLAKE2bp and BLAKE2sp that are up to 4 and 8 times faster, by taking advantage of SIMD and/or multiple cores. BLAKE2 has more benefits than just speed: BLAKE2 uses up to 32% less RAM than BLAKE, and comes with a comprehensive tree-hashing mode as well as an efficient MAC mode.

1 Introduction

The SHA-3 Competition succeeded in selecting a hash function that complements SHA-2 and is much faster than SHA-2 in hardware [10]. There is nevertheless a demand for fast software hashing for applications such as integrity checking and deduplication in filesystems and cloud storage, host-based intrusion detection, version control systems, or secure boot schemes.

SHA-3 does not fit these needs well—for example on Qualcomm's Krait microarchitecture[1] SHA-3-256 takes about 20% longer to hash a message than SHA-256 does, and on Intel's Ivy Bridge microarchitecture[2] SHA-3-512 takes about twice as long as SHA-512 does.

Many systems use faster algorithms like MD5, SHA-1, or a custom function to meet their speed requirements, even though those functions may be insecure. MD5 is famously vulnerable to collision and length-extension attacks [12, 22], but it is 2.53 times as fast as SHA-256 on Ivy Bridge and 2.98 times as fast as SHA-256 on Krait.

Despite MD5's significant security flaws, it continues to be among the most widely-used algorithms for file identification and data integrity. To choose just a handful of examples, the OpenStack cloud storage system [21], the popular version control system Perforce, and the recent object storage system used internally in AOL [19] all rely on MD5 for data integrity. The venerable `md5sum` unix tool remains one of the most widely-used tools for data integrity checking. The Sun/Oracle ZFS filesystem includes the option of using SHA-256 for data integrity, but the default configuration is to instead use a non-cryptographic 256-bit checksum,

[1] See http://bench.cr.yp.to/results-hash.html#armeabi-h9dragon, accessed 7 Dec 2012.
[2] See http://bench.cr.yp.to/results-hash.html#amd64-hydra8, accessed 7 Dec 2012.

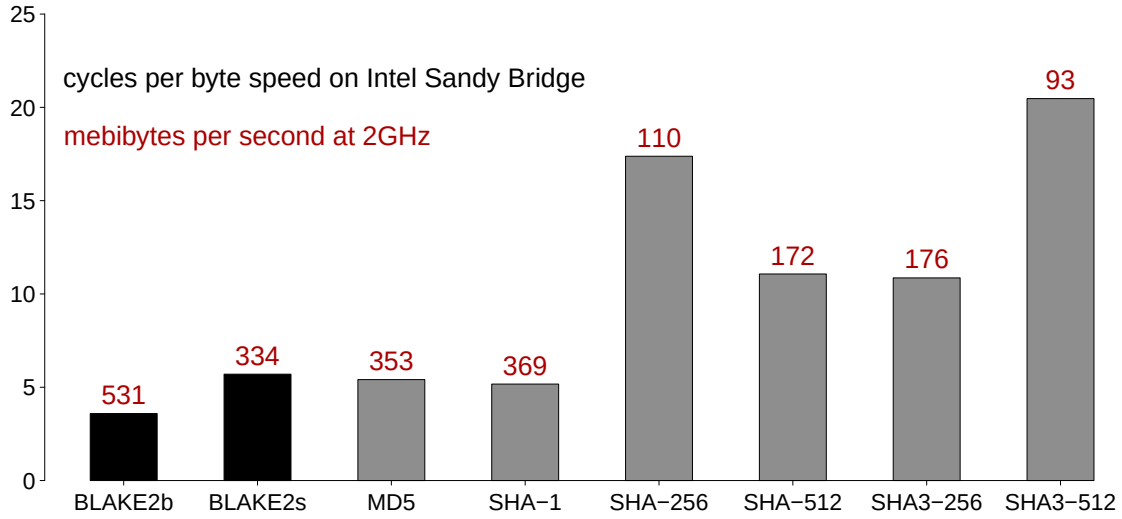


Figure 1: Speed comparison of various popular hash functions, taken from eBACS's "sandy" measurements. SHA-3 and BLAKE2 have no known security issues. SHA-1, MD5, SHA-256, and SHA-512 are susceptible to length-extension. SHA-1 and MD5 are vulnerable to collision. MD5 is vulnerable to cheap chosen-prefix collision.

for performance reasons. The Tahoe-LAFS distributed storage system uses SHA-256 for data integrity, but is investigating a faster hash function [13].

Some SHA-3 finalists outperform SHA-2 in software: for example, on Ivy Bridge BLAKE-512 is 1.41 times as fast as SHA-512, and BLAKE-256 is 1.70 times as fast as SHA-256. BLAKE-512 reaches 5.76 cycles per byte, or approximately 579 mebibytes per second, against 411 for SHA-512, on a CPU clocked at 3.5GHz.

BLAKE thus appears to be a good candidate for fast software hashing. Its security was evaluated by NIST in the SHA-3 process as having a "very large security margin", and the cryptanalysis published on BLAKE was noted as having "a great deal of depth" (see §4).

But as observed by Preneel [20], its design "reflects the state of the art in October 2008"; since then, and after extensive cryptanalysis, we have a better understanding of BLAKE's security and efficiency properties. We therefore introduce BLAKE2, an improved BLAKE with the following properties:

- **Faster than MD5** on 64-bit Intel platforms
- **32% less RAM** required than BLAKE
- Direct support, with no overhead, of
  - **Parallelism** for many-times faster hashing on multicore or SIMD CPUs
  - **Tree hashing** for incremental update or verification of large files
  - **Prefix-MAC** for authentication that is simpler and faster than HMAC
  - **Personalization** for defining a unique hash function for each application
- **Minimal padding**, which is faster and simpler to implement

The rest of the document is structured as follows: §2 describes BLAKE2, §3 discusses its efficiency on various platforms and reports preliminary benchmarks, and §4 argues that BLAKE2 is secure.

