SHA-3 proposal BLAKE

Jean-Philippe Aumasson* Luca Henzen† Willi Meier‡ Raphael C.-W. Phan§

version 1.2

Abstract

BLAKE is our proposal for SHA-3. BLAKE entirely relies on previously analyzed components: it uses the HAIFA iteration mode and builds its compression function on the ChaCha core function. BLAKE resists generic second-preimage attacks, length extension, and side-channel attacks. Theoretical and empirical security guarantees are given, against structural and differential attacks. BLAKE hashes on a Core 2 Duo at 12 cycles/byte, and on a 8-bit PIC microcontroller at 400 cycles/byte. In hardware BLAKE can be implemented in less than 9900 gates, and reaches a throughput of 6 Gbps.

*FHNW, Windisch, Switzerland, `jeanphilippe.aumasson@gmail.com`
†ETHZ, Zürich, Switzerland, `henzen@iis.ee.ethz.ch`
‡FHNW, Windisch, Switzerland, `willi.meier@fhnw.ch`
§Loughborough University, UK, `r.phan@lboro.ac.uk`

Contents

1 Introduction

In 1993, NIST published the first Secure Hash Standard SHA-0, which two years later was superseded by SHA-1 to fix a flaw in the message expansion. SHA-1 was still deemed secure by the end of the millenium, when researchers' attention turned to block ciphers through the AES competition. Shortly after that, an avalanche of results on hash functions culminated with collision attacks for MD5 and SHA-1, while in the meantime NIST had introduced the SHA-2 family, unbroken until now. But attacks on SHA-1 arguably raise doubts on the long-term security of SHA-2, because of its very similar structure. In response NIST announced the SHA-3 program, calling for proposals for a hash function that will augment the SHA-2 standard. Many recent results illustrate the obsolescence of designs based on MD5 and SHA-1: only in the first semester of 2008, were published new collision attacks for (reduced) SHA-256 [33] and the first preimage attacks for (reduced) MD5 [4], SHA-0, and SHA-1 [21].

BLAKE is our candidate for SHA-3. It meets all the criteria set by NIST, offers theoretical and empirical security guarantees, and performs well from high-end PC's to light hardware. We did not reinvent the wheel; BLAKE is built on previously studied components, chosen for their complementarity. The heritage of BLAKE is threefold:

- its **iteration mode** is HAIFA, an improved version of the Merkle-Damgård paradigm proposed by Biham and Dunkelman. It provides resistance to long-message second preimage attacks, and explicitly handles hashing with a salt.
- its **internal structure** is the local wide-pipe, which we already used with the LAKE hash function. It makes local collisions impossible in the BLAKE hash functions, a result that doesn't rely on any intractability assumption.
- its **compression algorithm** is a modified version of Bernstein's stream cipher ChaCha, whose security has been intensively analyzed and performance is excellent, and which is strongly parallelizable.

The iteration mode HAIFA would significantly benefit to the new hash standard, for it provides randomized hashing and structural resistance to second-preimage attacks. The LAKE local wide-pipe structure is a straightforward way to give strong security guarantees against collision attacks. Finally, the choice of borrowing from the stream cipher ChaCha (after agreement of its author) comes from our experience in cryptanalysis of Salsa20 and ChaCha [3], when we got convinced of their remarkable combination of simplicity and security.

Content of this document: The present chapter contains design principles, a short description of BLAKE, and security claims. Chapter 2 gives a complete specification of the BLAKE hash functions. Chapter 3 reports performance in FPGA, ASIC, 8-bit microcontroller, and 32- and 64-bit processor. Chapter 4 explains how to use BLAKE, detailing construction of HMAC, UMAC, and PRF ensembles. Chapter 5 gives elements of analysis, including attacks on simplified versions. We conclude with acknowledgments, references, and appendices containing source code and intermediate values.

