BLAKE3: one function, fast everywhere

November 28, 2019 — REQUEST FOR COMMENTS, NOT A FINAL VERSION

https://blake3.io

AUTHORS

Abstract. We present the cryptographic hash function BLAKE3, an improved derivative of BLAKE2 and of its predecessor, the SHA-3 finalist BLAKE. BLAKE3 supports an unbounded degree of parallelism, using an internal binary tree structure that scales up to any number of SIMD lanes and CPU cores. On Intel Kaby Lake, peak single-threaded throughput is 3× that of BLAKE2b, 4× that of SHA-2, and 8× that of SHA-3, and it can scale further using multiple threads. On the other end, BLAKE3 is also efficient on smaller architectures and microcontrollers. Throughput on ARM Cortex-M0 is 1.3× that of BLAKE2s or SHA-256, and 3× that of BLAKE2b, SHA-512, or SHA-3. Unlike BLAKE2 and SHA-2, which have incompatible variants better suited for different platforms, BLAKE3 is a single hash function, designed to be consistently fast in software across a wide variety of platforms and use cases.

1 Introduction

Since its announcement in 2012, BLAKE2 [1] has seen widespread adoption, in large part because of its strong performance in software. BLAKE2b and BLAKE2s are included in OpenSSL and in the Python and Go standard libraries. BLAKE2b is also included as the `b2sum` utility in GNU Coreutils, as the `generichash` API in Libsodium, and as the underlying hash function for Argon2, the winner of the Password Hashing Competition in 2015.

A drawback of BLAKE2 has been its large number of incompatible variants. The original BLAKE2 paper described 64-bit BLAKE2b, 32-bit BLAKE2s, the parallel variants BLAKE2bp and BLAKE2sp, and a framework for tree modes. The BLAKE2X paper added extendable output modes. None of these are compatible with each other, and choosing the right one for an application means understanding both the tradeoffs between them and also the state of language and library support. BLAKE2b, the most widely supported, is rarely the fastest on any particular platform. BLAKE2bp and BLAKE2sp, with much higher peak throughput on x86, are sparsely supported and almost never used.

BLAKE3 eliminates this drawback. It is a single hash function with no variants, designed to support all the use cases of BLAKE2, as well as new use cases like streaming verification. Furthermore, it improves on performance, in some cases dramatically. On Intel Kaby Lake, peak throughput on a single core is triple that of BLAKE2b, and BLAKE3 can scale further to any number of cores. On ARM Cortex-M0, throughput is 30% higher than BLAKE2s and again triple that of BLAKE2b.

Internally, BLAKE3 divides its input into 2 KiB chunks and arranges those chunks as the leaves of a binary tree. This tree structure means that there is no limit to the parallelism that BLAKE3 can exploit, given enough input. The direct benefit of this parallelism is very high throughput on platforms with SIMD support, including all modern x86 processors. Another

benefit of hashing chunks in parallel is that the implementation can use SIMD vectors of any width, regardless of the word size of the compression function. That leaves us free to use a compression function that is efficient on smaller architectures, without sacrificing peak throughput on x86-64.

The BLAKE3 compression function is closely based on BLAKE2s. BLAKE3 has the same 128-bit security level and 256-bit default output size. The round function is identical, along with the IV constants and the message schedule. Thus, cryptanalysis of BLAKE2 applies directly to BLAKE3. Based on that analysis and with the benefit of hindsight, we believe that BLAKE2 is overly conservative in its number of rounds, and BLAKE3 reduces the number of rounds from 10 to 7. BLAKE3 also changes the setup and finalization steps of the compression function to support the internal tree structure, more efficient keying, and extendable output.

The biggest changes from BLAKE2 to BLAKE3 are:

- BLAKE3 uses an **internal tree structure**.
- There are **no variants or flavors**.
- The compression function uses **fewer rounds**.
- In lieu of a parameter block, the BLAKE3 API supports **three domain-separated modes**: `hash(input)`, `keyed_hash(key, input)`, and `derive_key(key, context)`. These modes differ only in the internal flag bits they set.
- The space formerly occupied by the parameter block is now used for the optional 256-bit key, so **keying is zero-cost**.
- BLAKE3 has built-in support for **extendable output**. Like BLAKE2X, but unlike SHA-3 or HKDF, extended output is parallelizable and seekable.

