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 BLAKE2 cryptographic hash that is both faster and also more consistently fast across different platforms and input sizes. 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 Cascade Lake-SP, peak single-threaded throughput is 4$\times$ that of BLAKE2b, 8$\times$ that of SHA-512, and 12$\times$ that of SHA-256, and it can scale further using multiple threads. BLAKE3 is also efficient on smaller architectures: throughput on a 32-bit ARM1176 core is 1.3$\times$ that of SHA-256 and 3$\times$ that of BLAKE2b and SHA-512. Unlike BLAKE2 and SHA-2, with different variants better suited for different platforms, BLAKE3 is a single algorithm with no variants. It provides a simplified API supporting all the use cases of BLAKE2, including keying and extendable output. The tree structure also supports new use cases, such as verified streaming and incremental updates.

Version 20200221164500.

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 [9], the winner of the Password Hashing Competition in 2015.

A drawback of BLAKE2 has been its large number of incompatible variants. The performance tradeoffs between different variants are subtle, and library support is uneven. BLAKE2b is the most widely supported, but it is not the fastest on most platforms. BLAKE2bp and BLAKE2sp are more than twice as fast on modern x86 processors, but they are sparsely supported and rarely adopted.

BLAKE3 eliminates this drawback. It is a single algorithm with no variants, designed for consistent high performance in software on all platforms. The biggest changes from BLAKE2 to BLAKE3 are:

- **A binary tree structure**, providing an unbounded degree of parallelism.
- **Fewer rounds** in the BLAKE2s-based compression function (7 instead of 10).
- **Three modes**: hashing, keyed hashing, and key derivation. These modes replace the BLAKE2 parameter block and provide a simpler API.
- **Zero-cost keyed hashing**, using the space formerly occupied by the parameter block for the key.
- **Built-in extendable output** (a.k.a. XOF), which is parallelizable and seekable, like BLAKE2X but unlike SHA-3 or HKDF.

BLAKE3 splits its input into 1 KiB chunks and arranges them as the leaves of a binary tree. Each chunk is compressed independently, so the degree of parallelism is equal to the number of chunks [1, 2]. This makes BLAKE3 very fast on modern processors, where it can take advantage of SIMD instructions and multiple cores. Another benefit of compressing chunks in parallel is that the implementation can use SIMD vectors of any width, regardless of the word size of the compression function (see §7.2). That leaves us free to select a compression function that is efficient on smaller architectures, without sacrificing peak throughput on x86-64. Thus our choice of BLAKE2s instead of BLAKE2b.

BLAKE3 has the same 128-bit security level and 256-bit default output size as BLAKE2s. The internal mixing logic of the compression function is also the same, with a simplified message schedule derived from a single repeated permutation. Based on existing cryptanalysis of BLAKE and BLAKE2, BLAKE3 reduces the number of rounds in the compression function from 10 to 7 (see [3] for a 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

The input of BLAKE3 is split into contiguous chunks of 1024 bytes, such that 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. Two rules determine the structure of the tree:

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.

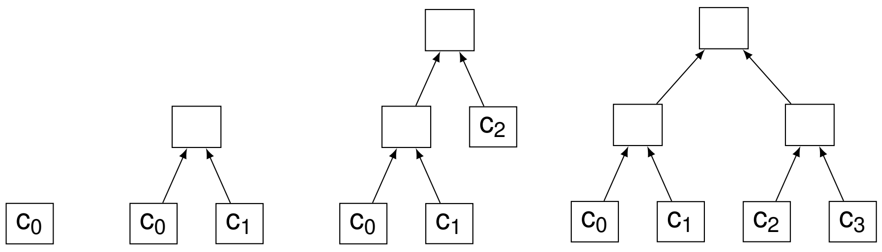

Figure 1: Example tree structures, from 1 to 4 chunks.

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

2.2 Compression Function

The compression function works with 32-bit words, and takes the following inputs:

- The input chaining value, $h_0 \dots h_7$ (256 bits).
- The message block, $m_0 \dots m_{15}$ (512 bits).
- 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 (32 bits).
- A set of domain separation bit flags, d (32 bits).

Table 1: Constants $IV_0 \ldots IV_7$ used in the initialization of BLAKE3.

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

The compression function initializes its 16-word internal state $v_0 \ldots v_{15}$ 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 $IV_0 \ldots IV_7$ constants are the same as in BLAKE2s, and they are reproduced in Table 1.

The compression function applies a 7-round keyed permutation $v' = E(m, v)$ to the state $v_0 \ldots v_{15}$, keyed by the message $m_0 \ldots m_{15}$.

The round function is the same as in BLAKE2s. A round consists of the following 8 operations:

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

That is, a round applies a G function to each column of the 4×4 state in parallel, and then to each of the diagonals in parallel.

The "quarter-round" $G_i(a, b, c, d)$ is defined as follows. Let $\oplus$ denote xor, $+$ denote addition modulo 2^{32}, $\ggg$ denote bitwise right rotation, and m_x be the xth message word:

$$\begin{aligned} a &\leftarrow a + b + m_{2i+0} \\ d &\leftarrow (d \oplus a) \ggg 16 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 12 \\ a &\leftarrow a + b + m_{2i+1} \\ d &\leftarrow (d \oplus a) \ggg 8 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 7 \end{aligned}$$

After each round (except the last one where it would be useless), the message words are permuted according to Table 2.

Table 2: Permutational key schedule for BLAKE3's keyed permutation.

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

Table 3: 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_CONTEXT`	2^5
`DERIVE_KEY_MATERIAL`	2^6

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 $E(m, v)$, the compression function may be written as

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

The output of the compression function is most 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 3.

2.3 Modes

BLAKE3 defines three domain-separated modes: `hash(input)`, `keyed_hash(key, input)`, and `derive_key(context, key_material)`. These are intended to be user-facing APIs in an implementation. The `hash` mode takes an input of any byte length $0 \leq \ell < 2^{64}$. The `keyed_hash` mode takes an input of any length together with a 256-bit key. The `derive_key` mode takes a context string and key material, both of any length. As described in §2.6, all three modes can produce output of any length. The intended usage of `derive_key` is given in §6.2.

The modes differ from each other in their key words $k_0 \ldots k_7$ and in the additional flags they set for every call to the compression function. In the `hash` mode, $k_0 \ldots k_7$ are the constants $IV_0 \ldots IV_7$, and no additional flags are set. In the `keyed_hash` mode, $k_0 \ldots k_7$ are parsed in little-endian order from the 256-bit key given by the caller, and the `KEYED_HASH` flag is set for every compression.

The third mode, `derive_key`, has two stages. First the context string is hashed, with $k_0 \ldots k_7$ set to the constants $IV_0 \ldots IV_7$, and the `DERIVE_KEY_CONTEXT` flag set for every compression. Then the key material is hashed, with $k_0 \ldots k_7$ set to the first 8 output words of the first stage, and the `DERIVE_KEY_MATERIAL` flag 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. If necessary, the last block is padded with zeros to be 64 bytes.

Each block is parsed in little-endian order into message words $m_0 \ldots m_{15}$ and compressed. The input chaining value $h_0 \ldots h_7$ for the first block of each chunk is composed of the key words $k_0 \ldots k_7$. 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 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 3), 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.