Public domain C and C# code of BLAKE2 is available on `https://blake2.net`. We are developing a tool `b2sum` similar to, and aiming to replace, `md5sum`.

2 Description of BLAKE2

BLAKE2 comes in two flavors:

- **BLAKE2b** (or just BLAKE2) is optimized for **64-bit platforms**—including NEON-enabled ARMs—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 are believed to be highly secure and have good performance on any platform, software or hardware. Each one is portable to any CPU, but can be up twice as fast when used on the CPU size for which it is optimized; for example, on a Tegra 2 (32-bit ARMv7-based SoC) BLAKE2s is expected to be about twice as fast as BLAKE2b, whereas on an AMD A10-5800K (64-bit, Piledriver microarchitecture), BLAKE2b is expected to be more than 1.5 times as fast as BLAKE2s.

Since BLAKE2 is very similar to BLAKE, we first describe the changes introduced with BLAKE2. The specification is complete with elements share with BLAKE in Appendix A. We refer to `https://131002.net/blake` for a complete specification of BLAKE.

2.1 Fewer rounds

BLAKE2b does **12 rounds**, and BLAKE2s does **10 rounds**, against 16 and 14 respectively for BLAKE. Based on the security analysis performed so far, and on reasonable assumptions on future progress, it is unlikely that 16 and 14 rounds are meaningfully more secure than 12 and 10 rounds. Recall that the initial BLAKE submission [1] had 14 and 10 rounds, respectively, and that the later increase [2] was motivated by the high speed of BLAKE.

This change gives a direct speed-up of about 25% and 29%, respectively, on long data. Speed on short data also significantly improves.

2.2 Rotations optimized for speed

The G function of BLAKE-512 performs four 64-bit word rotations of respectively 32, 25, 16, and 11 bits. **BLAKE2b replaces 25 with 24, and 11 with 63**:

- Using a 24-bit rotation allows SSSE3-capable CPUs to perform two rotations in parallel with a single SIMD instruction (namely, `pshufb`), whereas two shifts plus a logical OR are required for a rotation of 25 bits. This reduces the arithmetic cost of the G function, in recent Intel CPUs, from 18 single cycle instructions to 16 instructions, a 12% decrease.

- A 63-bit rotation can be implemented as an addition (doubling) and a shift followed by a logical OR. This provides a slight speed-up on platforms where addition and shift can be realized in parallel but not two shifts (i.e., some recent Intel CPUs). Additionally, since a rotation right by 63 is equal to a rotation left by 1, this may be slightly faster in some architectures where 1 is treated as a special case.

No platform suffers from these changes. For an in-depth analysis of optimized implementations of rotations, we refer to a previous work by two co-designers of BLAKE2 [17].

Past experiments by the BLAKE designers as well as third parties suggest that known differential attacks are unlikely to get significantly better, nor worse (cf. §4).

2.3 Minimal padding and finalization flags

BLAKE2 pads the last data block **if and only if necessary, with null bytes**. If the data length is a multiple of the block length, no padding byte is added.

BLAKE2 introduces **finalization flags** f_0 and f_1, as auxiliary inputs to the compression function:

- The security functionality of the padding is transferred to a finalization flag f_0, a word set to `ff...ff` if the block processed is the last, and to `00...00` otherwise. The flag f_0 is 64-bit for BLAKEb, and 32-bit for BLAKE2s.
- A second finalization flag f_1 is used to signal the last node of a layer in tree-hashing modes. When processing the last block—that is, when f_0 is `ff...ff`—the flag f_1 is also set to `ff...ff` if the node considered is the last, and to `00...00` otherwise.

The finalization flags are processed by the compression function as described in §2.4.

BLAKE2s thus supports hashing of data of at most $2^{64} - 1$ bytes, that is, almost 16 exbibytes (the amount of memory addressable by 64-bit processors). The upper bound for BLAKE2b is even more ridiculous, with up to $2^{128} - 1$ bytes supported.

2.4 Fewer constants

Whereas BLAKE used 8 word constants as IV plus 16 word constants for use in the compression function, BLAKE2 uses a total of **8 word constants, instead of 24**. This saves 128 ROM bytes and 128 RAM bytes in BLAKE2b implementations, and 64 ROM bytes and 64 RAM bytes in BLAKE2s implementations.

The compression function initialization phase is modified to:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 \oplus IV_4 & t_1 \oplus IV_5 & f_0 \oplus IV_6 & f_1 \oplus IV_7 \end{pmatrix}$$

Note the introduction of finalization flags f_0 and f_1, in place of BLAKE's redundant counter.

The G function of BLAKE2b is defined as:

$$\begin{aligned}
a &\leftarrow a + b + m_{\sigma_r(2i)} \\
d &\leftarrow (d \oplus a) \ggg 32 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 24 \\
a &\leftarrow a + b + m_{\sigma_r(2i+1)} \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 63
\end{aligned}$$

Note the aforementioned change of rotation counts.

Similarly, the G function of BLAKE2s is simplified to:

$$\begin{aligned}
a &\leftarrow a + b + m_{\sigma_r(2i)} \\
d &\leftarrow (d \oplus a) \ggg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 12 \\
a &\leftarrow a + b + m_{\sigma_r(2i+1)} \\
d &\leftarrow (d \oplus a) \ggg 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ggg 7
\end{aligned}$$

Omitting the constants in G gives an algorithm similar to the (unattacked) BLAZE toy version[3]. Constants in G initially aimed to guarantee early propagation of carries, but it turned out that the benefits (if any) are not worth the performance penalty. This change saves two xors and two loads per G, that is, 16% of the total arithmetic (addition and xor) instructions.

2.5 Little-endian

BLAKE, like SHA-1 and SHA-2, parses data blocks in the big-endian byte order. Like MD5, **BLAKE2 is little-endian**, because the large majority of target platforms is little-endian (AMD and Intel desktop processors, most mainstream ARM systems). Switching to little-endian may provide a slight speed-up, and often simplifies implementations.

