BLAKE3

one function, fast everywhere

Jack O'Connor (@oconnor663)
Jean-Philippe Aumasson (@veorq)
Samuel Neves (@sevenps)
Zooko Wilcox-O'Hearn (@zooko)

https://blake3.io

We present BLAKE3, an evolution of the hash BLAKE2 that is faster, simpler to use, and better suited to applications' needs. BLAKE3 supports an unbounded degree of parallelism, using a 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 scales further using multiple threads. BLAKE3 is also efficient on smaller architectures and microcontrollers: throughput on the 32-bit ARM1176 core 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 across a wide variety of software platforms and use cases.

Version 20191218100400.

Contents

1 Introduction

Since its announcement in 2012, BLAKE2 [5] has seen widespread adoption, in large part because of its superior 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 [8], the winner of the Password Hashing Competition in 2015.

The biggest changes from BLAKE2 to BLAKE3 are:

- An **internal tree structure**.
- A compression function with **fewer rounds**.
- A **single hash function**, with no variants or flavors.
- In lieu of a parameter block, an API supporting **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.

BLAKE3 thus eliminates the main drawback of BLAKE2, namely its number of incompatible variants; the original BLAKE2 paper described 64-bit BLAKE2b, 32-bit BLAKE2s, the parallel variants BLAKE2bp and BLAKE2sp, a framework for tree modes, and the BLAKE2X paper later 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 not the fastest on most platforms. BLAKE2bp and BLAKE2sp, with much higher peak throughput, are sparsely supported and rarely adopted. BLAKE3 instead is a single hash function, with no variants, designed to support all the use cases of BLAKE2, as well as new use cases like verified streaming (see §6.4).

BLAKE3 is also dramatically faster. For example, on an Intel Kaby Lake processor peak throughput on a single core is triple that of BLAKE2b, and BLAKE3 can scale further to any number of cores. On an ARM1176, throughput is $1.3\times$ that of BLAKE2s and again triple that of BLAKE2b. To achieve such speed, BLAKE3 splits its input into 1 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 [1, 2]. 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 that of 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 (see [3] for detailed rationale). BLAKE3 also changes the setup and finalization steps of the compression function to support the internal tree structure, more efficient keying, and extendable output.

2 Specification

2.1 Tree Structure

BLAKE3 splits its input into chunks of up to 1024 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.

In other words, given a message of n $>$ 1024 bytes, the left subtree consists of the first

$$2^{10+\lfloor \log_2(\lfloor \frac{n-1}{1024} \rfloor)\rfloor}$$

bytes, and the right subtree consists of the remainder.

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

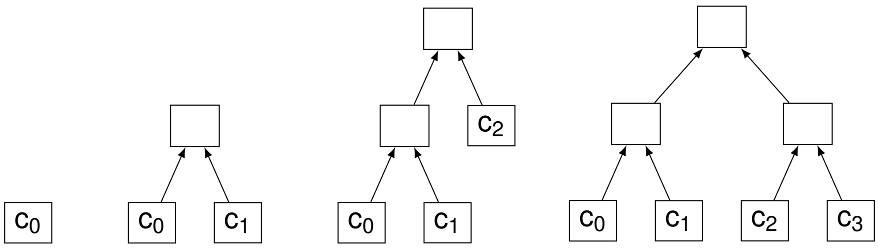

Figure 1: Example tree structures, from 1 to 4 chunks.

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 $0 \leq \ell < 2^{64}$.

Lots of references to “default length”. Given that the output is not different between different sizes—in theory, BLAKE3’s output is an infinite-length string—it’s much less of an issue than in BLAKE2.

2.2 Compression Function

The compression function takes an 8-word chaining value, a 16-word message block, and a 4-word parameter, 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 counter, $t = t_0, t_1$, 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 that of BLAKE2s, and is reproduced in Appendix B.

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

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

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 the affine transformation

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

The output of the compression function is often truncated to produce 256-bit chaining values.

The compression function input d is a bitfield, with each individual flag consisting of a power of 2. 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.

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

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:

- `hash`: $k_0 \dots k_7$ are the constants $IV_0 \dots IV_7$, and no additional flags are set.
- `keyed_hash`: $k_0 \dots k_7$ 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.
- `derive_key`: 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 1024 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.

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

- The counter t for each block is the chunk index, i.e., 0 for all blocks in the first chunk, 1 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 full 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 truncated compression function for the last block in a chunk is the chaining value of that chunk.