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,

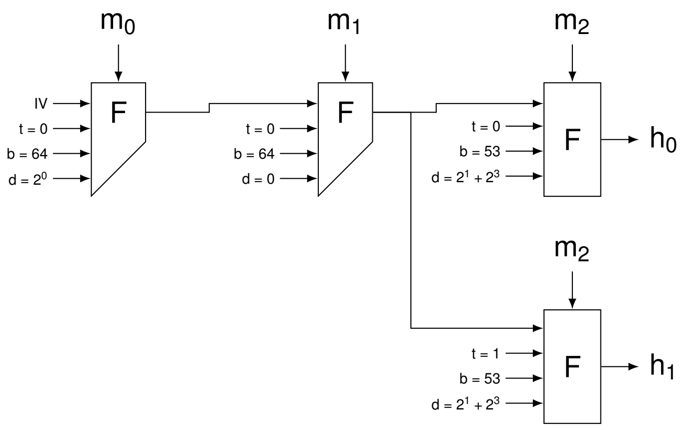

Figure 2: Example of compression function inputs when hashing a 181-byte input (m_0, m_1, m_2) into a 128-byte output (h_0, h_1). Trapezia indicate that the compression function output is truncated to 256 bits.

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 $0 \leq \ell < 2^{64}$. 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. See Figure 2 for an example.

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. The caller can extract as much output as needed, without knowing the final length in advance.

3 Performance

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

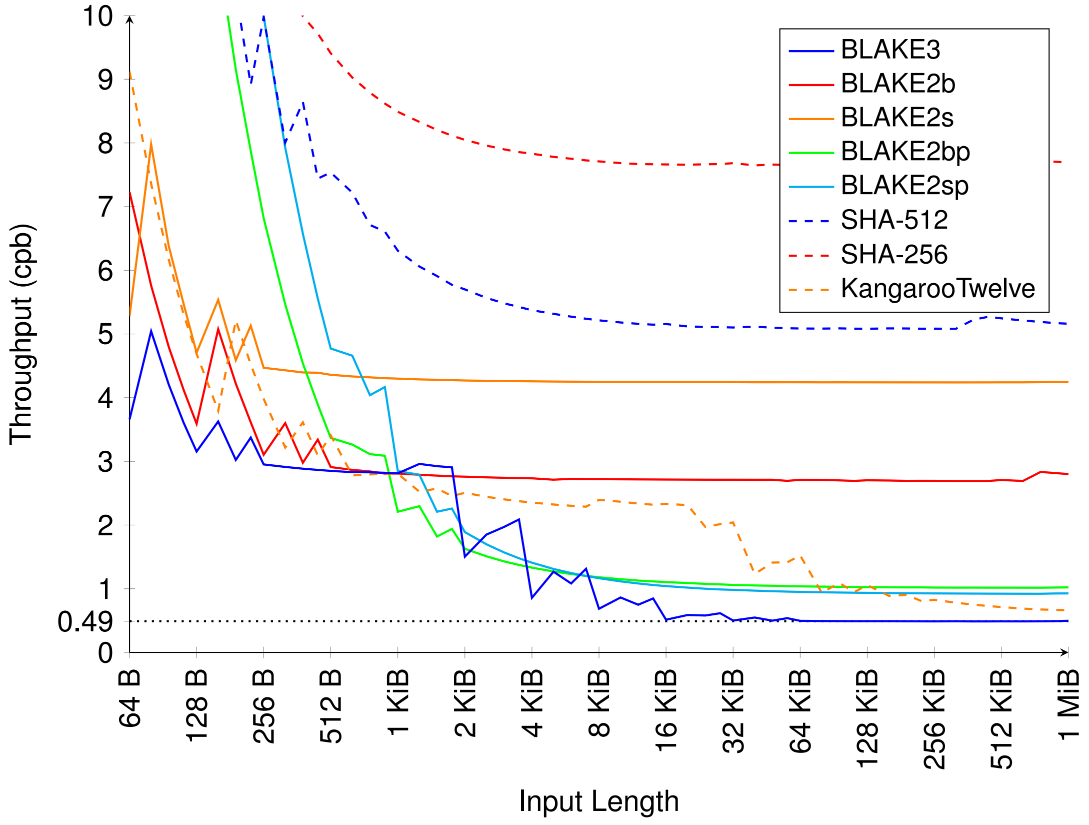

Figure 3: Throughput for single-threaded BLAKE3 and other hash functions on an AWS c5.metal instance, measured in cycles per byte. This is an Intel Cascade Lake-SP processor with AVX-512 support. Lower is faster.

- The implementation can compress groups of chunks in parallel using SIMD (cf. §5.3). This is visible at 2 KiB and above in Figure 3.
- The implementation can use multiple threads (cf. §5.2). This is the subject of Figure 4. Note that Figures 3 and 5 are single-threaded.
- The compression function performs fewer rounds than in BLAKE2s. This improves performance across the board, but it is especially visible in the difference between BLAKE3 and BLAKE2s in Figure 5.
- The compression function uses 32-bit words (cf. §7.2), which perform well on smaller architectures. This is the subject of Figure 5.
- The 64-byte block size is relatively small, which helps performance for very short inputs. This is visible at 64 bytes (the left edge) in Figures 3 and 5.

Figure 3 shows the throughput of single-threaded BLAKE3 on modern server hardware. This benchmark ran on an AWS c5.metal instance with a pair of Intel Xeon Platinum 8275CL (Cascade Lake-SP) processors, which support AVX-512 vector instructions.

The BLAKE3 curve passes through several different regimes in Figure 3. For a single chunk of input, up to 1 KiB, BLAKE3 closely mirrors BLAKE2s (as it does throughout all