Note that in BLAKE, the counter t is composed of two words t_0 and t_1, where t_0 holds the least significant bits of the integer encoded. This little-endian convention is preserved in BLAKE2.

2.6 Counter in bytes

The counter t counts **bytes rather than bits**. This simplifies implementations and reduce the risk of error, since target applications measure data volumes in bytes rather than bits. This change increases the amount of data that can be processed by 8 times, compared to BLAKE.

[3] See https://131002.net/blake/toyblake.pdf.

2.7 Salt processing

BLAKE's predecessor LAKE [3] introduced the built-in support for a salt, to simplify the use of randomized hashing within digital signature schemes.

In BLAKE2 the salt is processed as a one-time input to the hash function, through the IV, rather than as an input to each compression function. This simplifies the compression function, and saves a few instructions as well as a few bytes in RAM, since the salt doesn't have to be stored anymore. Using salt-independent compression functions has only negligible, and very theoretical, impact on security, as discussed in §4.

2.8 Parameter block

The parameter block of BLAKE2 is **xored with the IV** prior to the processing of the first data block. It encodes parameters for secure tree hashing, as well as key length (in keyed mode) and digest length.

The parameters are described below, and the block structure is shown in Tables 1 and 2:

- General parameters:
  - **Digest byte length** (1 byte): an integer in $[1, 64]$ for BLAKE2b, in $[1, 32]$ for BLAKE2s
  - **Key byte length** (1 byte): an integer in $[0, 64]$ for BLAKE2b, in $[0, 32]$ for BLAKE2s (set to 0 if no key is used)
  - **Salt** (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)
  - **Personalization** (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)
- Tree hashing parameters:
  - **Fanout** (1 byte): an integer in $[0, 255]$ (set to 0 if unlimited, and to 1 only in sequential mode)
  - **Maximal depth** (1 byte): an integer in $[1, 255]$ (set to 255 if unlimited, and to 1 only in sequential mode)
  - **Leaf maximal byte length** (4 bytes): an integer in $[0, 2^{32} - 1]$, that is, up to 4 GiB (set to 0 if unlimited, or in sequential mode)
  - **Node offset** (8 or 6 bytes): an integer in $[0, 2^{64} - 1]$ for BLAKE2b, and in $[0, 2^{48} - 1]$ for BLAKE2s (set to 0 for the first, leftmost, leaf, or in sequential mode)
  - **Node depth** (1 byte): an integer in $[0, 255]$ (set to 0 for the leaves, or in sequential mode)
  - **Inner hash byte length** (1 byte): an integer in $[0, 64]$ for BLAKE2b, and in $[0, 32]$ for BLAKE2s (set to 0 in sequential mode)

This is 50 bytes in total for BLAKE2b, and 32 bytes for BLAKE2s. Any bytes left are reserved for future and/or application-specific use, and are NULL. Values spanning more than one byte are written in **little-endian**. Note that tree hashing may be keyed, in which case leaf instances hash the key followed by a number of bytes equal to (at most) the maximal leaf length.

Offset	**0**	**1**	**2**	**3**
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8 **12**	Node offset			
16	Node depth	Inner length	RFU	
20 **24** **28**	RFU			
32 ... **44**	Salt			
48 ... **60**	Personalization			

Table 1: BLAKE2b parameter block structure (offsets in bytes).

Offset	**0**	**1**	**2**	**3**
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8	Node offset			
12	Node offset (cont.)		Node depth	Inner length
16 **20**	Salt			
24 **28**	Personalization			

Table 2: BLAKE2s parameter block structure (offsets in bytes).

Example parameter block of BLAKE2b. We take as example an instance of BLAKE2b with

- 64-byte digests, that is, with parameter digest length set to `40`,
- a 256-bit key, that is, with the parameter key length set to `20`,
- a salt set to the all-`55` string,
- a personalization set to the all-`ee` string.

BLAKE2b hashes data sequentially, thus tree parameters are set to the value specified for the sequential mode: fanout and maximal depth are set to `01`, leaf maximal length is set to `00000000`, node offset is set to `0000000000000000`, node depth and inner hash length are set to `00`.

The parameter block for this instance of BLAKE2b is thus the following[4]:

```
40200101 00000000 00000000 00000000  00000000 00000000 00000000 00000000
55555555 55555555 55555555 55555555  eeeeeeee eeeeeeee eeeeeeee eeeeeeee
```

Example parameter block of BLAKE2s. We take as example an instance of BLAKE2s with

- 32-byte digests, that is, with parameter digest length set to `20`,
- no key, that is, with the parameter key length set to `00`,
- no salt, and no personalization, that is, with all respective bytes set NULL.

BLAKE2s hashes data sequentially, thus tree parameters are set to the value specified for the sequential mode: fanout and maximal depth are set to `01`, leaf maximal length is set to `00000000`, node offset is set to `0000000000000000`, node depth and inner hash length are set to `00`.

The parameter block for this instance of BLAKE2s is thus

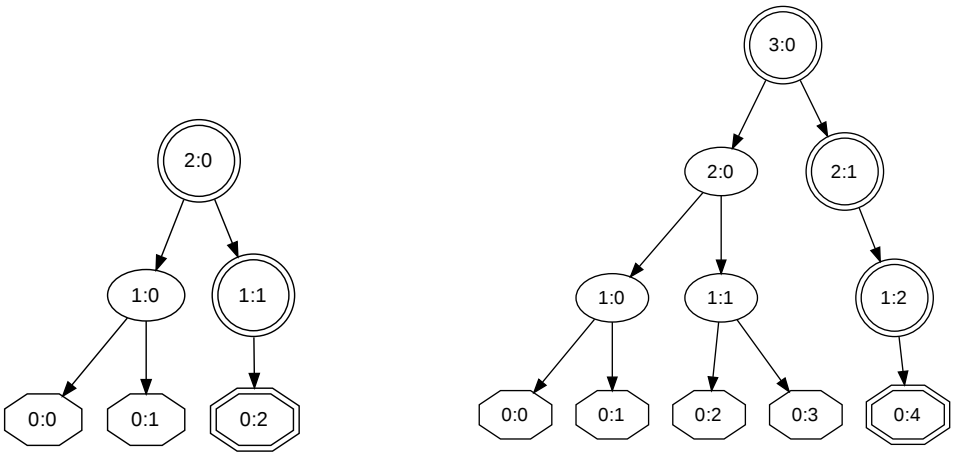
```
20000101 00000000 00000000 00000000  00000000 00000000 00000000 00000000
```