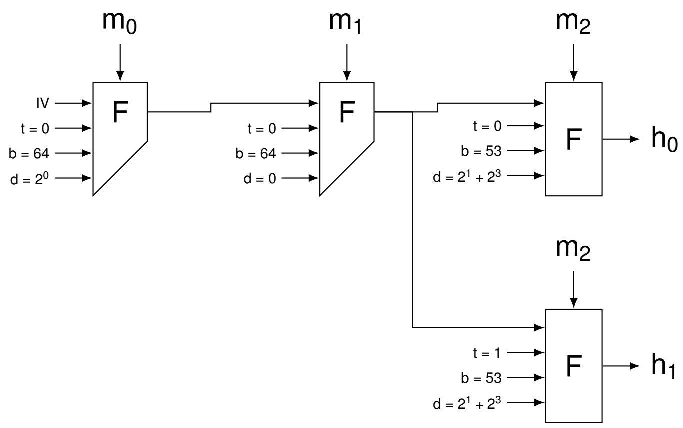

Figure 2: Example of compression function inputs when hashing a 181-byte input (m_0, m_1, m_2) into a 128-byte output (h_0, h_1). Trapezia indicate that the compression function output is truncated to 256 bits.

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 $h_0 \ldots h_7$ is the key words $k_0 \ldots k_7$. The message words $m_0 \ldots m_7$ are the chaining value of the left child, and the message words $m_8 \ldots m_{15}$ are the chaining value of the right child. The counter 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 output of the truncated 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 counter 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 based on §2.4 and §2.5 above, the first root compression always uses the counter value $t = 0$. That is either because it is the last block of the only chunk, which has a chunk index of 0, or because it is a parent node. After the first root compression, as long as more output bytes are needed, t is incremented by 1, and the root compression is repeated on otherwise 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

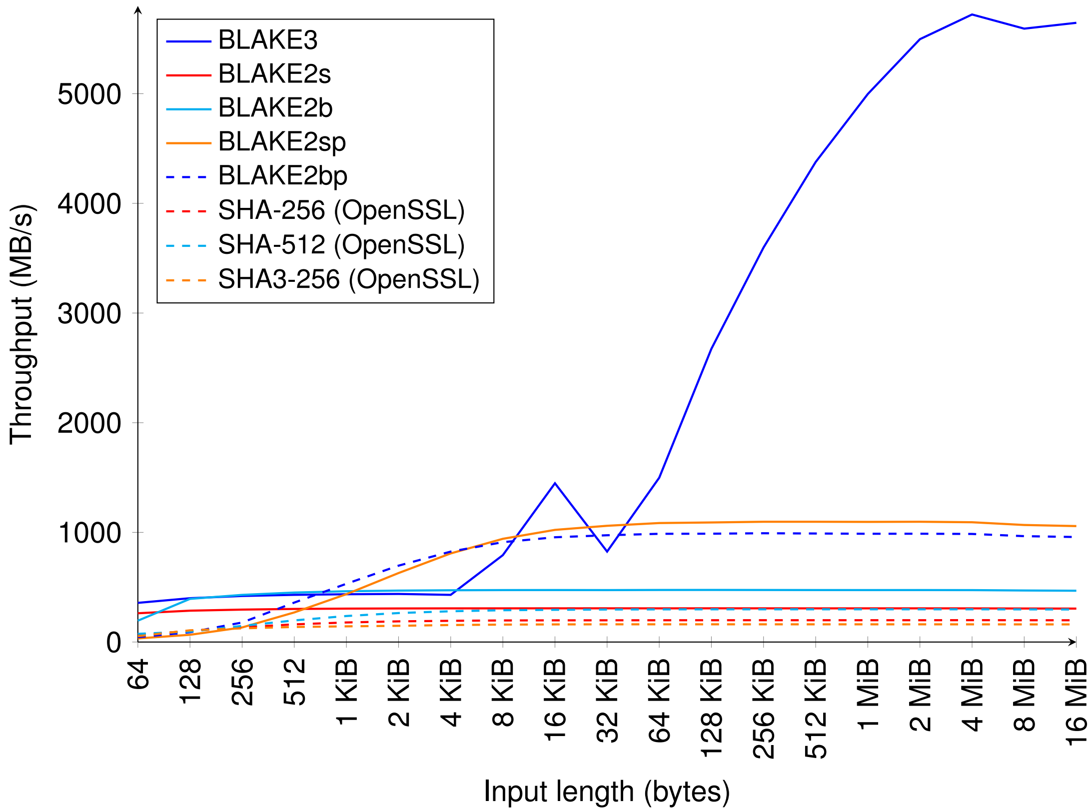

Figure 3: Throughput for a multi-threaded implementation of BLAKE3 at various input lengths on an Intel Kaby Lake processor.

