BLAKE3: one function, fast everywhere

November 17, 2019 — REQUEST FOR COMMENTS, NOT A FINAL VERSION

https://blake3.io

AUTHORS

Abstract. We present the cryptographic hash function BLAKE3, an improved version of BLAKE2 and of its predecessor, the SHA-3 finalist BLAKE. BLAKE3 supports high degrees of parallelism, using an internal tree structure that scales up to any number of SIMD lanes and CPU cores. On Intel Kaby Lake, peak single-threaded throughput is 3x that of BLAKE2b, 4x that of SHA-2, and 8x that of SHA-3, and it can scale further using multiple threads. At the same time, BLAKE3 is efficient on smaller architectures and microcontrollers. Throughput on ARM Cortex-M0 is 30% higher than BLAKE2s or SHA-256, and 3x that of BLAKE2b, SHA-512, or SHA-3. Unlike BLAKE2 and SHA-2, which have incompatible variants optimized for different platforms, BLAKE3 is a single hash function, designed to be consistently fast in software across a wide variety of platforms and use cases.

1 Introduction

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

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

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

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

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

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

The biggest changes from BLAKE2 to BLAKE3 are:

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

2 Specification

2.1 Tree Structure

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

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

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

The compression function is used to derive an 8-word 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 BLAKE3 hash of the input. BLAKE3 supports input of any byte length $0 \leq \ell < 2^{64}$.

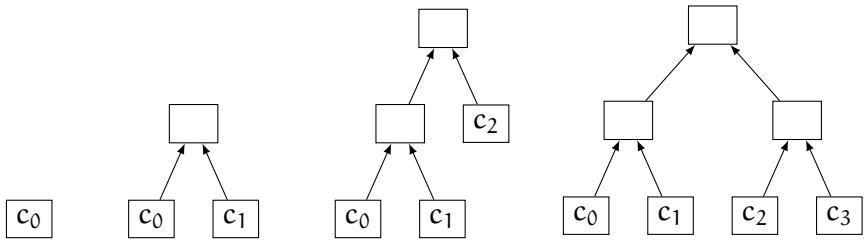

Figure 1: Example tree structures, from 1 to 4 chunks.

2.2 Compression Function

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

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

The compression function initializes its 16-word internal state $v_0 \dots v_{15}$ as follows. The $IV_0 \dots IV_7$ constants are the same as in BLAKE2s, and they are reproduced in appendix A. $\oplus$ denotes bitwise exclusive-or:

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

The compression function applies the round function to the state and the message block 7 times, each time with a different message schedule. The round function and the message schedule for each round are the same as in BLAKE2s. They are reproduced in appendix B.

After 7 rounds of compression, the new chaining value $h'_0 \dots h'_7$ is defined as:

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

The domain separation flags used for d are:

CHUNK_START	1
CHUNK_END	2
PARENT	4
ROOT	8
KEYED_HASH	16
DERIVE_KEY	32

The value of d is the bitwise-or of all the flags set for a given compression. Their usage is described below.

2.3 Modes

BLAKE3 defines three domain-separated modes: **hash**, **keyed_hash**, and **derive_key**. The modes differ from each other in their key words $k_0 \dots k_7$ and in the additional flags they set for every call to the compression function. In the **hash** mode, $k_0 \dots k_7$ are the constants $IV_0 \dots IV_7$, and no additional flags are set. In the **keyed_hash** mode, $k_0 \dots k_7$ are parsed in little-endian order from the 32-byte key given by the caller, and the KEYED_HASH flag is set for every compression. In the **derive_key** mode, the key is again given by the caller, and the DERIVE_KEY flag is set for every compression.

2.4 Chunk Chaining Values

Processing a chunk is structurally similar to processing input in BLAKE2s or BLAKE2b. Each chunk of up to 2048 bytes is split into blocks of up to 64 bytes. The last block of the last chunk may be shorter, but not empty, unless the entire input is empty. The short block, if any, 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 the key words $k_0 \dots k_7$. The input chaining value for subsequent blocks in each chunk is the output of the compression function for the previous block. The offset t for each block is the starting byte index of its chunk, 0 for all blocks in the first chunk, 2048 for all blocks in the second chunk, and so on. The block length b is the number of input bytes in each block, 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, and the last block of each chunk sets the CHUNK_END flag. If a chunk contains only one block, that block sets both CHUNK_START and CHUNK_END. If a chunk is the root of its tree, the last block of that chunk also sets the ROOT flag. The output of the compression function for the last block in a chunk is the chaining value of that chunk.