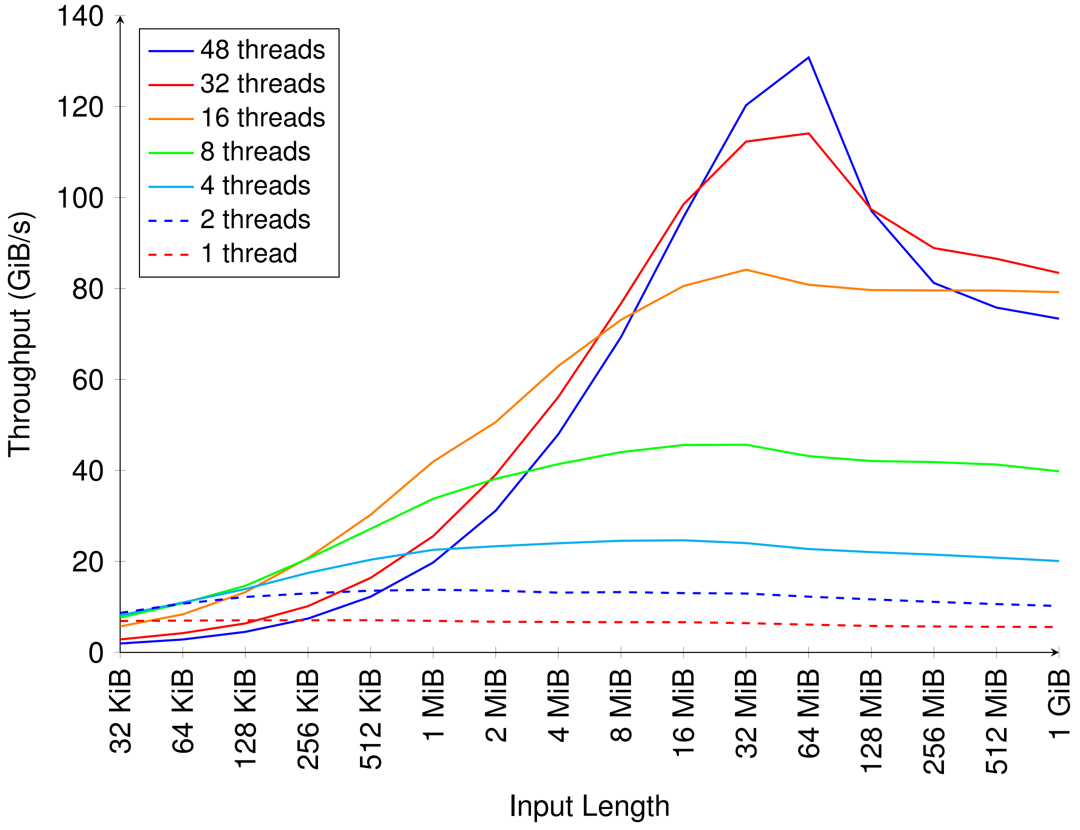

Figure 4: Throughput for multi-threaded BLAKE3 on the same c5.metal instance as in Figure 3, measured in GiB/s, varying the number of threads. This machine has 48 physical cores. Higher is faster.

of Figure 5). This is because their compression functions are very similar, apart from the number of rounds. Between 1 and 2 KiB, the gap between BLAKE3 and BLAKE2s closes slightly. This is because BLAKE3 divides the input into two chunks with a parent node, and compressing the parent node adds overhead. At 2 KiB, BLAKE3 begins compressing multiple chunks in parallel. This implementation uses 2-way row-based parallelism at 2 KiB, 4-way row-based parallelism at 4 KiB, 8-way word-based parallelism at 8 KiB, and 16-way word-based parallelism at 16 KiB (see §5.3). Beyond 16 KiB, throughput is relatively stable, until memory bandwidth effects appear for very long inputs.

On this platform, the biggest performance difference between algorithms is how much they can benefit from SIMD. Only BLAKE3 and KangarooTwelve are able to take full advantage of AVX-512. The fixed-interleaved tree structures of BLAKE2bp and BLAKE2sp limit those algorithms to 256-bit vectors. Other hash functions are much slower, because they cannot compress multiple blocks in parallel. BLAKE3 and KangarooTwelve are close to each other in peak throughput, but KangarooTwelve needs more input to reach its peak, in part because of its larger 8 KiB chunk size.

Figure 4 shows the throughput of multi-threaded BLAKE3. This benchmark ran on the same c5.metal instance as in Figure 3, which has 48 physical cores. This implementation uses the Rust Rayon library to manage threading, with the recursive divide-and-conquer approach described in §5.2. Below 128 KiB of input, the overhead of threading is high, and increasing

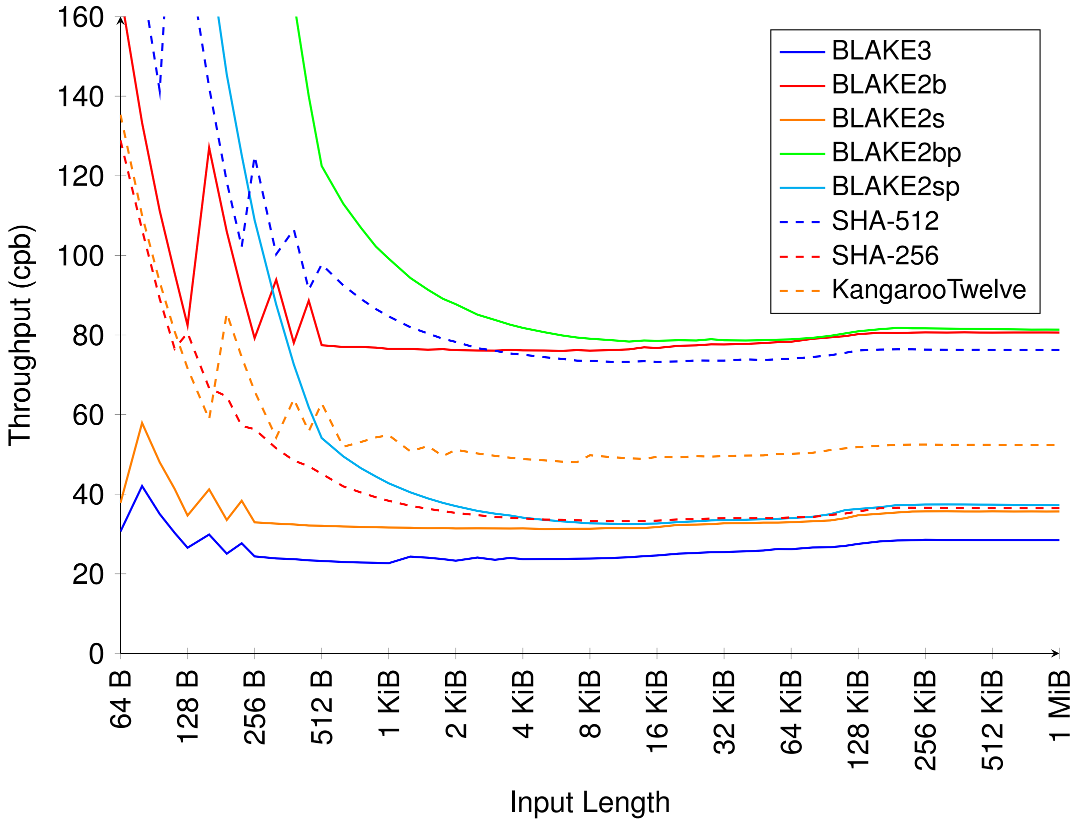

Figure 5: Throughput for single-threaded BLAKE3 and other hash functions on a Rasperry Pi Zero, measured in cycles per byte. This is a 32-bit ARM1176 processor. Lower is faster.

the number of threads causes a performance drop. For longer inputs, throughput scales almost linearly with the number of threads, up to 16 threads. Above 16 threads, throughput levels off due to memory bandwidth effects.

Figure 5 shows the throughput of single-threaded BLAKE3 on a smaller embedded platform. This benchmark ran on a Raspberry Pi Zero with a 32-bit ARM1176 processor. Here, the biggest difference between algorithms is whether they use 32-bit or 64-bit words. As noted above, BLAKE3 and BLAKE2s have similar performance profiles here, because their compression functions are closely related, and because there is no multi-core or SIMD parallelism for BLAKE3 to exploit.