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. §5.3). 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

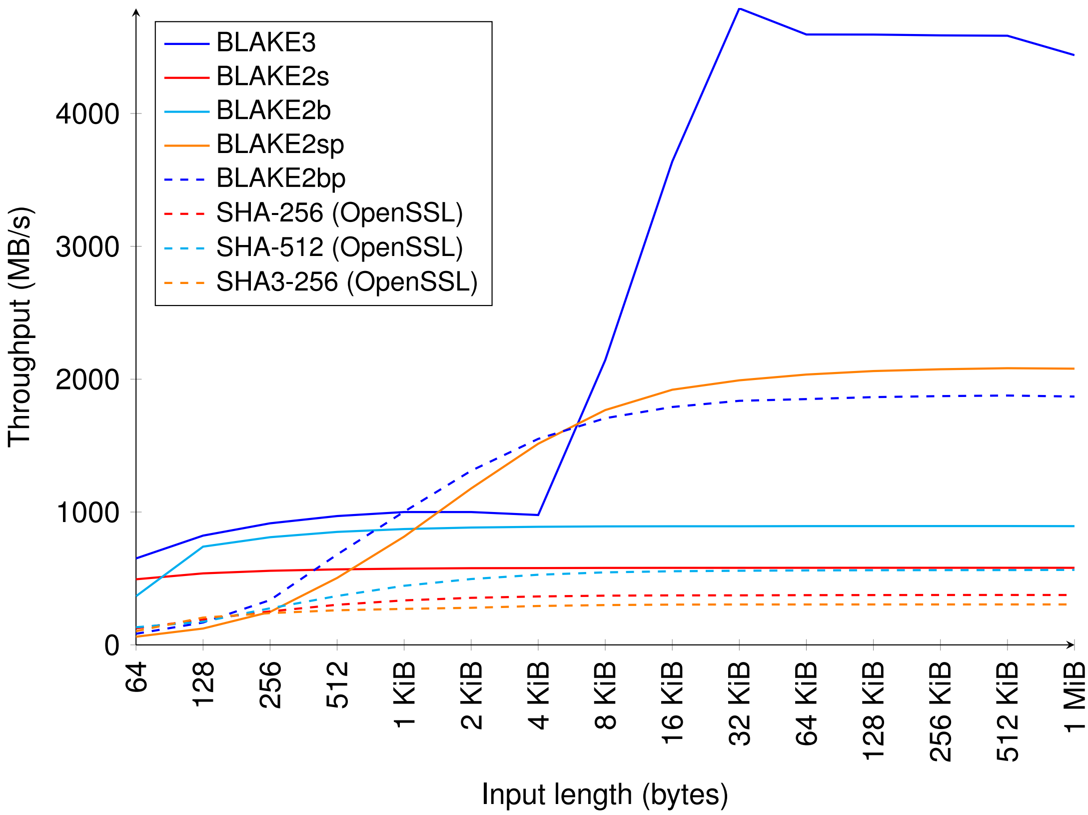

Figure 4: Throughput for a single-threaded implementation of BLAKE3 at various input lengths on an Intel Skylake-SP processor.

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 c5.metal instance with an Intel Skylake-SP 8175M processor, supporting AVX-512 vector instructions, and with Turbo Boost disabled. 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 a Raspberry Pi Zero with a 32-bit ARM1176 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