2.5 Parent Node Chaining Values

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

2.6 Extendable Output

The default output of BLAKE3 is 32 bytes, but BLAKE3 can also produce extended output of any byte length $\ell < 2^{64}$. 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 offset t. The results of these repeated root compressions are then concatenated into an extended output.

When building an extended output, BLAKE3 uses an extended version of the compression function that differs from the regular compression function in its final step, returning 16 words instead of 8:

$$
\begin{aligned}
h'_0 &\leftarrow v_0 \oplus v_8 & \qquad h'_8 &\leftarrow v_8 \oplus h_0 \\
h'_1 &\leftarrow v_1 \oplus v_9 & \qquad h'_9 &\leftarrow v_9 \oplus h_1 \\
h'_2 &\leftarrow v_2 \oplus v_{10} & \qquad h'_{10} &\leftarrow v_{10} \oplus h_2 \\
h'_3 &\leftarrow v_3 \oplus v_{11} & \qquad h'_{11} &\leftarrow v_{11} \oplus h_3 \\
h'_4 &\leftarrow v_4 \oplus v_{12} & \qquad h'_{12} &\leftarrow v_{12} \oplus h_4 \\
h'_5 &\leftarrow v_5 \oplus v_{13} & \qquad h'_{13} &\leftarrow v_{13} \oplus h_5 \\
h'_6 &\leftarrow v_6 \oplus v_{14} & \qquad h'_{14} &\leftarrow v_{14} \oplus h_6 \\
h'_7 &\leftarrow v_7 \oplus v_{15} & \qquad h'_{15} &\leftarrow v_{15} \oplus h_7
\end{aligned}
$$

Note that the first 8 words of the extended compression output are the same as the regular compression output. The last 8 words are the input chaining value fed forward into the last 8 words of the internal state. Each 16-word output is encoded as 64 bytes in little-endian order.

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

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

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

3 Performance

It's fast :)

4 Security

It's safe :)

5 Memory Requirement

It's big :(

6 Implementation

6.1 SIMD

How to do efficient parallel chunk hashing.

6.2 Incremental Hashing

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

7 Applications

7.1 Message Authentication Codes

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

7.2 Key Derivation

BLAKE3 has explicit support for a key derivation mode, **derive_key**. 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 input key material is not already a 256-bit key, it should first be converted into 256 bits using the default output of the BLAKE3 **hash** mode.

The **derive_key** mode is intended to replace the BLAKE2 personalization parameter for its most common use cases. Like **keyed_hash**, it 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.

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.

7.3 Streaming Verification

Bao, basically.

7.4 Incremental Update

Giant disk images.

8 Rationales

References

[1] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. *IACR Cryptology ePrint Archive*, 2013:322, 2013. URL: http://eprint.iacr.org/2013/322.

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

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

[4] 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.

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

[These references are placeholders. TODO.]

Appendix A IV Constants

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

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

Appendix B Round Function

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

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

That is, a round applies a G function to each of the columns in parallel, and then to each of the diagonals in parallel. $G_i(a, b, c, d)$ is defined as follows. $\ggg$ 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