2 Specification

2.1 Tree Structure

BLAKE3 splits its input into chunks of up to 2048 bytes and arranges those chunks as the leaves of a binary tree. The last chunk may be shorter, but not empty, unless the entire input is empty. If there is only one chunk, that chunk is the root node and only node of the tree. Otherwise, the chunks are assembled with parent nodes, each parent node having exactly two children. The structure of the tree is determined by two rules:

1. Left subtrees are full. Each left subtree is a complete binary tree, with all its chunks at the same depth, and a number of chunks that is a power of 2.
2. Left subtrees are big. Each left subtree contains a number of chunks greater than or equal to the number of chunks in its sibling right subtree.

For example, trees from 1 to 4 chunks have the structure shown in Figure 1.

The compression function is used to derive a chaining value from each chunk and parent node. The chaining value of the root node, encoded as 32 bytes in little-endian order, is the default-length BLAKE3 hash of the input. BLAKE3 supports input of any byte length $0 \leq \ell < 2^{64}$.

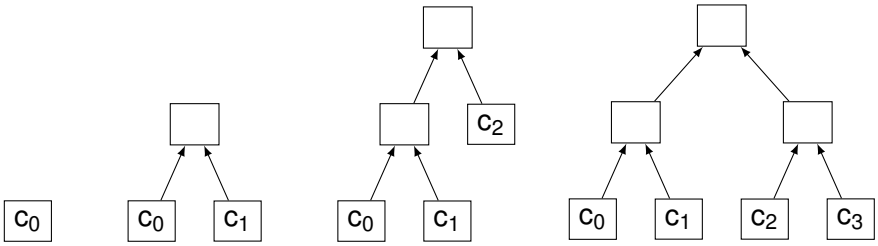

Figure 1: Example tree structures, from 1 to 4 chunks.

2.2 Compression Function

The compression function takes an 8-word chaining value and a 16-word message block, and it returns a new 16-word value. A word is 32 bits. The inputs to the compression function are:

- The input chaining value, $h_0 \dots h_7$.
- The message block, $m_0 \dots m_{15}$.
- A 64-bit offset, $t = t_0, t_1$, with t_0 the lower order word and t_1 the higher order word.
- The number of input bytes in the block, b.
- A set of domain separation bit flags, d.

The compression function initializes its 16-word internal state $v_0 \dots v_{15}$ as follows:

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

The $IV_0 \dots IV_7$ constants are the same as in BLAKE2s, and they are reproduced in Appendix A.

The compression function applies a 7-round keyed permutation $v' = E(m, v)$ to the state $v_0, \dots, v_{15}$, keyed by the message $m_0, \dots, m_{15}$. The keyed permutation here is identical to the one underlying BLAKE2s, and is reproduced in Appendix B.

The output of the compression function $h'_0 \dots h'_{15}$ is defined as:

$$\begin{aligned} h'_0 &\leftarrow v'_0 \oplus v'_8 & h'_8 &\leftarrow v'_8 \oplus h_0 \\ h'_1 &\leftarrow v'_1 \oplus v'_9 & h'_9 &\leftarrow v'_9 \oplus h_1 \\ h'_2 &\leftarrow v'_2 \oplus v'_{10} & h'_{10} &\leftarrow v'_{10} \oplus h_2 \\ h'_3 &\leftarrow v'_3 \oplus v'_{11} & h'_{11} &\leftarrow v'_{11} \oplus h_3 \\ h'_4 &\leftarrow v'_4 \oplus v'_{12} & h'_{12} &\leftarrow v'_{12} \oplus h_4 \\ h'_5 &\leftarrow v'_5 \oplus v'_{13} & h'_{13} &\leftarrow v'_{13} \oplus h_5 \\ h'_6 &\leftarrow v'_6 \oplus v'_{14} & h'_{14} &\leftarrow v'_{14} \oplus h_6 \\ h'_7 &\leftarrow v'_7 \oplus v'_{15} & h'_{15} &\leftarrow v'_{15} \oplus h_7 \,. \end{aligned}$$

Table 1: Admissible values for input d in the BLAKE3 compression function.