Figures 3 and 5 also highlight that BLAKE3 performs well for very short inputs, at or below its 64-byte block size. Very short inputs get padded up to a full block, and algorithms with smaller block sizes benefit from compressing less padding. The fixed-interleaved tree structures of BLAKE2bp and BLAKE2sp are especially costly when the input is short, because they always compress a parent node and a fixed number of leaves. The advantage of a fixed-interleaved tree structure comes at medium input lengths, where BLAKE3 has not yet reached a high degree of parallelism. This is the regime around 1–4 KiB in Figure 3, where BLAKE2bp and BLAKE2sp pull ahead.

The SHA-2 implementations used in these benchmarks are from OpenSSL 1.1.1.d, and the KangarooTwelve implementations are from the eXtended Keccak Code Package (compiled with `gcc -Os` for the ARM1176).

4 Security

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 rationale for targeting 128-bit security can be found in [3, §2].

The key length is nevertheless 256 bits, as an extra defense layer against weakly chosen keys, key material leaks, and potentially multi-target attacks.

4.2 Compression Function

The BLAKE3 compression function is, at a high level, identical to that of BLAKE2s, with the exception that:

- BLAKE2's f_0 and f_1 "flags" are replaced by the b and d inputs, which encode the functional context of the compression function instance.
- Finalization returns 16 words instead of 8, and only applies the feed-forward on the second half of the output.

The latter change causes the BLAKE3 function to no longer be indifferentiable, unlike its predecessor [22]. Indeed, an attacker with oracle access to $C(\dots)$ and $E(\dots)$ that obtains $y = C(x, m, b, d)$ can compute $(x, b, d) = E^{-1}(m, (y_l, y_h) \oplus (y_h, 0) \oplus (0, x_l))$; a simulator with access to a random oracle cannot perform the same inversion, and as such this scheme—much like Davies-Meyer—is not indifferentiable from a random oracle. This causes some surmountable complications in the next section.

We do note that the truncated compression function used in every compression except the root nodes is indifferentiable. The simulator and proof are very similar to those in [22], with the only change being the removal of the feed-forward, which has no effect on security when truncating the output.[1]

4.3 Mode of Operation

One important aspect of any hash mode, parallel or otherwise, is its *soundness*. Several works [7, 12–14] have studied the requirements for the mode of operation of a hash function such that, when instantiated with an ideal compression function, block cipher, or permutation, it remains indistinguishable from a random oracle. We adopt the requirements from Daemen et al. [13], which are:

- Subtree-freeness, i.e, no valid tree may be a subtree of another;
- Radical-decodability, i.e., preventing collisions by ambiguity of the tree shape;
- Message-completeness, i.e., the entire message can be recovered from the inputs to the primitive in question.

[1]A recent work [11] improves upon [14, 22] with a tighter bound for the indifferentiability of truncated permutations. A similar approach would work for BLAKE2 or truncated BLAKE3, but the benefits for our parameters are negligible.

Subtree-freeness Subtree-freeness ensures that generalized length-extension attacks, like the ones that plagued Merkle-Damgård, cannot happen. To ensure subtree-freeness, BLAKE3 defines the `ROOT` flag, which is only set on the last compression output. Thus, for any valid tree, a valid subtree must have the same root node. There are two cases to consider here:

1. The `ROOT` flag is combined with `CHUNK_END`; in this case we are dealing with a single-chunk message, and the root nodes must be the same, as well as its parents, grandparents, etc, until we reach a `CHUNK_START` node. Because both subtrees must start with `CHUNK_START`, we conclude that both trees must be the same.
2. The `ROOT` flag is applied to a parent node. Here the root node necessarily has two parents, each of which must be equal, recursing until they hit the same set of leaves marked by the `CHUNK_END` flag.

Radical-decodability For radical-decodability, we can define the subset of final nodes. For each final node, one can define a function radical() that returns a CV position for any final node. Here we have, again, two cases:

1. The final node has the `CHUNK_END` flag set; here the radical is in the CV bits.
2. The final node is a parent node; here the CV comes in the message bits.

Message-completeness The entire input can be recovered from the message bits input to the chunks.

If only considering 256-bit outputs or shorter, by [13, Theorem 1], for a distinguisher $\mathcal{D}$ of total complexity q we have

$$\mathrm{Adv}^{\mathrm{diff}}_{\mathrm{BLAKE3}}(\mathcal{D}) \leq \frac{\binom{q}{2}}{2^{256}} + \tag{1}$$

$$\frac{\binom{q}{2}}{2^{512}} + \frac{\binom{q}{2}}{2^{256}} + \frac{q}{2^{128}} \,. \tag{2}$$

Here (1) corresponds to the contribution of [13, Theorem 1] and (2) corresponds to the indifferentiability of the truncated compression function [22].

For the full-length output, we do not have a truncated ideal cipher, but instead use a feed-forward to prevent inversion. We remark, however, that the truncation requirement in [13, Theorem 3] is in place purely to prevent the distinguisher from inverting an output; as such, the feed-forward serves the same purpose as truncation, and the result still follows as long as the other soundness requirements are in place.

4.4 Cryptanalysis

The round reduction from 10 to 7 rounds is based on existent cryptanalysis, along with novel research. High confidence in the security of 7 rounds follows from cryptanalysis efforts over 10 years, first on BLAKE [4, 6, 10, 16, 21, 25, 26] and then on BLAKE2 [17–19]. Table 4 summarizes the current state of the art in BLAKE(2) cryptanalysis.

Table 4: Summary of cryptanalysis on BLAKE and BLAKE2.

Primitive	Type	Rounds	Complexity	Reference
BLAKE-256 k.p.	Boomerang	7	2^{44}	[6, 19]
BLAKE-256 k.p.	Boomerang	8	2^{198}	[19]
BLAKE-256 k.p.	Differential	4	2^{192}	[16]
BLAKE-256 k.p.	Differential	6	2^{456}	[16]
BLAKE-256 k.p.	Impossible Differential	6.5	–	[18]
BLAKE-256 c.f.	Boomerang	7	2^{242}	[10]
BLAKE-256 c.f.	Near collision (152/256)	4	2^{21}	[25]
BLAKE-256 c.f.	Near collision (232/256)	4	2^{56}	[4]
BLAKE-256 c.f.	Preimage	2.5	2^{241}	[21]
BLAKE-256 c.f.	Pseudo-preimage	6.75	$2^{253.9}$	[17]
BLAKE2s k.p.	Boomerang	7.5	2^{184}	[19]
BLAKE2s k.p.	Impossible Differential	6.5	–	[18]
BLAKE2s k.p.	Rotational	4	2^{489}	[18, 20]
BLAKE2s c.f.	Boomerang	5	$\approx 2^{86}$	[10, 18]
BLAKE2s c.f.	Pseudo-preimage	6.75	$2^{253.8}$	[17]

Furthermore, our own searches for good[2] differential trails, shown in Table 5, show that by restricting input differences the initialization of BLAKE2/3 makes things harder for the attacker, with probabilities quickly becoming very low. This was also observed in [18, §7].

The boomerang attacks of, e.g., [6, 10, 19] exploit the high probabilities on the first few rounds to connect two trails and get through a few more rounds. This is an established technique to get around the quick lowering of differential characteristics' probabilities. But such attacks do not present a threat to BLAKE3, considering its input constraints on the compression function (that is, the IV constants), plus the 128-bit security target.