2.9 Keyed hashing (MAC and PRF)

When keyed (that is, when the field key length is non-zero), BLAKE2 sets the first data block to the key padded with zeros, the second data block to the first block of the message, the third block to the second block of the message, etc. Note that the padded key is treated as arbitrary data, therefore:

- The counter `t` includes the 64 (or 128) bytes of the key block, regardless of the key length.
- When hashing the empty message with a key, BLAKE2b and BLAKE2s make only one call to the compression function.

[4] For readability we add a space between each 4-byte block, however the value represented is a string of bytes, not a sequence of 4-byte words (which makes a difference with respect to endianness).



(a) Hashing 3 blocks: the tree has depth 3.

(b) Hashing 5 blocks: the tree has depth 4.

Figure 2: Layouts of tree hashing with fanout 2, and maximal depth at least 4.

The main application of keyed BLAKE2 is as a message authentication code (MAC): BLAKE2 can be used securely in prefix-MAC mode, thanks to the indifferentiability property inherited from BLAKE [9]. Prefix-MAC is faster than HMAC, as it saves at least one call to the compression function. Keyed BLAKE2 can also be used to instantiate PRFs, for example within the PBKDF2 password hashing scheme.

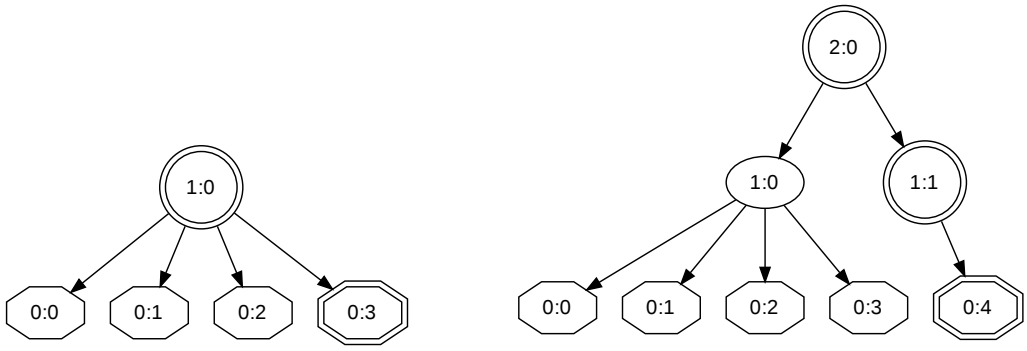
2.10 Tree hashing

The parameter block supports arbitrary tree hashing modes, be it binary or ternary trees, arbitrary-depth updatable tree hashing or fixed-depth parallel hashing, etc. Note that, unlike other functions, BLAKE2 does not restrict the leaf length and the fanout to be powers of 2.

Basic mechanism. Informally, tree hashing processes chunks of data of "leaf length" bytes independently of each other, then combines the respective hashes using a tree structure wherein each node takes as input the concatenation of "fanout" hashes. The "node offset" and "node depth" parameters ensure that each invocation to the hash function (leaf of internal node) uses a different hash function. The finalization flag $\mathtt{f}_1$ signals when a hash invocation is the last one at a given depth (where "last" is with respect to the node offset counter, for both leaves and intermediate nodes). The flag $\mathtt{f}_1$ can only be non-zero for the last block compressed within a hash invocation, and the root node always has $\mathtt{f}_1$ set to `ff...ff`.

The tree hashing mechanism is illustrated on Figures 2 and 3, which show layout of trees given different parameters and different input lengths. On those figures, octagons represent leaves (i.e., instances of the hash function processing input data), double-lined nodes (including leaves) are the last nodes of a layer, and thus have the flag $\mathtt{f}_1$ set). Labels "$\mathtt{i}$:$\mathtt{j}$" indicate a node's depth $\mathtt{i}$ and offset $\mathtt{j}$.

We refer to [6] for a comprehensive overview of secure tree hashing constructions.



(a) Hashing 4 blocks: the tree has depth 2.

(b) Hashing 5 blocks: the tree has depth 3.

Figure 3: Layouts of tree hashing with fanout 4, and maximal depth at least 3.

Message parsing. Unless specified otherwise, we recommend that data be parsed as contiguous blocks: for example, if leaf length is 1024 bytes, then the first 1024-byte data block is processed by the leaf with offset 0, the subsequent 1024-byte data block is processed by the leaf with offset 1, etc.

Special cases. We highlight some special cases of tree hashing:

- **Unlimited fanout**: When the fanout is unlimited (parameter set to 0), then the root node hashes the concatenation of as many leaves are required to process the message, as shown on Figure 4. That is, the depth of the tree is always 2, regardless of the maximal depth parameter. Nevertheless, changing the maximal depth parameter changes the final hash value returned. We thus recommend to set the depth parameter to 2.
- **Dealing with saturated trees**: If a tree hashing instance has fanout $\mathrm{f} \geq 2$, maximal depth $\mathrm{d} \geq 2$, and leaf maximal length $\ell \geq 1$ bytes, then up to $\mathrm{f}^{\mathrm{d}-1} \cdot \ell$ can be processed within a single tree. If more bytes have to be hashed, the fanout of the root node is extended to hash as many digests as necessary to respect the depth limit. This mechanism is illustrated on Figure 5. Note that if the maximal depth is 2, then the value does not affect the layout of the tree, which is identical to that of a tree hash with unlimited fanout (see Figure 4).