Flag name	Value
CHUNK_START	2^0
CHUNK_END	2^1
PARENT	2^2
ROOT	2^3
KEYED_HASH	2^4
DERIVE_KEY	2^5

If we define v_l (resp. v'_l) and v_h (resp. v'_h) as the first and last 8-words of the input (resp. output) of $E(m, v)$ as elements of $\mathbb{F}_2^{256}$, the compression function may be represented as

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v'_l \\ v'_h \end{pmatrix} + \begin{pmatrix} 0 \\ v_l \end{pmatrix} .$$

A common variant of the compression function, used internally to produce 256-bit chaining values, simply truncates the output to its first 256 bits. We defer to analysis of the compression function to §4.

The compression function input d is a bitfield, with each individual flag consisting of a power of 2 value. The value of d is the sum of all the flags defined for a given compression. Their names and values are given in Table 1.

2.3 Modes

BLAKE3 defines three domain-separated modes: `hash`, `keyed_hash`, and `derive_key`. The modes differ from each other in their key words $k_0 \dots k_7$ and in the additional flags they set for every call to the compression function:

- In the `hash` mode, $k_0 \dots k_7$ are the constants $IV_0 \dots IV_7$, and no additional flags are set.
- In the `keyed_hash` mode, $k_0 \dots k_7$ are parsed in little-endian order from the 32-byte key given by the caller, and the `KEYED_HASH` flag is set for every compression.
- In the `derive_key` mode, the key is again given by the caller, and the `DERIVE_KEY` flag is set for every compression.

2.4 Chunk Chaining Values

Processing a chunk is structurally similar to the sequential hashing mode of BLAKE2. Each chunk of up to 2048 bytes is split into blocks of up to 64 bytes. The last block of the last chunk may be shorter, but not empty, unless the entire input is empty. The last block, if necessary, is padded with zeros to be 64 bytes.

Each block is parsed in little-endian order into message words $m_0 \dots m_{15}$ and compressed. The input chaining value $h_0 \dots h_7$ for the first block of each chunk is comprised of the key words $k_0 \dots k_7$. The input chaining value for subsequent blocks in each chunk is the output of the truncated compression function for the previous block.

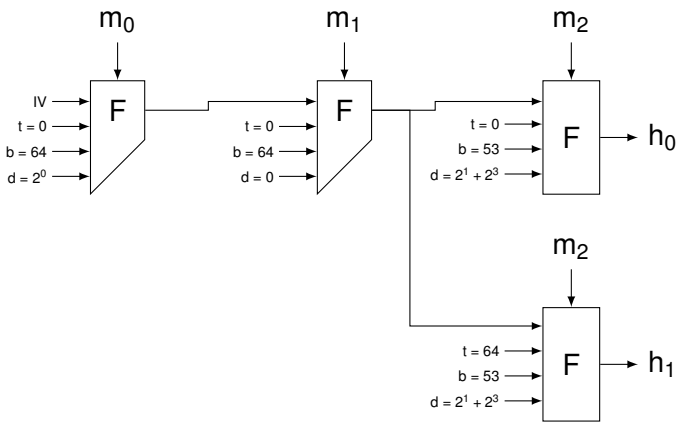

Figure 2: Example of compression function inputs when hashing a 181-byte input (m_0, m_1, m_2) into a 128-byte output (h_0, h_1). Trapezia indicate that the compression function output is truncated to 256 bits.

The remaining compression function parameters are handled as follows (see also Figure 2 for an example):

- The offset t for each block is the starting byte index of its chunk, i.e., 0 for all blocks in the first chunk, 2048 for all blocks in the second chunk, and so on.
- The block length b is the number of input bytes in each block, i.e., 64 for all blocks except the last block of the last chunk, which may be short.
- The first block of each chunk sets the `CHUNK_START` flag (cf. Table 1), and the last block of each chunk sets the `CHUNK_END` flag. If a chunk contains only one block, that block sets both `CHUNK_START` and `CHUNK_END`. If a chunk is the root of its tree, the last block of that chunk also sets the `ROOT` flag. The output of the compression function for the last block in a chunk is the chaining value of that chunk.