The general survey and analysis in [3] also pleads in favor of reduced round values, arguing that BLAKE2s and BLAKE2b would be as safe with 7 and 8 rounds, respectively.

5 Implementation

5.1 Incremental Hashing

An incremental implementation of BLAKE3 has 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 BLAKE2b or BLAKE2s, and it will be familiar to implementers of other hash functions. The CV stack is less familiar. Simplifying the management of the CV stack is especially important for a clear and correct implementation of BLAKE3. This section describes how this is done in the reference implementation, as an aid to implementers.

[2]Optimal under certain assumptions, which are not entirely correct, cf. [23]. Nevertheless we find these results useful as a first approximation.

Table 5: 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 rotationally symmetric differences were sought exclusively.

Function	0.5	1	1.5	2	2.5	3	3.5
BLAKE-256 k.p/c.f.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-6}	2^{-7}	2^{-38}
BLAKE2s k.p.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-6}	2^{-7}	2^{-38}
BLAKE3 k.p.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-7}	2^{-21}	2^{-52}
BLAKE2s c.f.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-32}	$[2^{-88}, 2^{-48})$	-
BLAKE3 c.f.	2^{-0}	2^{-0}	2^{-0}	2^{-1}	2^{-32}	$[2^{-159}, 2^{-48})$	-
BLAKE{-256,2s} (hash)	2^{-0}	2^{-1}	2^{-32}	$\geq 2^{-190*}$	-	-	-
BLAKE3 (hash)	2^{-0}	2^{-1}	2^{-29}	$\geq 2^{-185*}$	-	-	-
BLAKE-256 c.f. (fixed m)	2^{-0}	2^{-2}	2^{-12}	2^{-39}	-	-	-
BLAKE{2s,3} c.f. (fixed m)	2^{-0}	2^{-7}	2^{-32}	$\geq 2^{-161*}$	-	-	-

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 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 to this tree only after the caller has supplied at least 1 byte for the following chunk, so we know that none of these chunks or parent nodes is the root of the tree, and we do not need to worry about the `ROOT` flag yet.

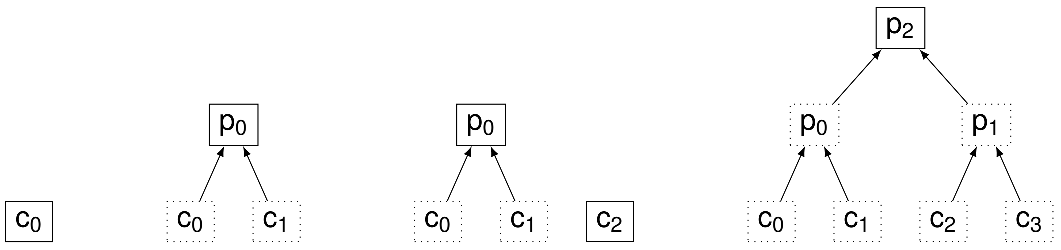

Figure 6: An incomplete tree growing incrementally from 1 to 4 chunks. Dotted boxes represent CVs that no longer need to be stored.

When the first chunk CV (c_0) is added, it is alone. When the second chunk CV (c_1) is added, it completes a two-chunk subtree, and we can compute the first parent CV (p_0). The third chunk CV (c_2) does not complete any subtrees. Its final position in the tree structure will depend on whether it gets a right sibling (see Figure 1), so we cannot create any parent nodes for it yet. The fourth chunk CV (c_3) provides a right sibling for c_2 and completes two subtrees, one of two chunks and one of four chunks. First it merges with c_2 to compute p_1, then p_1 merges with p_0 to compute p_2.

Note that once a subtree is complete, none of its child CVs will be used again, and we do not need to store them any longer. These unneeded CVs are represented by dotted boxes in Figure 6. Solid boxes represent CVs that are still needed. These are what we store in the CV stack, ordered based on when we computed them, with newer CVs on top.

Look through Figure 6 again, this time paying attention to the state of the CV stack at each step. When c_0 is added to the tree, it is the only entry in the stack. When c_1 is added, we pop c_0 off the stack to merge it, and p_0 becomes the only entry. When c_2 is added, there are two entries in the stack, with c_2 on top because it is newer. When c_3 is added, we perform two merges, first popping off c_2 and then popping off p_0, and only p_2 remains in the stack.

Note how the stack order of c_2 and p_0 above match the order in which we need to retrieve them for merging. By ordering CVs in the stack from newest at the top to oldest at the bottom, we also implicitly order them from tree-right to tree-left, and from the smallest subtree to the largest subtree. This invariant means that the next CV we need to merge is always on top of the stack.

The key question is then, when a new chunk CV is added to the tree, how many subtrees does it complete? Which is to say, how many CVs should we pop off the stack and merge with it, before pushing on the final result?

To answer this question, note another pattern. The entries in the CV stack behave like the 1-bits in the binary representation of the total number of chunks so far. For example with 3 chunks (`0b11`), there are two entries in the stack, corresponding to the two 1-bits. With 4 chunks (`0b100`), only one entry remains, corresponding to the single 1-bit.

To see why this pattern holds, note that any completed subtree represented by a CV in the stack contains a number of chunks that is a power of 2. Furthermore, each subtree is a distinct power of 2, because when we have two subtrees of the same size, we always merge them. Thus, the subtrees in the stack correspond to the distinct powers of 2 that sum up to the current total number of chunks. This is equivalent to the 1-bits in the binary representation of that number.

In this sense, popping a CV off the stack to merge it is like flipping a 1-bit to 0, and pushing the final result back onto the stack is like flipping a 0-bit to 1. The fact that we do two merges when adding the fourth chunk, corresponds to the fact that two 1-bits flip to 0 as the total changes from 3 to 4. In other words, we do two merges when we add the fourth chunk, because there are two trailing 1-bits in the number 3, and equivalently two trailing 0-bits in the number 4.

This pattern leads to an algorithm for adding a new chunk CV to the tree: For each trailing 0-bit in the new total number of chunks, pop a CV off the stack, and merge it with the new CV we were about to add. Finally, push the fully merged result onto the stack. Listing 1 shows this algorithm as it appears in the Rust reference implementation.

Once we reach the end of the input, we can use a simpler algorithm to compute the root hash. Finalize the CV of the current chunk, which may be incomplete or empty. Then, merge that CV with every CV currently in the stack, regardless of the number of chunks so far. Set

```
fn add_chunk_chaining_value(&mut self, mut new_cv: [u32; 8], mut total_chunks: u64) {
    // This chunk might complete some subtrees. For each completed subtree,
    // its left child will be the current top entry in the CV stack, and
    // its right child will be the current value of `new_cv`. Pop each left
    // child off the stack, merge it with `new_cv`, and overwrite `new_cv`
    // with the result. After all these merges, push the final value of
    // `new_cv` onto the stack. The number of completed subtrees is given
    // by the number of trailing 0-bits in the new total number of chunks.
    while total_chunks & 1 == 0 {
        new_cv = parent_cv(self.pop_stack(), new_cv, self.key, self.flags);
        total_chunks >>= 1;
    }
    self.push_stack(new_cv);
}
```

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

the ROOT flag for the last parent node to be merged. Or if there are no CVs in the stack, and thus no parents to merge, set the ROOT flag for chunk finalization instead.