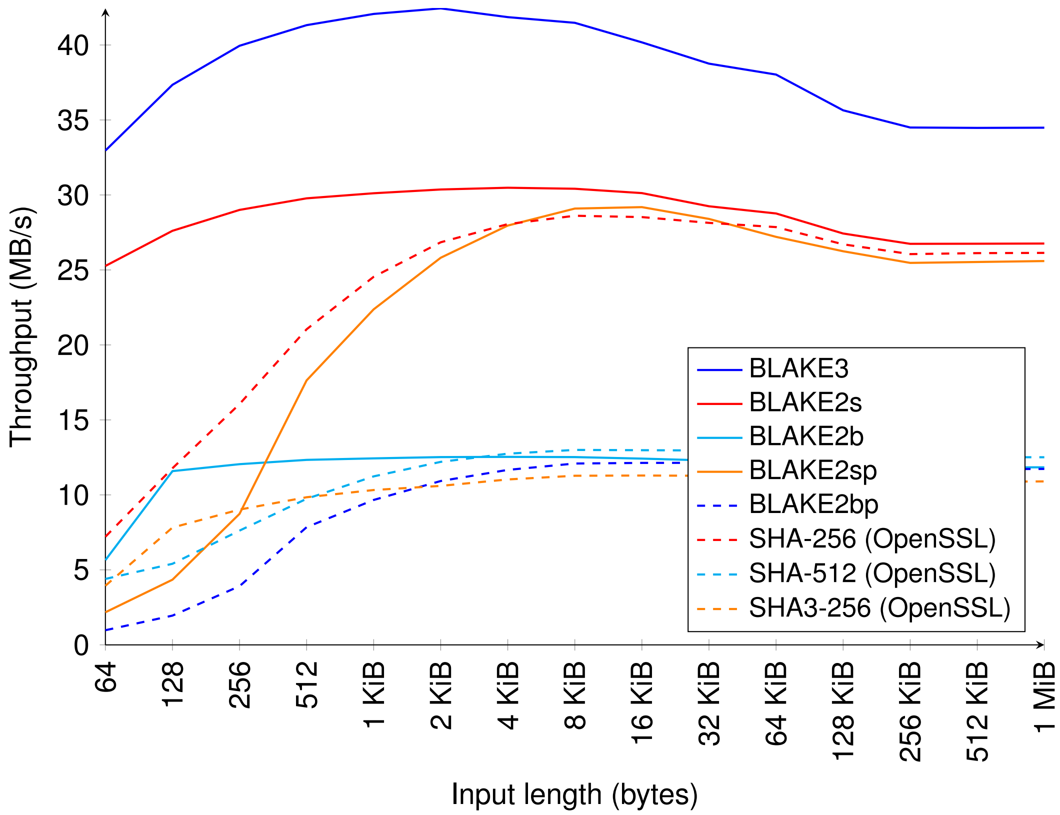

Figure 5: Throughput at various input lengths on an ARM1176 processor.

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

4 Security

TODO

4.1 Security Goals

BLAKE3 targets 128-bit security for all of its security goals. That is, 128-bit security against (second-)preimage, collision, or differentiability attacks.

The key length is nevertheless 256 bits, which is useful against multi-target or possible certificational attacks.

Users who need 256-bit security can remain using BLAKE2.

On why 128-bit security is enough [3].

4.2 Mode of Operation

Indifferentiability / tree soundness stuff here [7, 10, 16].

Explain sufficient conditions satisfied..

Table 2: Best differential trail probabilities for increasing round numbers of the compression functions of BLAKE-256, BLAKE2s, and BLAKE3, respecting their constraints on inputs to the keyed permutation. Probabilities marked with * indicate symmetric differences were sought exclusively.

Function	0.5	1	1.5	2	2.5	3	3.5
BLAKE-256 c.f.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-6}	2^{-7}	2^{-38}
BLAKE{2s,3} c.f.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-32}	$2^{-88} \leq p < 2^{-48}$	-
BLAKE{-256,2s,3}	2^{-0}	2^{-1}	2^{-32}	$\geq 2^{-190*}$	-	-	-
BLAKE-256 c.f. (CV only)	2^{-0}	2^{-2}	2^{-12}	2^{-39}	-	-	-
BLAKE{2s,3} c.f. (CV only)	2^{-0}	2^{-7}	2^{-32}	$\geq 2^{-161*}$	-	-	-

4.3 Cryptanalysis

The rather aggressive round reduction from 10 to 7 rounds is based on existent cryptanalysis, along with novel research.

See [3].

- Cryptanalysis of BLAKE [4, 6, 9, 11, 15, 17, 18]
- Cryptanalysis of BLAKE2 [12–14]
- Table 2 shows that the initialization of BLAKE2/3, by restricting inputs, makes things harder for the attacker, with probabilities getting very low very quickly. As also observed in [13, §7].
- The boomerang attacks of, e.g., [6, 9, 14] exploit the high probabilities on the first few rounds to connect two trails and get through a few more rounds. This is a usual trick to get around the very quick lowering of probabilities in ARX constructs, along with differential-linear attacks. But such attacks do not present a threat to BLAKE3, considering its 128-bit security target, and input restrictions on the compression function input (i.e., the IV).

5 Implementation

5.1 Incremental Hashing

An incremental implementation of BLAKE3 needs two major components: the state of the current chunk and a stack of subtree chaining values (the “CV stack”). The chunk state is structurally similar to an incremental implementation of the sequential mode of BLAKE2. Simplifying the management of the CV stack is especially important for a concise and correct implementation of BLAKE3. This section is best read together with the code of the reference implementation.

5.1.1 Chunk State

The chunk state contains the 32-byte CV of the previous block and a 64-byte input buffer for the next block, and typically also the compression function parameters t and d. Input bytes from the caller are copied into the buffer until it is full. Then the buffer is compressed together with the previous CV, using the truncated compression function. The output CV overwrites the previous CV, and the buffer is cleared. An important detail here is that the last block of a chunk is compressed differently, setting the `CHUNK_END` flag and possibly the `ROOT` flag. In an incremental setting, any block could potentially be the last, until the caller has supplied enough input to place at least 1 byte in the block after. For that reason, the chunk state waits to compress the next block until both the buffer is full and the caller has supplied more input. Note that the CV stack takes the same approach immediately below: chunk CVs are added to the stack only after the caller supplies at least 1 byte for the following chunk.