2.5 Parent Node Chaining Values

Each parent node has exactly two children, each either a chunk or another parent node. The chaining value of each parent node is given by a single call to the compression function. The input chaining value $h_0 \dots h_7$ is the key words $k_0 \dots k_7$. The message words $m_0 \dots m_{15}$ are the concatenated chaining values of the two children, first the left then the right. The offset t for parent nodes is always 0. The number of bytes b for parent nodes is always 64. Parent nodes set the `PARENT` flag. If a parent node is the root of the tree, it also sets the `ROOT` flag. The first 8 words of the output of the compression function is the chaining value of the parent node.

2.6 Extendable Output

BLAKE3 can produce outputs of any byte length up to 2^{64} bytes. This is done by repeating the root compression—that is, the very last call to the compression function, which sets

the ROOT flag—with incrementing values of the offset t. The results of these repeated root compressions are then concatenated to form the output.

When building the output, BLAKE3 uses the full output of the compression function (cf. §2.2). Each 16-word output is encoded as 64 bytes in little-endian order.

Observe that for the first output, the root compression always uses offset t = 0. That is either because it is the last block of the only chunk, which has a chunk offset of 0, or because it is a parent node. Then as long as more bytes are needed, t is incremented by 64, and the root compression is repeated on the same inputs. If the target output length is not a multiple of 64, the final compression output is truncated.

Because the repeated root compressions differ only in the value of t, the implementation can execute any number of them in parallel. The caller can also adjust t to seek to any point in the output stream.

Note that in contrast to BLAKE2 and BLAKE2X, BLAKE3 does not domain separate outputs of different lengths. Shorter outputs are prefixes of longer ones.

3 Performance

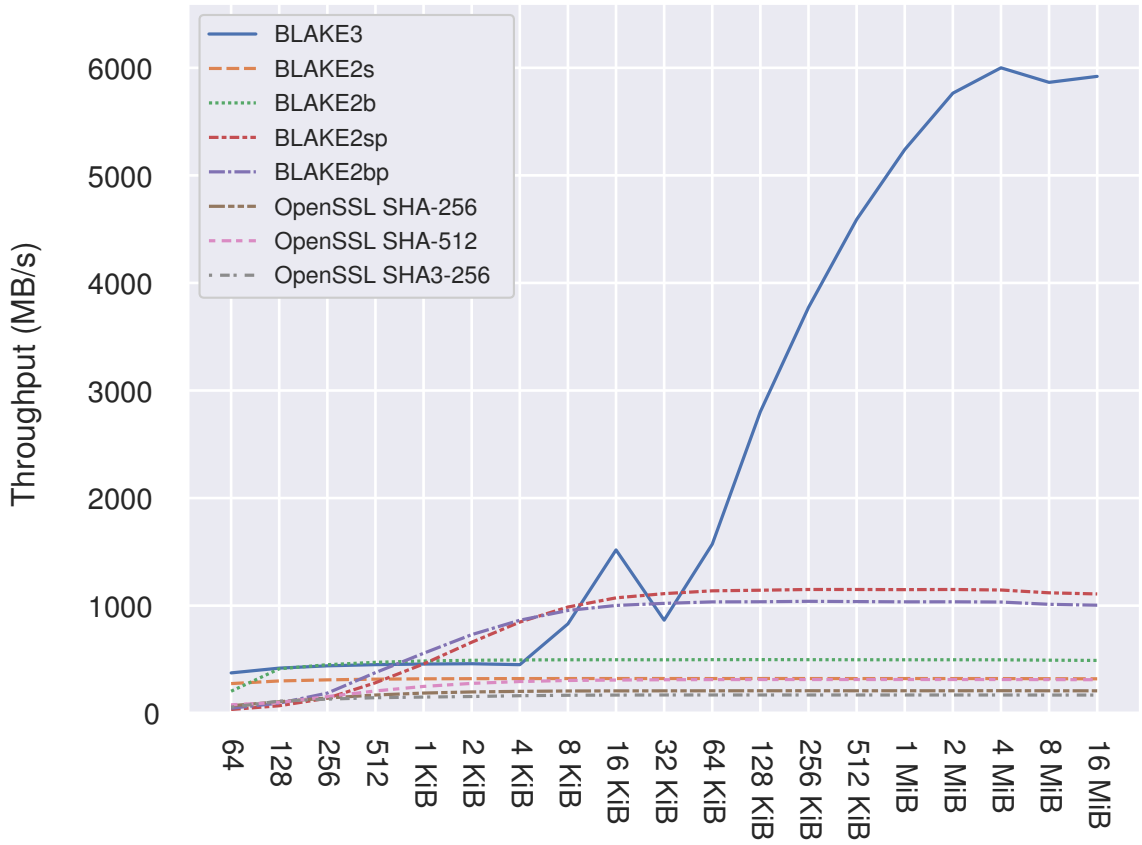

Figure 3: Throughput for a multi-threaded implementation of BLAKE3 at various input lengths on an Intel Kaby Lake processor.