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 (see §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 requires 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 recursive divide-and-conquer approach can be more efficient. 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 call stack, each pair of left and right child CVs is compressed into a parent CV.

This recursive approach is good fit for 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 call site.

The recursive approach is simplest when the entire input is available at once, since no CV stack is needed in that case. 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 algorithm works with no modification. If each incremental input is instead a variable size (for example if input from the caller is compressed directly without copying), the implementation needs to carefully 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 to achieve that the implementation might need to break up an incremental input into smaller pieces. In the non-buffering case, the implementation must also take care not to compress a parent node that could be the root, when it is unknown whether more input is coming.

5.3 SIMD

There are two primary 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 (see §2.2) 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.

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.

A third approach is also possible, though it is less widely applicable than the first two. Entire 4-word rows can be stored together within a vector, as in the first approach, but larger vectors can be used to store multiple such rows, each belonging to a different state matrix. For example, 256-bit vectors can store two full rows, thus compressing two chunks in parallel. Diagonalization and undiagonalization steps are still required in this arrangement, so the second approach is preferable when enough input is available. However, on platforms like x86 that do not support 64-bit vectors, this approach is the only way to implement 2-way SIMD parallelism for 2-chunk inputs. The benefit of using wider vectors may also outweigh the cost of diagonalization, though this varies between different microarchitectures. The performance of this approach is visible at 2 KiB and 4 KiB in Figure 3.

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 call stack.

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. Similarly, 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 provides 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 provides a key derivation mode, `derive_key`. This mode accepts a context string of any length and key material of any length, and it produces a derived key of any length. The context string should be hardcoded, globally unique, and application-specific. A good default format for context strings is `"[application] [commit timestamp] [purpose]"`, e.g., `"example.com 2019-12-25 16:18:03 session tokens v1"`.

This mode should not be used with a dynamic context string. The context string should not contain variable data, like salts, IDs, or the current time. (If needed, those can be part of the key material, or mixed with the derived key afterwards.) Higher-level abstractions that wrap `derive_key` should either hardcode their own context string, or impose the same requirement on their caller. The purpose of this requirement is to ensure that there is no way for an attacker in any scenario to cause two different applications or components to inadvertently use the same context string. The safest way to guarantee this is to prevent the context string from including input of any kind.

Given this requirement, we then make an explicit exception to an otherwise universal rule of practical cryptography: Applications may use `derive_key` with key material that is already in use with other algorithms. This includes other hash functions, ciphers, and abstractions like HMAC and HKDF. This also includes the `hash` and `keyed_hash` modes, which are domain-separated from `derive_key`. The only limitation in practice is algorithms that forbid key reuse entirely, like a one-time pad. (For other theoretical limitations, see §7.8.)

In general, using the same key with multiple algorithms is not recommended. Some algorithms are built from the same core components, and it can be difficult to know when two algorithms might have related output, or when one algorithm might publish a value that another algorithm considers secret. One especially relevant example of this problem is HMAC and HKDF themselves. Although these algorithms have entirely different purposes, HKDF is defined in terms of HMAC. In fact, in the “expand only” mode of HKDF, its output is equivalent to a single call to HMAC. If an application used the same key with HMAC and with HKDF, and an HMAC input happened to collide with an HKDF context string, this interaction could ruin the application’s security.

The standard advice for such an application is to use its original key only with a key derivation function, and to derive independent subkeys for other algorithms. This is good and practical advice in many cases. But it may not be practical when an application or a protocol evolves over time. If a key was originally used only with HMAC, for example, and perhaps years later a second use case arises, the developers may have painted themselves into a corner. Backwards compatibility might prevent them from using a derived subkey with HMAC. In cases like this, the standard advice amounts to either building a time machine, or over-engineering applications and protocols to derive subkeys at every step, in anticipation of unknown future use cases.

This is why it is valuable for `derive_key` to make an exception to the rule against mixing algorithms. It makes key derivation a practical option not only for new designs, but also for existing applications that are adding new features. In exchange, it is the responsibility of each caller to provide (or for higher-level abstractions, to require) a hardcoded, globally unique, application-specific context string, which can never be made to collide with any other.

The `derive_key` mode is intended to replace the BLAKE2 personalization parameter for most of its use cases. Key derivation can encourage better security than personalization with a shared key, 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.

6.3 Stateful Hash Object

To abstract away various uses of the hash function in the Noise protocol framework, the notion of *stateful hash objects* (SHO) [24] has been suggested. An SHO is an interface with 4 methods:

- `init(label)`: Initializes the internal state using a custom domain-separation label.
- `absorb(data)`: Hashes an arbitrary number of bytes into the state.

- `ratchet()`: Updates the state to be a one-way cryptographic function of all preceding inputs, with empty buffers.
- `squeeze(length)`: Produces an arbitrary amount of pseudorandom output based on the current state. The SHO is not used again after this.

Due to its zero-overhead keying and variable-length output, it is straightforward to adapt BLAKE3 into an SHO. The internal state is an incremental implementation of `keyed_hash`:

- The label key is obtained as `hash(label)`;
- The incremental hasher is initialized with the label key;
- `absorb(data)` updates the incremental hasher as usual;
- `ratchet()` finalizes the incremental hasher, extracts the first 32 bytes of output, and then re-initializes the hasher with those 32 bytes as the key;
- `squeeze(length)` finalizes the incremental hasher, skips the first 64 bytes of output, and then extracts `length` bytes. We only need to skip 32 bytes to conceal the `ratchet` key, but skipping a full 64-byte output block is more efficient.

6.4 Verified Streaming

Because BLAKE3 is a tree hash, it supports new use cases that serial hash functions do not. One such 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. 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, note that neither of those prevents length extension, and the latter (if extracted from the BitTorrent protocol) does not provide second-preimage resistance [13, §8.5].

6.5 Incremental Updates

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 efficient, but insertions and removals within a file are either slow or unsupported.

7 Rationales

This section goes into more detail about some of the performance tradeoffs and security considerations in the BLAKE3 design.

7.1 Chunk Size

BLAKE3 uses 1 KiB chunks. The chunk size is a tradeoff between the peak throughput for long inputs, which benefits from larger chunks, and the degree of parallelism for medium-length inputs, which benefits from shorter chunks.

The benefit of a larger chunk size is that the tree contains fewer parent nodes, so the overhead cost of compressing them is lower. The number of parent nodes is equal to the number of chunks minus one, so doubling the chunk size cuts the number of parent nodes in half. However, note that parent nodes at the same level of the tree can be compressed in parallel, so having twice as many parent nodes does not necessarily mean that compressing them takes twice as long.

The benefit of a smaller chunk size is a higher potential degree of parallelism. This is irrelevant for very long inputs, where all the SIMD lanes and possibly CPU cores of the machine will be occupied regardless. But at medium input lengths, this has a huge impact on performance. Consider an 8 KiB input. With a chunk size of 1 KiB, this input can occupy 8 SIMD lanes, which lets the implementation use AVX2 vector instructions on modern x86 processors. But if we increased the chunk size to 2 KiB, the same input would only occupy 4 SIMD lanes, and the high throughput of AVX2 would be left on the table.