Appendix C Rust Reference Implementation

```rust
const OUT_LEN: usize = 32;
const KEY_LEN: usize = 32;
const BLOCK_LEN: usize = 64;
const CHUNK_LEN: usize = 2048;
const ROUNDS: usize = 7;

const CHUNK_START: u8 = 1 << 0;
const CHUNK_END: u8 = 1 << 1;
const PARENT: u8 = 1 << 2;
const ROOT: u8 = 1 << 3;
const KEYED_HASH: u8 = 1 << 4;
```

```rust
const DERIVE_KEY: u8 = 1 << 5;

const IV: [u32; 8] = [
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A, 0x510E527F, 0x9B05688C,
    0x1F83D9AB, 0x5BE0CD19,
];

const MSG_SCHEDULE: [[usize; 16]; ROUNDS] = [
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
    [14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3],
    [11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4],
    [7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8],
    [9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13],
    [2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9],
    [12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11],
];

fn words_from_litte_endian_bytes(bytes: &[u8], words: &mut [u32]) {
    for (b, w) in bytes.chunks_exact(4).zip(words.iter_mut()) {
        *w = u32::from_le_bytes(core::convert::TryInto::try_into(b).unwrap());
    }
}

fn little_endian_bytes_from_words(words: &[u32], bytes: &mut [u8]) {
    for (w, b) in words.iter().zip(bytes.chunks_exact_mut(4)) {
        b.copy_from_slice(&w.to_le_bytes());
    }
}

// The mixing function, G, which mixes either a column or a diagonal.
fn g(state: &mut [u32; 16], indices: [usize; 4], mx: u32, my: u32) {
    let [a, b, c, d] = indices;
    state[a] = state[a].wrapping_add(state[b]).wrapping_add(mx);
    state[d] = (state[d] ^ state[a]).rotate_right(16);
    state[c] = state[c].wrapping_add(state[d]);
    state[b] = (state[b] ^ state[c]).rotate_right(12);
    state[a] = state[a].wrapping_add(state[b]).wrapping_add(my);
    state[d] = (state[d] ^ state[a]).rotate_right(8);
    state[c] = state[c].wrapping_add(state[d]);
    state[b] = (state[b] ^ state[c]).rotate_right(7);
}

fn round(state: &mut [u32; 16], m: &[u32; 16], schedule: &[usize; 16]) {
    // Mix the columns.
    g(state, [0, 4, 8, 12], m[schedule[0]], m[schedule[1]]);
    g(state, [1, 5, 9, 13], m[schedule[2]], m[schedule[3]]);
    g(state, [2, 6, 10, 14], m[schedule[4]], m[schedule[5]]);
```

```
    g(state, [3, 7, 11, 15], m[schedule[6]], m[schedule[7]]);
    // Mix the diagonals.
    g(state, [0, 5, 10, 15], m[schedule[8]], m[schedule[9]]);
    g(state, [1, 6, 11, 12], m[schedule[10]], m[schedule[11]]);
    g(state, [2, 7, 8, 13], m[schedule[12]], m[schedule[13]]);
    g(state, [3, 4, 9, 14], m[schedule[14]], m[schedule[15]]);
}

fn compress_inner(
    chaining_value: &[u32; 8],
    block_words: &[u32; 16],
    offset: u64,
    block_len: u8,
    flags: u8,
) -> [u32; 16] {
    let mut state = [
        chaining_value[0],
        chaining_value[1],
        chaining_value[2],
        chaining_value[3],
        chaining_value[4],
        chaining_value[5],
        chaining_value[6],
        chaining_value[7],
        IV[0],
        IV[1],
        IV[2],
        IV[3],
        IV[4] ^ (offset as u32),
        IV[5] ^ ((offset >> 32) as u32),
        IV[6] ^ block_len as u32,
        IV[7] ^ flags as u32,
    ];
    for r in 0..ROUNDS {
        round(&mut state, &block_words, &MSG_SCHEDULE[r]);
    }
    state
}

// The standard compression function updates an 8-word chaining value in place.
fn compress(
    chaining_value: &mut [u32; 8],
    block_words: &[u32; 16],
    offset: u64,
    block_len: u8,
    flags: u8,
) {
```

```
    let state =
        compress_inner(chaining_value, block_words, offset, block_len, flags);
    for i in 0..8 {
        chaining_value[i] = state[i] ^ state[i + 8];
    }
}

// The extended compression function returns a new 16-word extended output.
// Note that the first 8 words of output are the same as with compress().
// Implementations that do not support extendable output can omit this.
fn compress_extended(
    chaining_value: &[u32; 8],
    block_words: &[u32; 16],
    offset: u64,
    block_len: u8,
    flags: u8,
) -> [u32; 16] {
    let mut state =
        compress_inner(chaining_value, block_words, offset, block_len, flags);