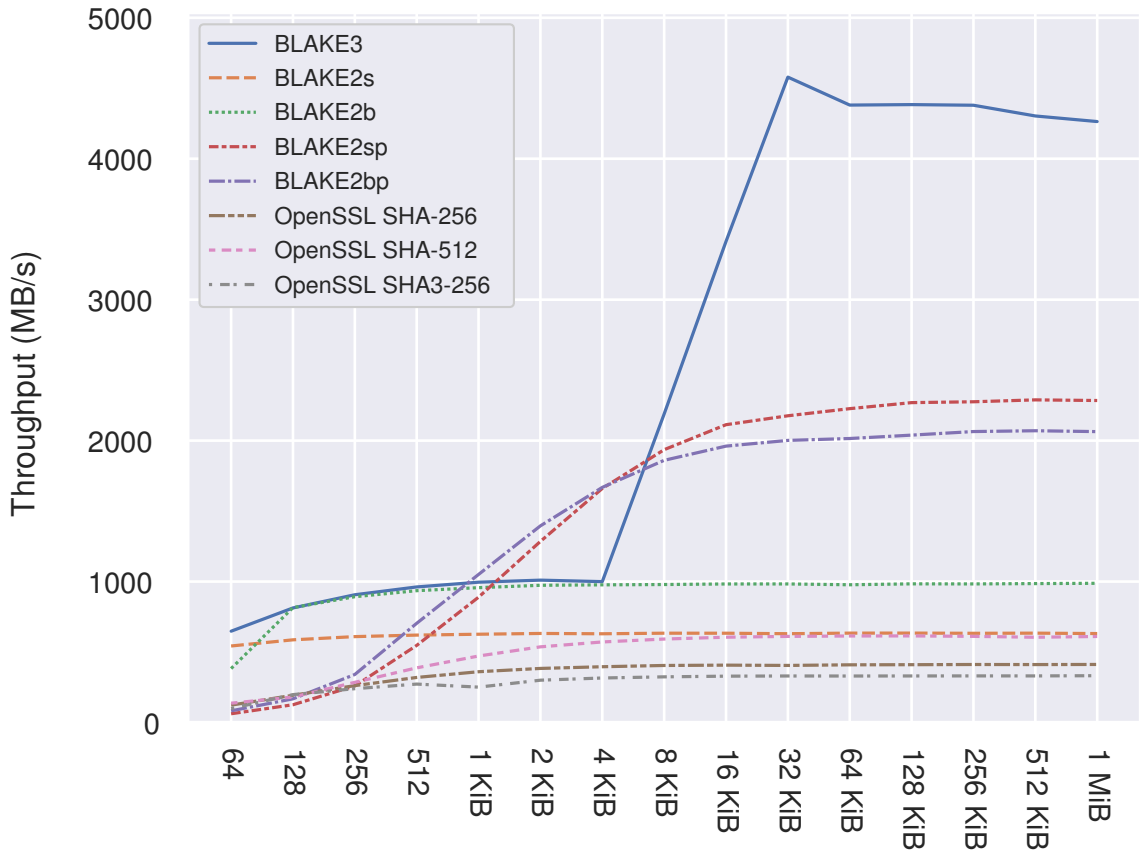

Figure 4: Throughput for a single-threaded implementation of BLAKE3 at various input lengths on an Intel Skylake-SP processor.

Several factors contribute to the performance of BLAKE3, depending on the platform and the size of the input:

- The tree structure allows an implementation to compress multiple chunks in parallel using SIMD (cf. §6.1). With enough input, this can occupy SIMD vectors of any width.
- The tree structure allows an implementation to use multiple threads. With enough input, this can occupy any number of cores.
- The compression function uses fewer rounds than in BLAKE2.
- The compression function performs well on smaller architectures.