5.1.2 Chaining Value Stack

To help picture the role of the CV stack, Figure 6 shows a growing tree as chunk CVs are added incrementally. As just discussed above, chunk CVs are added only after the caller has supplied at least 1 byte for the following chunk, so we know that these chunks and parent nodes are not the root of the tree, and we do not need to worry about the `ROOT` flag yet. When the first chunk CV is added, c_0, it is alone. When the second chunk CV is added, c_1, it completes a subtree of size 2, and we can compute the first parent CV, p_0. The third chunk CV, c_2, does not complete any subtrees. The fourth chunk CV, c_3, completes two subtrees, one of size 2 and one of size 4. First it joins with c_2 giving a new parent CV, p_1. Then p_1 joins with p_0, giving p_2.

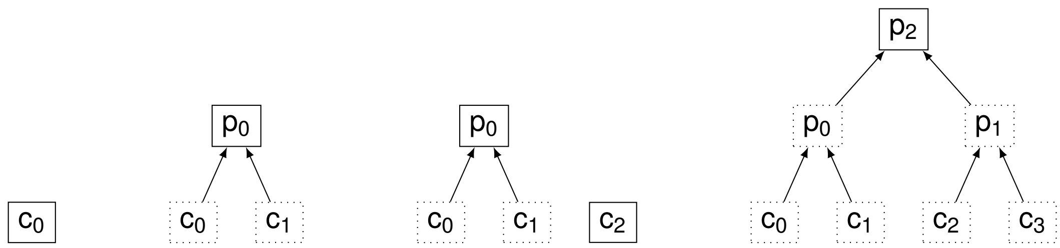

Figure 6: An incomplete tree growing incrementally from 1 to 4 chunks. Dotted boxes represent CVs that no longer need to be stored.

Note that once a subtree is complete, none of its child CVs need to be stored any longer. We store the CVs that we still need in the CV stack. We can look through Figure 6 again, this time focusing on the solid and dotted boxes, with dotted boxes representing the CVs that no longer need to be stored. When c_0 is alone, it is the only entry in the CV stack. When c_1 is added, c_0 is removed from the stack and merged into p_0, and p_0 becomes the only entry. When c_2 is added, both p_0 and c_2 remain in the stack. When c_3 is added, c_2 and p_0 are both removed and merged into p_2, and p_2 is the only entry that remains. In this sense, c_1, c_3, and p_1 are never directly stored in the stack.

Thus the key question for managing the CV stack is, when a new chunk CV is added, how many subtrees does it complete? Which is to say, how many CVs should we pop off the stack and merge with it first?

To answer that question, note the following invariant. At each step, the number of CVs remaining in the stack is the same as the number of 1 bits in the binary representation of the total number of chunks so far. With one chunk in the tree (0b001), the stack contains one CV. With two chunks in the tree (0b010), the stack still contains one CV. With three chunks in the tree (0b011), the stack now contains two CVs. And with four chunks in the tree (0b100), the stack once again contains one CV. When we eventually reach seven chunks (0b111), the stack will contain three CVs.

To understand why this works, decompose the number of chunks hashed so far in its base 2 form: $b_0 \cdot 2^0 + b_1 \cdot 2^1 + \cdots + b_{63} \cdot 2^{63}$. Each $b_i \neq 0$ corresponds to a full power-of-2 subtree that has been completed and whose CV can be stored without its children. (Saying the same as above, but powers of 2 make it seem less magical than bits.)

This invariant leads to an algorithm for adding a chunk CV to the stack: Continue popping CVs off the stack to form new parent nodes, as long as the number of entries in the stack is greater than or equal to the number of 1 bits in the new total number of chunks. Listing 1 shows this algorithm as it appears in the Rust reference implementation.

```rust
fn push_chunk_chaining_value(&mut self, mut cv: [u32; 8], total_chunks: u64) {
    // The new chunk chaining value might complete some subtrees along the
    // right edge of the growing tree. For each completed subtree, pop its
    // left child CV off the stack and compress a new parent CV. After as
    // many parent compressions as possible, push the new CV onto the
    // stack. The final length of the stack will be the count of 1 bits in
    // the total number of chunks so far.
    let final_stack_len = total_chunks.count_ones() as u8;
    while self.cv_stack_len >= final_stack_len {
        cv = parent_output(&self.pop_stack(), &cv, &self.key, self.chunk_state.flags)
            .chaining_value();
    }
    self.push_stack(&cv);
}
```

Listing 1: The algorithm in the Rust reference implementation that manages the chaining value stack when a new chunk CV is added.