Generic tree parameters. Tree parameters supported by the parameter block allow for a wide range of implementation trade-offs, for example to efficiently support updatable hashing, which is typically an advantage when hashing many (small) chunks of data.

Although optimal performance will be reached by choosing the parameters specific to one's application, we specify the following parameters for a **generic tree mode**: binary tree (i.e., fanout 2), unlimited depth, and leaves of 4 KiB (the typical size of a memory page).

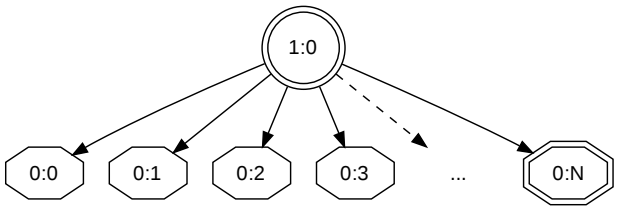


Figure 4: Tree hashing with unbounded fanout (0) and arbitrary maximal depth (de facto, 2).

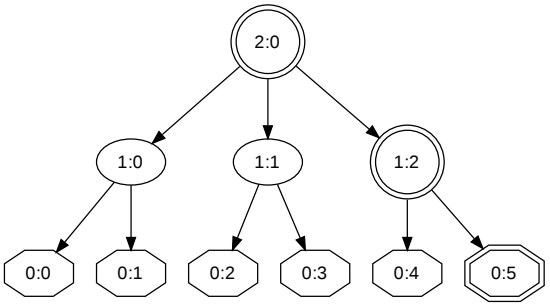


Figure 5: Tree hashing with maximal depth 3, fanout 2, but a root with larger fanout due to the reach of the maximal depth.

Updatable hashing example. Assume one has to provide a digest of a 1-tebibyte filesystem disk image that is updated every day. Instead of recomputing the digest by reading all the 2^{40} bytes, one can use our generic tree mode to implement an updatable hashing scheme:

1. Apply the generic tree mode, and store the $2^{40}/4096 = 2^{28}$ hashes from the leaves as well as the $2^{28} - 2$ intermediate hashes
2. When a leaf is changed, update the final digest by recomputing the 28 intermediate hashes

If BLAKE2b is used with intermediate hashes of 32 bytes, and that it hashes at a rate of 500 mebibytes per second, then step 1 takes approximately 35 minutes and generates about 16 gibibytes of intermediate data, whereas step 2 is instantaneous.

Note however that much less data may be stored: For many applications it is preferable to only store the intermediate hashes for larger pieces of data (without increasing the leaf size), which reduces memory requirement by only storing "higher" intermediate values. For example, storing intermediate values for 4 MiB chunks instead of all 4 KiB leaves reduces the storage to only 16 MiB. Indeed, using 4 KiB leaves allows allows applications with different piece sizes (as long as they are powers-of-two of at least 4 KiB) to produce the same root hash, while allowing them to make different granularity vs. storage trade-offs.

2.11 Parallel hashing: BLAKE2sp and BLAKE2bp

We specify 2 parallel hash functions (that is, with depth 2 and unlimited leaf length):

- **BLAKE2bp** runs **4 instances** of BLAKE2b in parallel
- **BLAKE2sp** runs **8 instances** of BLAKE2s in parallel

These functions use a different **parsing rule** than the default one in §§2.10: The first instance (node offset 0) hashes the message composed of the concatenation of all message blocks of index zero modulo 4; the second instance (node offset 1) hashes blocks of index 1 modulo 4, etc. Note that when the leaf length is unlimited, parsing the input as contiguous blocks would require the knowledge of the input length before any parallel operation, which is undesirable (e.g. when hashing a stream of data of undefined length, or a file received over a network).

When hashing one single large file, and when incrementability is not required, such parallel modes with unlimited leaves length seems the most appropriate, since

- They **minimize the computation overhead** by doing only one non-leaf call to the sequential hash function
- They **maximize the usage of the CPU** by keeping multiple cores and instruction pipelines busy simultaneously
- They require **realistic bandwidth and memory**

Note that parallel hashes have exactly the same interfaces as their sequential counterparts; for example, for BLAKE2bp one can define

```
blake2bp( uint8_t *out, uint8_t *in, uint8_t *key, int outlen, int inlen, int keylen )
```

Within a parallel hash, the same parameter block, except for the node offset, is used for all 4 or 8 instance of the sequential hash. For example, with no key, no salt, and no personalization, a version of BLAKE2sp producing 32-byte digests uses the four following parameters blocks for the four leaves:

```
20000802 00000000 00000000 00000020  00000000 00000000 00000000 00000000
20000802 00000000 01000000 00000020  00000000 00000000 00000000 00000000
20000802 00000000 02000000 00000020  00000000 00000000 00000000 00000000
20000802 00000000 03000000 00000020  00000000 00000000 00000000 00000000
```

Here the fanout is set to `08`, the depth is set to `02`, the leaf length is set to `00` (unlimited), the node depth is set to `00` (leaves), and the inner hash length is set to `20`. The node offset ranges from 0 to 7, and is little-endian encoded to, e.g., the byte string `07000000` (representing the integer `00000007`). Note that the last node (offset 7) sets the finalization flag f_1 in its last call to the compression function.

Finally, the root hash function in this version of BLAKE2sp uses the following parameter block (note the node depth set to `01`):

```
20000802 00000000 00000000 00000120  00000000 00000000 00000000 00000000
```

3 Performance

BLAKE2 is much faster than BLAKE, mainly due to its reduced number of rounds. On long messages, the BLAKE2b and BLAKE2s versions are expected to be approximately 25% and 29% faster, ignoring any savings from the absence of constants, optimized rotations, or little-endian conversion. The parallel versions BLAKE2bp and BLAKE2sp are expected to be 4 and 8 times faster than BLAKE2b and BLAKE2s on long messages, when implemented with multiple threads on a CPU with 4 or more cores (as most desktop and server processors: AMD FX-8150, Intel Core i5-2400S, etc.). Parallel hashing also benefits from advanced CPU technologies, as previously observed [18, §5.2].