Thus we want to pick the smallest chunk size we can to maximize medium-length parallelism, without incurring "excessive" overhead from parent nodes. The point of comparison here is the peak throughput for a very large chunk size, like 64 KiB, where parent node overhead becomes small enough to be negligible. In our measurements, we find that implementations with 1 KiB chunks can reach at least 90% of that peak throughput.

There are other minor performance considerations to be aware of. A larger chunk size slightly reduces the space requirement of the CV stack as described in §5.4. Also, verified streaming requires buffering chunks (see §6.4), and incremental updates require re-hashing chunks (see §6.5), so both of those use cases benefit from a smaller chunk size.

7.2 Word Size

BLAKE3 uses 32-bit words. The performance tradeoffs in the choice between 32-bit and 64-bit words are complex, and they depend on both the platform and the size of the input.

It is common advice that SHA-512 (because it uses 64-bit words) is faster on 64-bit platforms than SHA-256 (because it uses 32-bit words). A similar effect applies to BLAKE2b and BLAKE2s. However, this effect does not apply to algorithms that incorporate more SIMD parallelism. BLAKE2sp has a higher peak throughput than BLAKE2bp on x86-64. This is because the SIMD instructions that operate on 8 lanes of 32-bit words have the same throughput as those that operate on 4 lanes of 64-bit words. The key factor for exploiting

SIMD performance is not word size but vector size, and both of those cases occupy a 256-bit vector. (The remaining effect making BLAKE2sp faster is that its smaller states require fewer rounds of compression.)

For that reason, peak throughput on x86-64 is largely independent of the word size. The performance differences that remain are restricted to shorter input lengths: 32-bit words perform better for lengths less than one block, while 64-bit words perform better for lengths between one block and one chunk, comparable to BLAKE2s and BLAKE2b. These effects are minor, and in any case short input performance is often dominated by other sources of overhead in typical applications.

Instead, the decisive advantage of 32-bit words over 64-bit words is performance on smaller architectures and embedded systems. BLAKE3 is designed to be a single general-purpose hash function, suitable for users of both BLAKE2b and BLAKE2s. On 32-bit platforms like the ARM1176, where BLAKE2s does especially well, a 64-bit hash function could be a performance regression. Choosing 32-bit words is a substantial benefit for these platforms, with little or no downside for x86-64. Cross-platform flexibility is also important for protocol designers, who have to consider the performance of their designs across a wide range of hardware, and who are becoming increasingly skeptical of designs supporting multiple algorithms [15].

7.3 Compression Function Feed Forward

The output step of the compression function (see §2.2) mixes the first and second halves of the state, and then feeds forward the input CV into the second half. The feed-forward step allows us to use all 16 words of the state as output, doubling the output rate of the extendable output feature (see §2.6). If we exposed all 16 state words without the feed-forward, the compression function would be reversible when the message bytes were known.

When the full output is used, mixing the first and second halves of the state is redundant. However, when the truncated output is used, mixing the halves makes the output use all the state words computed during the last round. The default 32-byte output size is a truncated output, and truncated outputs are also used for chaining values in the interior of the tree.

Putting the feed-forward in the second half of the state allows an implementation that does not support extendable output to skip the feed-forward entirely.

7.4 Keyed Permutation Key Schedule

One of the main historical pain points of implementing BLAKE and BLAKE2 has been the distinct message permutation at each round. This has several effects, including forcing the complete unrolling of the keyed permutation to maximize speed, complicated and error-prone implementation of the permutations in the SIMD case, and complications in the analysis. (E.g., some attacks do not start at round 0 because the permutation is most attack-friendly at some later point.)

To this end, we decided to replace the 10 distinct permutations with a single one, which is applied to the message after each round. The criteria for selecting the permutation were similar to the ones used for BLAKE[3]:

- For 8 rounds, no message word should be input twice at the same point;

[3]One of the requirements, regarding constants, was discarded since it does not apply to BLAKE3.

Table 6: Candidate permutations for BLAKE3's single-permutation key schedule.

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

- For 8 rounds, each message word should appear exactly 4 times in a column step and 4 times in a diagonal step;
- For 8 rounds, each message word should appear exactly 4 times in the first position in G and 4 times in the second.

For 8 rounds, there are exactly 12 permutations that satisfy these requirements, cf. Table 6. Of these, one of them clearly stood out as having the lowest differential probability trail for the keyed permutation after 3 rounds—2^{-21} compared to 2^{-13} for the next best permutations—and thus was the one adopted.

7.5 Chunk Counter

The main job of the compression function's 64-bit counter parameter, t, is to support extendable output. By incrementing t (see §2.6), the caller can extract as much output as needed, without imposing any extra steps on the default-length case.

However, we also use the t parameter during the input phase, to indicate the chunk number (see §2.4). This means that each chunk has a distinct CV with high probability, even if two chunks have the same input bytes. This is not strictly necessary for security, but it discourages a class of dangerous optimizations.

Consider an application that hashes sparse files, which are mostly filled with zeros. The majority of this application's input chunks could be the all-zero chunk. This application might try to compute the CV of the zero chunk only once, and then check each chunk before compressing it to see whether it can use that precomputed CV. This check is cheap, and for this application it could be a big speedup. But this optimization is dangerous, because it is not constant-time. The runtime of the hash function would leak information about its input.

If tricks like this were possible, an unsafe implementation would inevitably find its way into some privacy-sensitive use case. By distinguishing each chunk, BLAKE3 deliberately sabotages these tricks, in the hopes of keeping every implementation constant-time.

7.6 Tree Fanout

BLAKE3 uses a binary tree structure, which is to say a fanout of 2. A binary tree is simpler to implement than a wider tree, and its memory requirement is also a local minimum.

The CV stack algorithm described in §5.1.2 is important for simplifying a BLAKE3 implementation. It also saves space, because the implementation does not need to store size or level metadata along with each CV. Consider how this algorithm would change, if BLAKE3 used a fanout of 4. Instead of "count the trailing 0-bits, and pop that many CVs" it would be "count the trailing 0-0-bit pairs, and pop that many CV triplets." That can still be done with a bit mask, but programmers would need to reason about quaternary numbers to understand what was going on. Other complexities would crop up as well: parent nodes would no longer be a fixed size, and new ambiguities in the tree layout would require a third rule in §2.1 along the lines of "all non-rightmost sibling subtrees have the same number of chunks."

A fanout of 4 would also would also increase the memory requirement of the CV stack. Instead of storing one CV per level of the tree, we would need to store three. The tree would be half as tall, but ultimately the CV stack would be larger by a factor of $3/2$. In general, the size of the CV stack for a fanout F is proportional to $(F - 1)/\log_2(F)$, which increases as F increases.

A different approach to the CV stack is possible, which could save space at larger fanouts. Rather than storing CV bytes directly, the implementation could store an incremental state for each incomplete parent node, similar to the incremental chunk state. An incremental state requires a 32-byte CV and a 64-byte buffer, so at fanout 4 it would be no better than storing 3 CVs directly. (Perhaps the 64-byte buffer could be reduced to 32 bytes in some cases, depending on the details of parent node finalization, and at the cost of more complexity.) But at higher fanouts, the memory requirement would be proportional to $1/\log_2(F)$. At fanout 16 and above, this approach could save space relative to the binary tree.