Once the caller indicates that the input is complete, the current chunk is finalized, and the new chunk CV is merged with each entry in the CV stack from top to bottom. This happens regardless of the number of chunks so far, reflecting the fact that subtrees along the right edge of the tree may be incomplete. The last parent node compression in this case sets the `ROOT` flag. Or if there are no CVs in the stack, and thus no parent nodes to compress, finalization of the last (and only) chunk sets the `ROOT` flag.

5.2 Multi-threading

Most of the work of computing a BLAKE3 hash is compressing chunks. Each chunk can be compressed independently, and one approach to multi-threading is to farm out individual chunks or groups of chunks to tasks on a thread pool. In this approach, a leader thread owns the chaining value stack (cf. §5.1.2) and awaits a CV from each task in order.

This leader-workers approach has some inefficiencies. Spawning tasks and creating channels usually require heap allocation, which is a performance cost that needs to be amortized over larger groups of chunks. At high degrees of parallelism, managing the CV stack itself can become a bottleneck.

A more efficient approach to multi-threading is based on a recursive tree traversal. The input is split into left and right parts. As per the rules in §2.1, the left part receives the largest power-of-2 number of chunks that leaves at least 1 byte for the right part. Each part then repeats this splitting step recursively, until the parts are chunk-sized, and each chunk is compressed into a CV. On the way back up the callstack, each pair of left and right child CVs is compressed into a parent CV.

This recursive approach fits well into a fork-join concurrency model, like those provided by OpenMP, Cilk, and Rayon (Rust). Each left and right part becomes a separate task, and a work-stealing runtime parallelizes those tasks across however many threads are available. This can work without heap allocation, because the runtime can make a fixed-size stack allocation at each recursive callsite.

The recursive approach is simplest when the entire input is available at once, since no CV stack is needed. In an incremental setting, a hybrid approach is also possible. A large buffer of input can be compressed recursively into a single subtree CV, and that CV can be pushed onto the CV stack using the same algorithm as in §5.1.2. If each incremental input is a fixed power-of-2 number of chunks in size (for example if all input is copied into an internal buffer before compression), the push algorithm works with no modification. If each input is a variable size (for example if input from the caller is compressed directly without copying), the implementation needs to maintain the largest-to-smallest ordering invariant of the CV stack. The size of each subtree being compressed must evenly divide the total number of input bytes received so far, and the implementation might need to break up the caller's input into separate pieces.

5.3 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 $v_0 \ldots v_{15}$ is arranged into four 128-bit vectors. The first vector contains the state words $v_0 \ldots v_3$, the second vector contains the state words $v_4 \ldots v_7$, 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.

For SIMD implementations with multiple 128-bit lanes, such as AVX2 and AVX-512, there is a 3rd approach that simply replicates the first approach without transposing the state /

message. This might be advantageous in some short-message regimes to be determined.

The second approach can be integrated with the CV stack algorithm from §5.1.2 by computing multiple chunk CVs in parallel and then pushing each of them into the CV stack one at a time. It can also be combined with either of the multi-threading strategies from §5.2. Rather than having each task or recursive leaf compress one chunk at a time, each can compress multiple chunks in parallel.

5.4 Memory Requirements

BLAKE3 has a larger memory requirement than BLAKE2, because of the chaining value stack described in §5.1.2. An incremental implementation needs space for 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^{10} bytes, giving a maximum tree depth of $64 - 10 = 54$. The CV stack thus requires $54 \cdot 32 = 1728$ bytes. The chunk state (cf. §5.1.1) also requires at least 104 bytes for the chaining value, the message block, and the chunk counter. The size of the reference implementation is 1880 bytes on the callstack.

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 22 and the CV stack is $22 \cdot 32 = 704$ 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 4 and the CV stack is $4 \cdot 32 = 128$ bytes.

6 Applications

As a general-purpose hash function, BLAKE3 is suitable whenever a collision-resistant or preimage-resistant hash function is needed to map some arbitrary-size input to a fixed-length output. BLAKE3 further supports keyed modes—in order to be used as a pseudorandom function, MAC, or key derivation function—as well as streaming and incremental processing features.

6.1 Pseudorandom Function and MAC

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.

6.2 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.