Figure 3 shows the throughput of a multi-threaded implementation of BLAKE3 on a typical laptop computer. This benchmark ran on an Intel Kaby Lake i5-8250U processor with 4 physical cores, supporting 256-bit AVX2 vector instructions, and with Turbo Boost disabled. In this setting, the implementation remains single-threaded for inputs up to 16 KiB. Parallel compression using SIMD begins with 8 KiB inputs using 128-bit vectors, and with 16 KiB inputs using 256-bit vectors. The implementation begins multi-threading with 32 KiB inputs, in this case causing a performance drop. (An implementation tuned for this specific platform

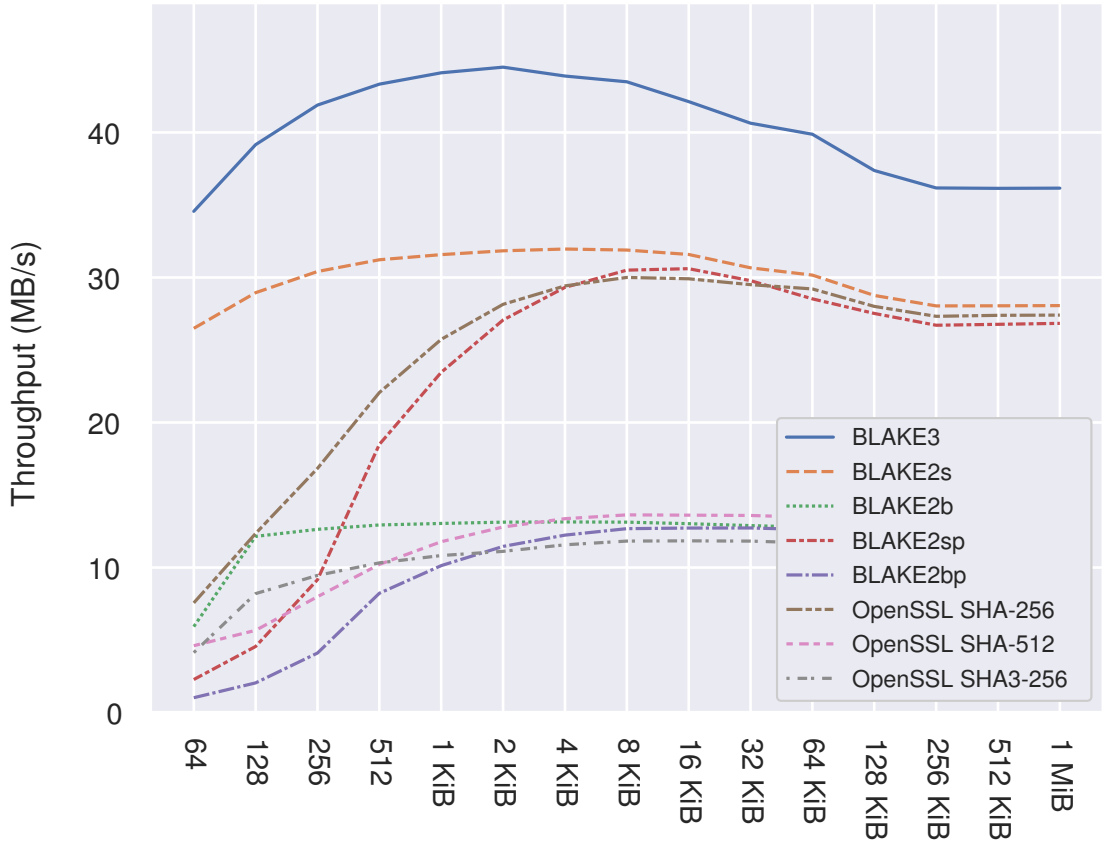

Figure 5: Throughput at various input lengths on an ARM Cortex-M0 processor.

would forgo multi-threading for inputs shorter than e.g. 128 KiB.) Peak throughput occurs at 4 MiB of input, at which point BLAKE3 is 5× faster than BLAKE2bp and BLAKE2sp and an order of magnitude faster than BLAKE2b, BLAKE2s, SHA-2, and SHA-3.