    for i in 0..8 {
        state[i] ^= state[i + 8];
        state[i + 8] ^= chaining_value[i];
    }
    state
}

// The output of a chunk or parent node, which could be an 8-word chaining
// value or any number of root output bytes.
struct Output {
    input_chaining_value: [u32; 8],
    block_words: [u32; 16],
    offset: u64,
    block_len: u8,
    flags: u8,
}

impl Output {
    fn chaining_value(&self) -> [u32; 8] {
        let mut cv = self.input_chaining_value;
        compress(
            &mut cv,
            &self.block_words,
            self.offset,
            self.block_len,
            self.flags,
        );
        cv
```

```
    }

    fn root_output(&self, out_slice: &mut [u8]) {
        let mut offset = self.offset;
        for out_block in out_slice.chunks_mut(2 * OUT_LEN) {
            let words = compress_extended(
                &self.input_chaining_value,
                &self.block_words,
                offset,
                self.block_len,
                self.flags | ROOT,
            );
            little_endian_bytes_from_words(&words, out_block);
            offset += 2 * OUT_LEN as u64;
        }
    }
}

struct ChunkState {
    chaining_value: [u32; 8],
    offset: u64,
    block: [u8; BLOCK_LEN],
    block_len: u8,
    blocks_compressed: u8,
    flags: u8,
}

impl ChunkState {
    fn new(key: &[u32; 8], offset: u64, flags: u8) -> Self {
        Self {
            chaining_value: *key,
            offset,
            block: [0; BLOCK_LEN],
            block_len: 0,
            blocks_compressed: 0,
            flags,
        }
    }

    fn len(&self) -> usize {
        BLOCK_LEN * self.blocks_compressed as usize + self.block_len as usize
    }

    fn start_flag(&self) -> u8 {
        if self.blocks_compressed == 0 {
            CHUNK_START
        } else {
```

```
            0
        }
    }

    fn update(&mut self, mut input: &[u8]) {
        while !input.is_empty() {
            if self.block_len as usize == BLOCK_LEN {
                let mut block_words = [0; 16];
                words_from_litte_endian_bytes(&self.block, &mut block_words);
                let block_flags = self.start_flag() | self.flags;
                compress(
                    &mut self.chaining_value,
                    &block_words,
                    self.offset,
                    BLOCK_LEN as u8,
                    block_flags,
                );
                self.blocks_compressed += 1;
                self.block = [0; BLOCK_LEN];
                self.block_len = 0;
            }

            let want = BLOCK_LEN - self.block_len as usize;
            let take = core::cmp::min(want, input.len());
            self.block[self.block_len as usize..][..take]
                .copy_from_slice(&input[..take]);
            self.block_len += take as u8;
            input = &input[take..];
        }
    }

    fn output(&self) -> Output {
        let mut block_words = [0; 16];
        words_from_litte_endian_bytes(&self.block, &mut block_words);
        let block_flags = self.flags | self.start_flag() | CHUNK_END;
        Output {
            input_chaining_value: self.chaining_value,
            block_words,
            block_len: self.block_len,
            offset: self.offset,
            flags: block_flags,
        }
    }
}

fn parent_output(
    left_child_cv: &[u32; 8],
```

```
    right_child_cv: &[u32; 8],
    key: &[u32; 8],
    flags: u8,
) -> Output {
    let mut block_words = [0; 16];
    block_words[..8].copy_from_slice(left_child_cv);
    block_words[8..].copy_from_slice(right_child_cv);
    Output {
        input_chaining_value: *key,
        block_words,
        offset: 0,                    // Always 0 for parent nodes.
        block_len: BLOCK_LEN as u8, // Always BLOCK_LEN (64) for parent nodes.
        flags: PARENT | flags,
    }
}

/// An incremental hasher that can accept any number of writes.
pub struct Hasher {
    chunk_state: ChunkState,
    key: [u32; 8],
    subtree_stack: [[u32; 8]; 53], // Space for 53 subtree chaining values:
    num_subtrees: u8,              // 2^53 * CHUNK_LEN = 2^64
}

impl Hasher {
    fn new_internal(key: &[u32; 8], flags: u8) -> Self {