3.1 Why BLAKE2 is fast in software

BLAKE2, along with its parallel variant, can take advantage of the following architectural features, or combinations thereof:

Instruction-level parallelism. Most modern processors are superscalar, that is, able to run several instructions per cycle through pipelining, out-of-order execution, and other related techniques. BLAKE2 has a natural instruction parallelism of 4 instructions within the G function; processors that are able to handle more instruction-level parallelism can do so in BLAKE2bp, by interleaving independent compression function calls. Examples of processors with notorious amount of instruction parallelism are Intel's Core 2, i7, and Itanium or AMD's K10, Bulldozer, and Piledriver.

SIMD instructions. Initially designed to speed up multimedia tasks, many modern processors contain *vector units*, which enable SIMD processing of data. Again, BLAKE2 can take advantage of vector units not only in its G function, but also in tree modes (such as the mode proposed in §§2.11), by running several compression instances within vector registers. Microarchitectures with SIMD capabilities are found in recent Intel and AMD CPUs, NEON-extended ARM-based SoC, PowerPC and Cell CPUs.

Multiple cores. Limits in both semiconductor manufacturing processes, as well as instruction-level parallelism have driven CPU manufacturers towards yet another kind of coarse-grained parallelism, where multiple independent CPUs are placed inside the same die, and enable the programmer to get thread-level parallelism. While sequential BLAKE2 does not take advantage of this, the parallel mode described in §§2.11, and other tree modes, can run each intermediate hashing in its own thread. Candidate processors for this approach are recent Intel and AMD chips, the IBM Cell, and recent ARM, UltraSPARC and Loongson models.

3.2 64-bit CPUs

To test our performance expectations, we implemented optimized versions of BLAKE2 that take advantage of the AVX and XOP instruction sets. Table 3 reports the timings obtained, in the following recent CPUs:

- AMD FX-8150 ("Bulldozer" microarchitecture), 4×3.6GHz

Microarchitecture	BLAKE2b			BLAKE2s		
	Long	1536	64	Long	1536	64
Sandy Bridge	3.59	3.96	8.56	5.70	5.74	6.63
Bulldozer	5.47	5.78	18.48	8.67	8.75	10.61

Table 3: Speed, in cycles per byte, of BLAKE2 in sequential mode.

Microarchitecture	BLAKE2b			BLAKE2s		
	Long	1536	64	Long	1536	64
Sandy Bridge (@2GHz)	531	482	223	335	332	288
Bulldozer (@3.6GHz)	628	330	103	220	330	180

Table 4: Speed, in mebibytes per second, of BLAKE2 in sequential mode (at the nominal frequency of each CPU).

- Intel Core i7-2630QM ("Sandy Bridge" microarchitecture), 4×2GHz

We used a methodology similar to that of SUPERCOP to measure speed on 64- and 1536-byte input as well as long data, on a single core. To ensure accurate measurements, we disabled dynamic overclocking features active by default on each of those CPUs. Furthermore, Table 4 reports the actual hashing speeds experienced on those CPUs, when running at the default frequency.

Compared to the best known timings for BLAKE [18],

- On Sandy Bridge, BLAKE2b is 59.05% faster than BLAKE-512, and BLAKE2s is 31.40% faster than BLAKE-256,
- On Bulldozer, BLAKE2b is 25.96% faster than BLAKE-512, and BLAKE2s is 35.99% faster than BLAKE-256.

Due to the lack of native rotation instructions on SIMD registers, the speedup of BLAKE2b is greater on the Intel processors, which benefit not only from the round reduction, but also from the easier-to-implement rotations.

On short messages, the speed advantage of the improved padding on BLAKE2 is quite noticeable. On Sandy Bridge, no other cryptographic hash function measured in SUPERCOP[5] (including MD5 and MD4) is faster than BLAKE2s on 64-byte messages, while BLAKE2b is as fast as MD4.

As expected, the parallel versions provide a speed-up of a factor close to the parallelization degree: for example, using our tool `b2sum` on Bulldozer, the file `ubuntu-12.04-beta1-desktop-amd64.iso` is hashed in 1.16s with BLAKE2b, 0.33s with BLAKE2bp (that is, 3.51 times faster), in 1.72s with BLAKE2s, and in 0.27s with BLAKE2sp (that is, 6.37 times faster). Similarly, on Sandy Bridge BLAKE2bp is 3.76 times faster than BLAKE2b (1.58s vs 0.42s) hashing the same file, while BLAKE2sp is 3.68 times faster than BLAKE2s (2.21s vs 0.60s). Enabling hyperthreading (8 virtual cores) increases the latter speedup to 5.66, hashing the file in 0.39s. We expect these speedups to converge to 4 and 8 respectively, as implementations (and CPUs) improve.

[5] http://bench.cr.yp.to/results-hash.html#amd64-hydra7

3.3 Low-end platforms

A typical implementation of BLAKE-256 in **embedded software** stores in RAM at least the chaining value (32 bytes), the message (64 bytes), the constants (64 bytes), the permutation internal state (64 bytes), the counter (8 bytes), and the salt, if used (16 bytes); that is, 232 bytes, and 248 with a salt. BLAKE2s reduces these figures to **168 bytes**—recall that the salt doesn't have to be stored anymore—that is, a gain of respectively 28% and 32%. Similarly, BLAKE2b only requires 336 bytes of RAM, against 464 or 496 for BLAKE-512.