6.3 Stateful Hash Object

Trevor's thing: https://github.com/noiseprotocol/sho_spec/blob/master/sho.md

Due to its zero-overhead keying and variable-length output, BLAKE3 may be adapted into a SHO quite easily. Let l be some initial label. The initial key—the key at each step representing the SHO state—is obtained as $k_0 = \texttt{derive\_key}(l)$, after which we have $k_{i+1}, c_{i+1} = \texttt{keyed\_hash}(k_i, m_i)$, where m_i is the input at step i, and c_i is the "squeezing" output.

This SHO should be sound as long as BLAKE3 behaves like a random function, and there are no key collisions. So up to about 2^{128} compression function calls.

6.4 Verified Streaming

Because BLAKE3 is a tree hash, it supports new use cases that serial hash functions do not. One new use case is verified streaming. Consider a video streaming application that fetches video files from an untrusted host. The application knows the hash of the file it wants, and it needs to verify that the video data it receives matches that hash. With a serial hash function, nothing can be verified until the application has downloaded the entire file. But with BLAKE3, verified streaming is possible. The application can verify and play individual chunks of video as soon as they arrive.

To verify an individual chunk without re-hashing the entire file, we verify each parent node on the path from the root to that chunk. For example, suppose the file is composed of four chunks, like the four-chunk tree in Figure 1. To verify the first chunk, we start by fetching the root node. (Specifically, we fetch its message bytes, the concatenated chaining values of its children.) We compute the root node's CV as per §2.5 and confirm that it matches the 32-byte BLAKE3 hash of the entire file. Then, we fetch the root node's left child, which is the first chunk's parent. We compute that node's CV and confirm that it matches the first 32-bytes of the root node. Finally, we fetch the first chunk itself. We compute its CV as per §2.4 and verify that it matches the first 32-bytes of its parent. This verifies that the first chunk is authentic, and we can pass it along to application code.

To continue streaming, we can immediately fetch the second chunk. It shares the parent node of the first chunk, and its CV should match the second 32 bytes of that parent node. For the third chunk, we need to fetch its parent. The CV of that parent node should match the second 32 bytes of the root node, and then the CV of the third chunk should match the first 32 bytes of its parent. Note that whenever we fetch a parent node, we immediately use its first 32 bytes to check its left child's CV, and then we store its second 32 bytes to check its right child in the future. We can represent this with a stack of expected CVs. We push CVs onto this stack and pop them off, as we would with node pointers in a depth-first traversal of a binary tree. Note that this is different from the "CV stack" used for incremental hashing in §5.1.2; that stack holds CVs we have computed in the past, while this stack holds CVs we expect to compute in the future, and we manage them with different algorithms.

Observe that if the application eventually verifies the entire file, it will have fetched all the nodes of the tree in pre-order. This suggests a simple wire format for a streaming protocol: The host can concatenate all the parent nodes and chunks together in pre-order and serve the concatenated bytes as a single stream. If the client application knows the length of the file in advance, it does not need any other information to parse the stream and verify each node. In this format, the space overhead of the parent nodes approaches two CVs per chunk, or 6.25%.

If the client does not know the length of the file in advance, it may receive the length from the host, either separately or at the front of the stream. In this case, the length is untrusted. If the host reports an incorrect length, the effect will be that at least the final chunk will fail verification. For this reason, if an application reads the file sequentially, it does not need to explicitly verify the length. The stream may terminate with an error partway through if verification fails, as it might if the network connection failed. An important edge case is that, if the reported length is 0, the file consists of a single empty chunk, and the implementation must not forget to verify that the file hash matches the empty chunk's CV. (One way to make this mistake is for the implementation to return end-of-file as soon as the current read position equals the expected length, verifying nothing in the zero-length case.)

On the other hand, if an application implements seeking, length verification is required. Seeking past the reported end-of-file, or performing an EOF-relative seek, would mean trusting an unverified length. In these cases, the implementation must first verify the length by seeking to and verifying the final chunk. If that succeeds, the length is authentic, and the implementation can proceed with the caller's requested seek.

This verified streaming protocol has been implemented by the Bao tool. It is conceptually similar to the existing Tree Hash Exchange format, and to the BEP 30 extension of the BitTorrent protocol. However, neither of those prevents length extension, and the latter (if extracted from the BitTorrent protocol) does not provide second-preimage resistance [10, §8.5].

6.5 Incremental Update

Another new use case supported by BLAKE3 is incrementally updating the root hash when an input is edited in place. A serial hash function can efficiently append new bytes to the end of an input, but editing bytes at the beginning or in the middle requires re-hashing everything to the right of the edit. With BLAKE3, an application can edit bytes anywhere in its input and re-hash just the modified chunks and their transitive parent nodes.