        Self {
            chunk_state: ChunkState::new(key, 0, flags),
            key: *key,
            subtree_stack: [[0; 8]; 53],
            num_subtrees: 0,
        }
    }

    /// Construct a new `Hasher` for the default **hash** mode.
    pub fn new() -> Self {
        Self::new_internal(&IV, 0)
    }

    /// Construct a new `Hasher` for the **keyed_hash** mode.
    pub fn new_keyed(key: &[u8; KEY_LEN]) -> Self {
        let mut key_words = [0; 8];
        words_from_litte_endian_bytes(key, &mut key_words);
        Self::new_internal(&key_words, KEYED_HASH)
    }

    /// Construct a new `Hasher` for the **derive_key** mode.
```

```
pub fn new_derive_key(key: &[u8; KEY_LEN]) -> Self {
    let mut key_words = [0; 8];
    words_from_litte_endian_bytes(key, &mut key_words);
    Self::new_internal(&key_words, DERIVE_KEY)
}

// Take two subtree hashes off the end of the stack, compress them into a
// parent hash, and put that hash back on the stack.
fn merge_two_subtrees(&mut self) {
    let left_child = &self.subtree_stack[self.num_subtrees as usize - 2];
    let right_child = &self.subtree_stack[self.num_subtrees as usize - 1];
    let parent_hash = parent_output(
        left_child,
        right_child,
        &self.key,
        self.chunk_state.flags,
    )
    .chaining_value();
    self.subtree_stack[self.num_subtrees as usize - 2] = parent_hash;
    self.num_subtrees -= 1;
}

fn push_chunk_chaining_value(&mut self, cv: &[u32; 8], total_bytes: u64) {
    self.subtree_stack[self.num_subtrees as usize] = *cv;
    self.num_subtrees += 1;
    // After pushing the new chunk hash onto the subtree stack, hash as
    // many parent nodes as we can. The number of 1 bits in the total
    // number of chunks so far is the same as the number of subtrees that
    // should remain in the stack.
    let total_chunks = total_bytes / CHUNK_LEN as u64;
    while self.num_subtrees as usize > total_chunks.count_ones() as usize {
        self.merge_two_subtrees();
    }
}

/// Add input to the hash state. This can be called any number of times.
pub fn update(&mut self, mut input: &[u8]) {
    while !input.is_empty() {
        if self.chunk_state.len() == CHUNK_LEN {
            let chunk_cv = self.chunk_state.output().chaining_value();
            let new_chunk_offset =
                self.chunk_state.offset + CHUNK_LEN as u64;
            self.push_chunk_chaining_value(&chunk_cv, new_chunk_offset);
            self.chunk_state = ChunkState::new(
                &self.key,
                new_chunk_offset,
                self.chunk_state.flags,
```

```
                );
            }

            let want = CHUNK_LEN - self.chunk_state.len();
            let take = core::cmp::min(want, input.len());
            self.chunk_state.update(&input[..take]);
            input = &input[take..];
        }
    }

    /// Finalize the hash and return the default 32-byte output.
    pub fn finalize(&self) -> [u8; OUT_LEN] {
        let mut bytes = [0; OUT_LEN];
        self.finalize_extended(&mut bytes);
        bytes
    }

    /// Finalize the hash and write any number of output bytes.
    pub fn finalize_extended(&self, out_slice: &mut [u8]) {
        // If the subtree stack is empty, then the current chunk is the root.
        if self.num_subtrees == 0 {
            self.chunk_state.output().root_output(out_slice);
            return;
        }

        // Otherwise, finalize the current chunk, and then merge all the
        // subtrees along the right edge of the tree.
        let mut right_child = self.chunk_state.output().chaining_value();
        let mut subtrees_remaining = self.num_subtrees as usize;
        loop {
            let left_child = &self.subtree_stack[subtrees_remaining - 1];
            let output = parent_output(
                left_child,
                &right_child,
                &self.key,
                self.chunk_state.flags,
            );
            if subtrees_remaining == 1 {
                output.root_output(out_slice);
                return;
            }
            right_child = output.chaining_value();
            subtrees_remaining -= 1;
        }
    }
}
```