Figure 4 shows the throughput of a single-threaded implementation of BLAKE3 on modern server hardware. This benchmark ran on an AWS t3.nano instance with an Intel Skylake-SP 8175M processor, supporting AVX-512 vector instructions. Here, BLAKE3 is the only algorithm able to take advantage of 512-bit vector arithmetic. (Note that the KangarooTwelve algorithm, not included in this benchmark, can also use 512-bit vectors.) The fixed tree structures of BLAKE2bp and BLAKE2sp limit those algorithms to 256-bit vectors.

Figure 5 shows the throughput of BLAKE3 on a smaller embedded platform. This benchmark ran on an ARM Cortex-M0 processor, without multiple cores or SIMD support. Here, BLAKE2b, SHA-512, and SHA-3 perform relatively poorly, because their compression functions require 64-bit arithmetic. BLAKE3 and BLAKE2s have similar performance profiles here, and the BLAKE3 compression function uses fewer rounds.

Figures 3, 4, and 5 also highlight that BLAKE3 performs well for short inputs. The fixed tree structures of BLAKE2bp and BLAKE2sp are costly when the input is short, because they always compress a parent node and a fixed number of leaves. The advantage of their fixed tree structures, however, comes at medium input lengths around 1-4 KiB, where BLAKE3 does not yet have enough chunks to operate in parallel. This is the regime where BLAKE2bp

and BLAKE2sp pull ahead of BLAKE3 in figures 3 and 4.

4 Security

It's safe :)

5 Memory Requirement

BLAKE3 has a larger memory requirement than BLAKE2. An incremental implementation needs to store a 32-byte chaining value for every level of the tree below the root. The maximum input size is $2^{64} - 1$ bytes, and the chunk size is 2^{11} bytes, giving a maximum tree depth of $64 - 11 = 53$. The stack of chaining values thus requires $53 \cdot 32 = 1696$ bytes. The incremental state of the current chunk also requires at least 104 bytes for the chaining value, the message block, and the chunk offset. The total size of the reference implementation is 1848 bytes.

For comparison, BLAKE2s has a memory footprint similar to the BLAKE3 chunk state alone, at least 104 bytes. BLAKE2b has twice the chaining value size and block size, requiring at least 200 bytes. And the parallel modes BLAKE2bp and BLAKE2sp both require at least 776 bytes.

Space-constrained implementations of BLAKE3 can save space by restricting the maximum input size. For example, the maximum size of an IPv6 "jumbogram" is $2^{32} - 1$ bytes, or just under 4 GiB. At this size, the tree depth is 21 and the chaining value stack is $21 \cdot 32 = 672$ bytes. For another example, the maximum size of a TLS record is 2^{14} bytes, or exactly 16 KiB. At this size, the tree depth is 3 and the chaining value stack is $3 \cdot 32 = 96$ bytes.

6 Implementation

6.1 SIMD

There are two approaches to using SIMD in a BLAKE3 implementation, and both are important for high performance at different input lengths. The first approach is to use 128-bit vectors to represent the 4-word rows of the state matrix. The second approach is to use vectors of any size to represent words in multiple states, which are compressed in parallel.

The first approach is similar to how SIMD is used in BLAKE2b or BLAKE2s, and it is applicable to inputs of any length, particularly short inputs where the second approach does not apply. The state $v_0 \dots v_{15}$ is arranged into four 128-bit vectors. The first vector contains the state words $v_0 \dots v_3$, the second vector contains the state words $v_4 \dots v_7$, and so on. Implementing the G function (Appendix B) with vector instructions thus mixes all four columns of the state matrix in parallel. A diagonalization step then rotates the words within each row so that each diagonal now lies along a column, and the vectorized G function is repeated to mix diagonals. Finally the state is undiagonalized, to prepare for the column step of the following round.

The second approach is similar to how SIMD is used in BLAKE2bp or BLAKE2sp. In this approach, multiple chunks are compressed in parallel, and each vector contains one word from the state matrix of each chunk. That is, the first vector contains the v_0 word from

each state, the second vector contains the v_1 word from each state, and so on, using 16 vectors in total. The width of the vectors determines the number of chunks, so for example 128-bit vectors compress 4 chunks in parallel, and 256-bit vectors compress 8 chunks in parallel. Here the G function operates on one column or diagonal at a time, but across all of the states, and no diagonalization step is required. When enough input is available, this approach is much more efficient than the first approach. It also scales to wider instruction sets like AVX2 and AVX-512.