However, recall from §5.4 that space-constrained implementations of BLAKE3 can reduce the size of the CV stack by restricting their maximum input length. We expect that the majority of space-constrained applications already have strict limits on the size of their inputs. Indeed it is hard to imagine how an application without 2 KB of memory to spare could even get its hands on a gigabyte of input. Realistically, a large fanout would only benefit a small fringe of space-constrained applications, but the added complexity would hurt everyone.

As a special case, we could also have chosen an infinite fanout, a single parent node with an unlimited number of leaves. This approach does not require a CV stack at all. Instead, the implementation only needs to manage the incremental state of the current chunk and the incremental state of the single parent node. Notably, this is the approach taken by KangarooTwelve [8]. It has large benefits for simplicity and space efficiency, but also several downsides. Parent blocks cannot be compressed in parallel, requiring a larger chunk size to keep overhead low (see §7.1). Multi-threading is more complex and generally requires heap allocation, because the input cannot be split in half recursively (see §5.2). And incremental use cases like verified streaming (see §6.4) are no longer practical.

As a totally different approach, we could also have chosen a fixed-interleaved tree layout similar to that of BLAKE2bp and BLAKE2sp. For example, BLAKE2sp divides its input into 8 interleaved pieces, with blocks 0, 8, 16... forming the first piece, blocks 1, 9, 17... forming the second piece, and so on. The 8 resulting CVs are then concatenated into a single parent node. The upsides of this approach are that full parallelism can begin at relatively short input lengths, 512 bytes in the case of BLAKE2bp and BLAKE2sp, and that the memory requirement is lower than that of the binary tree. The downsides of this approach are that the degree of parallelism is fixed, and that performance is poor for shorter inputs (see Figures 3 and 5).

7.7 Domain Flags

The `ROOT` and `CHUNK_START` domain flags are strictly necessary for security. The `ROOT` flag separates the root node from all other nodes, and it separates the final block from all other blocks in a root chunk, both of which prevent length extension. The `CHUNK_START` flag separates chunks from parent nodes, and it separates an IV provided by the caller (the key in the keyed hashing mode) from block CVs computed within a chunk, both of which prevent collisions.

The `PARENT` and `CHUNK_END` domain flags are not strictly necessary, but we include them to be conservative. As with `CHUNK_START`, the `PARENT` flag prevents a collision when the caller-provided IV bytes match a block CV extracted from some other input. However, the parent's message bytes will also be different from the block's with high probability in that case. The benefit of the `PARENT` flag is rather that we get separation without needing to reason about the message bytes.

The `CHUNK_END` flag prevents length extensions for all chunks, beyond just a root chunk. However, other chunk CVs are not usually published. In incremental use cases like verified streaming (see §6.4), chunk CVs may be published, but only when the input is also published. In a keyed incremental use case, the keyed parent nodes also prevent an attacker from making any use of length-extended keyed chunk CVs. Nonetheless, we include the `CHUNK_END` flag, because letting an attacker length-extend chunk CVs is an unnecessary risk that might impact unknown future applications.

7.8 Key Derivation from Reused Key Material

As described in §6.2, the `derive_key` mode is intended to be safe to use even with keys that are already in use with other algorithms. It is impossible to guarantee such a property in general, however. As noted in that section, some algorithms explicitly forbid key reuse, even within the same algorithm. In the case of a one-time pad, if the message happens to be known or predictable, some or all of the key is revealed to an eavesdropper. No key derivation function can extract useful security from key material that is not secret.

There are some algorithms that impose a one-time-use restriction on their key, but that still keep the key secret as long as that restriction is followed. The `crypto_onetimeauth` function in the NaCl and libsodium libraries is one such example. It might be safe to use the same key with `crypto_onetimeauth` (once) and with `derive_key` (any number of times). However, precisely because of its one-time-use restriction, `crypto_onetimeauth` is almost always used with derived subkeys anyway. Furthermore, this sort of algorithm-specific reasoning is exactly what `derive_key` is designed to avoid. It is simpler and safer to forbid `derive_key` from sharing key material with this entire class of one-time-use algorithms.

Another limitation on key reuse is applications that violate the security requirement of the `derive_key` context string, namely that it should be hardcoded, globally unique, and application-specific. If one component of an application feeds arbitrary user input into the context string, it is not safe for any other component to share key material with it. The context string no longer provides domain separation. This is the basic problem of sharing key material, and the reason that deriving separate subkeys is preferable wherever possible. When two different components share a key, each has to assume that the other will not violate the security requirements of `derive_key`, just as each has to assume that the other will not leak the key.

Another more theoretical limitation for key material reuse could be future algorithms that are internally identical to BLAKE3, but with different domain-separation conventions. A modified algorithm might sabotage domain separation entirely, for example, by exposing the d parameter of the compression function to arbitrary user input. We will never publish such an algorithm. The common practice when designing a closely related algorithm is to change the IV constants, so that the new output is independent. This was the approach used to derive SHA-512/256 from SHA-512. The same approach could work for a hypothetical derivative of BLAKE3 by changing the $\mathrm{IV}_0 \ldots \mathrm{IV}_3$ constants, which are used for compression function setup in all modes.

Acknowledgments

The development of BLAKE3 was sponsored by Electric Coin Company and Teserakt. 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. `https://eprint.iacr.org/2019/1492`.

[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] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. KangarooTwelve: Fast hashing based on Keccak-p. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2018. doi:10.1007/978-3-319-93387-0_21.

[9] 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.

[10] 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.

[11] Wonseok Choi, ByeongHak Lee, and Jooyoung Lee. Indifferentiability of truncated random permutations. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 175–195. Springer, 2019. doi:10.1007/978-3-030-34578-5_7.

[12] Joan Daemen, Tony Dusenge, and Gilles Van Assche. Sufficient conditions for sound hashing using a truncated permutation. *IACR Cryptology ePrint Archive*, 2011:459, 2011. URL: http://eprint.iacr.org/2011/459.

[13] 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. URL: https://www.cs.ru.nl/~bmennink/pubs/18fsenew.pdf.

[14] Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 104–121. Springer, 2009. doi:10.1007/978-3-642-03317-9_7.

[15] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2017*, 2015. URL: https://www.wireguard.com/papers/wireguard.pdf.

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

[17] 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.

[18] 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.

[19] 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.

[20] Dmitry Khovratovich, Ivica Nikolic, Josef Pieprzyk, Przemyslaw Sokolowski, and Ron Steinfeld. Rotational cryptanalysis of ARX revisited. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 519–536. Springer, 2015. doi:10.1007/978-3-662-48116-5_25.

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

[22] 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.

[23] Nicky Mouha and Bart Preneel. Towards finding optimal differential characteristics for ARX: Application to Salsa20. Cryptology ePrint Archive, Report 2013/328, 2013. https://eprint.iacr.org/2013/328.

[24] Trevor Perrin. Stateful hash objects: API and constructions, 2018. URL: https://github.com/noiseprotocol/sho_spec.

[25] 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.

[26] 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.