3.4 Hardware

Hardware directly benefit from the 29% and 25% speed-up in sequential mode, due to the round reduction, for any message length. Parallelism is straightforward to implement by replicating the architecture of the sequential hash. BLAKE2 enjoys the same degrees of freedom as BLAKE to implement various space-time tradeoffs (horizontal and vertical folding, pipelining, etc.). In addition, parallel hashing provides **another dimension for trade-offs** in hardware architectures: depending on the system properties (e.g. how many input bits can be read per cycle), one may choose between, for example, BLAKE2sp based on 8 high-latency compact cores, or BLAKE2s based on a single low-latency unrolled core.

4 Security

BLAKE2 aims to provide the highest security level, be it in terms of classical notions as (second) preimage or collision resistance, or of theoretical notions as pseudorandomness (a.k.a. indistinguishability) or indifferentiability.

BLAKE2 builds on the high confidence built by BLAKE in the SHA-3 competition. Although BLAKE2 performs fewer rounds than BLAKE, this does not imply a lower security, as explained below.

4.1 BLAKE legacy

The security of BLAKE2 is closely related to that of the SHA-3 finalist BLAKE, since they rely on a similar core permutation originally used in Bernstein's ChaCha stream cipher [4] (itself a variant of Salsa20 [5], co-winner in the eSTREAM project[6]).

The final SHA-3 report [10, p5] comments that, like Keccak, BLAKE has a "very large security margin", and that the cryptanalysis performed on BLAKE to date "appears to have a great deal of depth, while the cryptanalysis on Keccak has somewhat less depth".

Indeed, since 2009, at least 14 research papers have described cryptanalysis results on reduced versions of BLAKE. The most advanced attacks on the BLAKE as hash function—as opposed to its building blocks—are **preimage attacks on 2.5 rounds** by Ji and Liangyu, with respective complexities 2^{241} and 2^{481} for BLAKE-256 and BLAKE-512 [14]. Most research actually considered reduced versions of the compression function or core permutation of BLAKE, regardless of the constraints imposed by the IV. The most recent results of this type are the following

[6] See http://www.ecrypt.eu.org/stream/.

- A “distinguisher” on 6 rounds of the permutation of BLAKE-256, with complexity 2^{456}, by Dunkelman and Khovratovich [11];
- A “boomerang distinguisher” on 8 rounds of the core permutation of BLAKE-512, with complexity 2^{242}, by Biryukov, Nikolic, and Roy [8] (recent works question the correctness of this result [16]).

The exact attacks as described in research papers may not directly apply to BLAKE2, due to the changes of rotation counts (typically, differential characteristics for BLAKE do not apply to BLAKE2). Nevertheless, we expect attacks on reduced BLAKE with n rounds to adapt to BLAKE2 with n rounds, though with slightly different complexities.

4.2 Implications of BLAKE2 tweaks

We have argued that the reduced number of rounds and the optimized rotations are unlikely to meaningfully reduce the security of BLAKE2, compared to that of BLAKE. We summarize the security implications of other tweaks:

Salt-independent compressions. BLAKE2 salts the hash function in the IV, rather than each compression. This preserves the uniqueness of the hash function for any distinct salt, but facilitates theoretical multicollision attacks relying on offline precomputations (see [7,15]). However, this leaves less “controlled” bits in the initial state of the compression function, which complicates the finding of fixed points.

Many valid IVs. Due to the high number of valid parameter blocks, BLAKE2 admits many valid initial chaining values. For example, if an attacker has an oracle that returns collisions for random chaining values and messages, she is more likely to succeed in attacking the hash function because she has many valid targets, rather than a valid one. However, such a scenario assumes that collisions can be found efficiently, that is, that the hash function is already broken.

Simplified padding. The new padding does not include the message length of the message, unlike BLAKE. However, it is easy to see that the length is indirectly encoded through the counter, and that the padding preserves the unambiguous encoding of the initial padding. That is, the padding simplification does not affect the security of the hash function.

References

[1] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 1/2), 2008.

[2] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.

[3] Jean-Philippe Aumasson, Willi Meier, and Raphael C.-W. Phan. The hash function family LAKE. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 36–53. Springer, 2008.

[4] Daniel J. Bernstein. ChaCha, a variant of Salsa20. http://cr.yp.to/chacha.html.

[5] Daniel J. Bernstein. Snuffle 2005: the Salsa20 encryption function. http://cr.yp.to/snuffle.html.

[6] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. Cryptology ePrint Archive, Report 2009/210, 2009. http://eprint.iacr.org/2009/210.

[7] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. http://eprint.iacr.org/2007/278.

[8] Alex Biryukov, Ivica Nikolic, and Arnab Roy. Boomerang attacks on BLAKE-32. In Antoine Joux, editor, *FSE*, volume 6733 of *LNCS*. Springer, 2011.

[9] Donghoon Chang, Mridul Nandi, and Moti Yung. Indifferentiability of the Hash Algorithm BLAKE. Cryptology ePrint Archive, Report 2011/623, 2011. http://eprint.iacr.org/2011/623.

[10] Shu-jen Chang, Ray Perlner, William E. Burr, Meltem Sönmez Turan, John M. Kelsey, Souradyuti Paul, and Lawrence E. Bassham. Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition. NISTIR 7896, National Institute for Standards and Technology, November 2012.

[11] Orr Dunkelman and Dmitry Khovratovich. Iterative differentials, symmetries, and message modification in BLAKE-256. In *ECRYPT2 Hash Workshop*, 2011.

[12] Thai Duong and Juliano Rizzo. Flickr's API Signature Forgery Vulnerability. http://netifera.com/research/, September 2009.

[13] Eirik Haver and Pål Ruud. Experimenting with SHA-3 candidates in Tahoe-LAFS. Technical report, Norwegian University of Science and Technology, 2010.

[14] Li Ji and Xu Liangyu. Attacks on round-reduced BLAKE. Cryptology ePrint Archive, Report 2009/238, 2009. http://eprint.iacr.org/2009/238.