6.2 Incremental Hashing

How to merge subtrees, using that trick with counting 1 bits. Include a pretty diagram.

7 Applications

7.1 Message Authentication Codes

Like BLAKE2, BLAKE3 has explicit support for a keyed mode, `keyed_hash`. This removes the need for a separate construction like HMAC. The `keyed_hash` mode is also more efficient than keyed BLAKE2 or HMAC for short messages. BLAKE2 requires an extra compression for the key block, and HMAC requires three extra compressions. The `keyed_hash` mode in BLAKE3 does not require any extra compressions.

7.2 Key Derivation

BLAKE3 has explicit support for a key derivation mode, `derive_key`. In this mode the input bytes should be a hardcoded, globally unique context string. A good default format for such strings is `"[application] [date] [purpose]"`, e.g., `"example.com 2019-12-25 16:18:03 session tokens v1"`. If the key material is not already 256 bits, it can be converted to 256 bits using the `hash` mode. The `derive_key` mode is intended to replace the BLAKE2 personalization parameter for its most common use cases.

Key derivation can encourage better security than personalization, by cryptographically isolating different components of an application from one another. This limits the damage that one component can cause by accidentally leaking its key. When an application needs to use one secret in multiple algorithms or contexts, the best practice is to derive a separate key for each use case, such that the original secret is only used with the key derivation function. However, because the BLAKE3 modes are domain-separated, it is also possible to use the same key with `keyed_hash` and with `derive_key`. This can be necessary for backwards compatibility when an application adds a new use case for an existing key, if the original use case did not include a context-specific key derivation step.

Like `keyed_hash`, `derive_key` does not require any extra compressions. For context strings up to 64 bytes and derived key lengths up to 64 bytes, `derive_key` is a single compression. For comparison, HKDF generally requires eight compressions.

7.3 Streaming Verification

Bao, basically.

7.4 Incremental Update

Giant disk images.

8 Rationales

References

[1] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In Jr. et al. [4], pages 119–135. doi:10.1007/978-3-642-38980-1_8.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. *Int. J. Inf. Sec.*, 13(4):335–353, 2014. doi:10.1007/s10207-013-0220-y.

[3] Joan Daemen, Bart Mennink, and Gilles Van Assche. Sound hashing modes of arbitrary functions, permutations, and block ciphers. *IACR Trans. Symmetric Cryptol.*, 2018(4):197–228, 2018. doi:10.13154/tosc.v2018.i4.197-228.

[4] Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors. *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*. Springer, 2013. doi:10.1007/978-3-642-38980-1.

[5] Atul Luykx, Bart Mennink, and Samuel Neves. Security analysis of BLAKE2's modes of operation. *IACR Trans. Symmetric Cryptol.*, 2016(1):158–176, 2016. doi:10.13154/tosc.v2016.i1.158-176.

[6] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). *RFC*, 7693:1–30, 2015. doi:10.17487/RFC7693.

[These references are placeholders. TODO.]

Appendix A IV Constants

The constants $IV_0 \ldots IV_7$ used by the compression function are the same as in BLAKE2s. They are:

IV_0 = 0x6a09e667 IV_1 = 0xbb67ae85
IV_2 = 0x3c6ef372 IV_3 = 0xa54ff53a
IV_4 = 0x510e527f IV_5 = 0x9b05688c
IV_6 = 0x1f83d9ab IV_7 = 0x5be0cd19

Appendix B Round Function

The compression function transforms the internal state $v_0 \dots v_{15}$ through a sequence of 7 rounds. The round function is the same as in BLAKE2s. A round does:

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

That is, a round applies a G function to each of the columns in parallel, and then to each of the diagonals in parallel. $G_i(a, b, c, d)$ is defined as follows. $\oplus$ denotes bitwise exclusive-or, $\ggg$ denotes bitwise right rotation, and $m_{\sigma_r(x)}$ is the message word whose index is the x^{th} entry in the message schedule for round r:

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

The message schedules σ_r are:

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