For example, consider an input composed of four chunks, again like the four-chunk tree in Figure 1. Suppose we overwrite some bytes in the second chunk. To update the root hash, we first compute the second chunk's new CV as per §2.4. Then we recompute the CV for the second chunk's parent as per §2.5. Note that because the first chunk is unchanged, the first 32 message bytes for this parent node are unchanged also, and we do not need to re-read the first chunk. Finally, we recompute the CV of the root node, and we similarly do not need to re-read anything from the right half of the tree.

Note that this strategy cannot efficiently insert or remove bytes from the middle of an input. That would change the position of all the bytes to the right of the edit and force re-hashing. This is similar to the constraints of a typical filesystem, where appends and in-place edits are efficent, but insertions and removals within a file are either slow or unsupported.

7 Rationales

Acknowledgments

Thanks to Liming Luo for many discussions about the tree layout.

References

[1] Kevin Atighehchi and Alexis Bonnecaze. Asymptotic analysis of plausible tree hash modes for SHA-3. *IACR Trans. Symmetric Cryptol.*, 2017(4):212–239, 2017. doi:10.13154/tosc.v2017.i4.212-239.

[2] Kevin Atighehchi and Robert Rolland. Optimization of tree modes for parallel hash functions: A case study. *IEEE Trans. Computers*, 66(9):1585–1598, 2017. doi:10.1109/TC.2017.2693185.

[3] Jean-Philippe Aumasson. Too much crypto. In *Real World Crypto*, 2020.

[4] Jean-Philippe Aumasson, Jian Guo, Simon Knellwolf, Krystian Matusiewicz, and Willi Meier. Differential and invertibility properties of BLAKE. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 318–332. Springer, 2010. doi:10.1007/978-3-642-13858-4_18.

[5] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In 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*, pages 119–135. Springer, 2013. doi:10.1007/978-3-642-38980-1_8.

[6] Dongxia Bai, Hongbo Yu, Gaoli Wang, and Xiaoyun Wang. Improved boomerang attacks on round-reduced SM3 and keyed permutation of BLAKE-256. *IET Information Security*, 9(3):167–178, 2015. doi:10.1049/iet-ifs.2013.0380.

[7] 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.

[8] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302. IEEE, 2016. doi:10.1109/EuroSP.2016.31.

[9] Alex Biryukov, Ivica Nikolic, and Arnab Roy. Boomerang attacks on BLAKE-32. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2011. doi:10.1007/978-3-642-21702-9_13.

[10] 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.

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

[12] Thomas Espitau, Pierre-Alain Fouque, and Pierre Karpman. Higher-order differential meet-in-the-middle preimage attacks on SHA-1 and BLAKE. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 683–701. Springer, 2015. doi:10.1007/978-3-662-47989-6_33.

[13] Jian Guo, Pierre Karpman, Ivica Nikolic, Lei Wang, and Shuang Wu. Analysis of BLAKE2. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 402–423. Springer, 2014. doi:10.1007/978-3-319-04852-9_21.

[14] Yonglin Hao. The boomerang attacks on BLAKE and BLAKE2. In Dongdai Lin, Moti Yung, and Jianying Zhou, editors, *Information Security and Cryptology - 10th International Conference, Inscrypt 2014, Beijing, China, December 13-15, 2014, Revised Selected Papers*, volume 8957 of *Lecture Notes in Computer Science*, pages 286–310. Springer, 2014. doi:10.1007/978-3-319-16745-9_16.

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

[16] 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.

[17] Bozhan Su, Wenling Wu, Shuang Wu, and Le Dong. Near-collisions on the reduced-round compression functions of Skein and BLAKE. In Swee-Huay Heng, Rebecca N.

Wright, and Bok-Min Goi, editors, *Cryptology and Network Security - 9th International Conference, CANS 2010, Kuala Lumpur, Malaysia, December 12-14, 2010. Proceedings*, volume 6467 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2010. doi:10.1007/978-3-642-17619-7_10.

[18] Janos Vidali, Peter Nose, and Enes Pasalic. Collisions for variants of the BLAKE hash function. *Inf. Process. Lett.*, 110(14-15):585–590, 2010. doi:10.1016/j.ipl.2010.05.007.

Appendix A IV Constants

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

$$\begin{aligned} IV_0 &= \texttt{0x6a09e667} & IV_1 &= \texttt{0xbb67ae85} \\ IV_2 &= \texttt{0x3c6ef372} & IV_3 &= \texttt{0xa54ff53a} \\ IV_4 &= \texttt{0x510e527f} & IV_5 &= \texttt{0x9b05688c} \\ IV_6 &= \texttt{0x1f83d9ab} & IV_7 &= \texttt{0x5be0cd19} \end{aligned}$$

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:

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

That is, a round applies a G function to each of the columns in parallel, and then to 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