[15] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*. Springer, 2004.

[16] Gaëtan Leurent. ARXtools: A toolkit for ARX analysis. In *The Third SHA-3 Candidate Conference*, March 2012.

[17] Samuel Neves and Jean-Philippe Aumasson. BLAKE and 256-bit advanced vector extensions. In *The Third SHA-3 Candidate Conference*, March 2012.

[18] Samuel Neves and Jean-Philippe Aumasson. Implementing BLAKE with AVX, AVX2, and XOP. Cryptology ePrint Archive, Report 2012/275, 2012. http://eprint.iacr.org/2012/275.

[19] Daniel Pollack. HSS: A simple file storage system for web applications. In *26th Large Installation System Administration Conference (LISA '12)*, 2012.

[20] Bart Preneel. The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *LNCS*. Springer, 2010.

[21] R. Slipetskyy. Security issues in OpenStack. Master's thesis, Norwegian University of Science and Technology, 2011.

[22] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*. Springer, 2009.

A Specification complements

To make the document self-contained, we complete the specification of BLAKEb and BLAKE2s, describing mechanisms inherited from BLAKE and refering to the new features introduced in §2.

Recall that BLAKE2b works with 64-bit words, and BLAKE2s works with 32-bit words, and that both parse byte streams as word arrays in a little-endian way.

A.1 BLAKE2b

BLAKE2b supports data of any byte length $0 \leq \ell < 2^{128}$. Data is first padded as per §§2.3 to form a sequence of $N = \lceil \ell/128 \rceil$ 16-word blocks $m^0, m^1, \ldots, m^{N-1}$, and then hashed by doing

$$
\begin{array}{l}
h^0 \leftarrow IV \oplus P \\
\textbf{for } i = 0, \ldots, N-1 \\
\qquad h^{i+1} \leftarrow \textbf{compress}(h^i, m^i, \ell^i) \\
\textbf{return } h^N
\end{array}
$$

where ℓ^i denotes the number of data bytes in $m^0, m_1, \ldots, m^i$ (that is, not counting any padding byte), P is the parameter block specified in §§2.8, and IV is (as in BLAKE and SHA-512) the following 64-bit words:

$$
\begin{array}{ll}
IV_0 = \texttt{6a09e667f3bcc908} & IV_1 = \texttt{bb67ae8584caa73b} \\
IV_2 = \texttt{3c6ef372fe94f82b} & IV_3 = \texttt{a54ff53a5f1d36f1} \\
IV_4 = \texttt{510e527fade682d1} & IV_5 = \texttt{9b05688c2b3e6c1f} \\
IV_6 = \texttt{1f83d9abfb41bd6b} & IV_7 = \texttt{5be0cd19137e2179}
\end{array}
$$

The compression function **compress** takes as input

- a 64-byte chain value $h = h_0, \ldots, h_7$
- a 128-byte message block $m = m_0, \ldots, m_{15}$
- a counter $t = t_0, t_1$, and finalization flags f_0, f_1

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Table 5: Permutations of $\{0, \ldots, 15\}$ used by the BLAKE2 functions.

First, **compress** initializes a 16-word internal state $v_0, \ldots, v_{15}$ as per §§2.4, that is

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 \oplus IV_4 & t_1 \oplus IV_5 & f_0 \oplus IV_6 & f_1 \oplus IV_7 \end{pmatrix}$$

where f_0 and f_1 are the finalization flags defined in §§2.3.

The internal state v is then transformed through a sequence of 12 rounds, where a round does

$$\begin{array}{llll} G_0(v_0, v_4, v_8, v_{12}) & G_1(v_1, v_5, v_9, v_{13}) & G_2(v_2, v_6, v_{10}, v_{14}) & G_3(v_3, v_7, v_{11}, v_{15}) \\ G_4(v_0, v_5, v_{10}, v_{15}) & G_5(v_1, v_6, v_{11}, v_{12}) & G_6(v_2, v_7, v_8, v_{13}) & G_7(v_3, v_4, v_9, v_{14}) \end{array}$$

That is, a round applies a G function to each of the columns in parallel, and then to all of the diagonals in parallel. The G function of BLAKE2b is defined in §§2.4, and uses the constants in Table 5.

After the 12 rounds, the new chain value $h'_0, \ldots, h'_7$ is defined as

$$\begin{aligned} h'_0 &\leftarrow h_0 \oplus v_0 \oplus v_8 \\ h'_1 &\leftarrow h_1 \oplus v_1 \oplus v_9 \\ h'_2 &\leftarrow h_2 \oplus v_2 \oplus v_{10} \\ h'_3 &\leftarrow h_3 \oplus v_3 \oplus v_{11} \\ h'_4 &\leftarrow h_4 \oplus v_4 \oplus v_{12} \\ h'_5 &\leftarrow h_5 \oplus v_5 \oplus v_{13} \\ h'_6 &\leftarrow h_6 \oplus v_6 \oplus v_{14} \\ h'_7 &\leftarrow h_7 \oplus v_7 \oplus v_{15} \end{aligned}$$

Note the absence of the salt, compared to BLAKE.

A.2 BLAKE2s

BLAKE2s supports data of any byte length $0 \leq \ell < 2^{64}$. It works similarly to BLAKE2b, but on 32-bit words instead of 64-bit words (the byte length of a chaining value, a message block, a counter or finalization flag are thus divided by two).

BLAKE2s uses the following IV:

$$\begin{aligned} IV_0 &= \texttt{6a09e667} & IV_1 &= \texttt{bb67ae85} \\ IV_2 &= \texttt{3c6ef372} & IV_3 &= \texttt{a54ff53a} \\ IV_4 &= \texttt{510e527f} & IV_5 &= \texttt{9b05688c} \\ IV_6 &= \texttt{1f83d9ab} & IV_7 &= \texttt{5be0cd19} \end{aligned}$$

BLAKE2s does 10 rounds, and uses the G function defined in §§2.4.