1.1 Design principles

The BLAKE hash functions were designed to meet all NIST criteria for SHA-3, including:

- message digests of 224, 256, 384, and 512 bits
- same parameter sizes as SHA-2
- one-pass streaming mode
- maximum message length of at least $2^{64} - 1$ bits

In addition, we imposed BLAKE to:

- explicitly handle hashing with a salt
- be parallelizable
- allow performance trade-offs
- be suitable for lightweight environments

We briefly justify these choices: First, a built-in salt simplifies a lot of things; it provides an interface for an extra input, avoids insecure homemade modes, and encourages the use of randomized hashing. Parallelism is a big advantage for hardware implementations, which can also be exploited by certain large microprocessors. In addition, BLAKE allows a trade-off throughput/area to adapt the implementation to the hardware available.

Oppositely, we excluded the following goals:

- have a reduction to a supposedly hard problem
- have homomorphic or incremental properties
- have a scalable design
- have a specification for variable length hashing

We justify these choices: The relative unsuccess of provably secure hash functions stresses the limitations of the approach: though of theoretical interest, such designs tend to be inefficient, and their highly structured constructions expose them to attacks with respect to notions other than the proved one. The few advantages of homomorphic and incremental hash functions are not worth their cost; more importantly, these properties are undesirable in many applications. Scalability of the design to various parameter sizes has no real advantage in practice, and the security of scalable designs is difficult to assess. Finally, we deemed unnecessary to complicate the function with variable-length features; in practice users can just truncate the hash values for shorter hashes, and there is no demand for hash values of more than 512 bits.

To summarize, we made our candidate as simple as possible, and combined well-known and trustable building blocks so that BLAKE already looks familiar to cryptanalysts. We avoided any show-off feature, and just provide what users really need or will need in a close future (like hashing with a salt). It was essential for us to build on previous knowledge—be it about security or implementation—in order to adapt our proposal to the low resources available for analyzing the SHA-3 candidates.

1.2 BLAKE in a nutshell

BLAKE is a family of four hash functions: BLAKE-28, BLAKE-32, BLAKE-48, and BLAKE-64 (see Table 1.1). As with SHA-2, we have a 32-bit version (BLAKE-32) and a 64-bit one (BLAKE-64), from which other instances are derived using different initial values, different padding, and truncated output.

Algorithm	Word	Message	Block	Digest	Salt
BLAKE-28	32	$<2^{64}$	512	224	128
BLAKE-32	32	$<2^{64}$	512	256	128
BLAKE-48	64	$<2^{128}$	1024	384	256
BLAKE-64	64	$<2^{128}$	1024	512	256

Table 1.1: Properties of the BLAKE hash functions (sizes in bits).

The BLAKE hash functions follow the HAIFA iteration mode [10]: the compression function depends on a *salt*[1] and the *number of bits hashed so far* (counter), to compress each message block with a distinct function. The structure of BLAKE's compression function is inherited from LAKE [5] (see Fig. 1.1): a large inner state is initialized from the initial value, the salt, and the counter. Then it is injectively updated by message-dependent *rounds*, and it is finally compressed to return the next chain value. This strategy was called *local wide-pipe* in [5], and is inspired by the wide-pipe iteration mode [30].

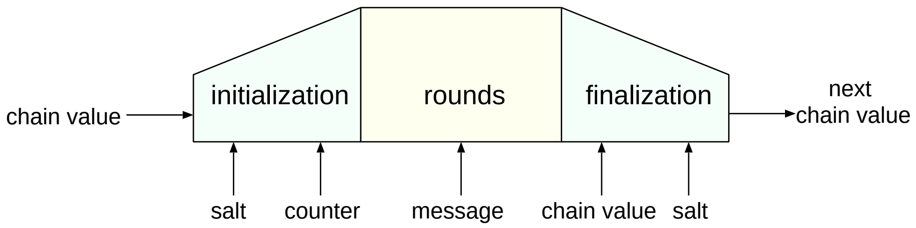

Figure 1.1: The local wide-pipe construction of BLAKE's compression function.

The inner state of the compression function is represented as a 4×4 matrix of words. A round of BLAKE-32 is a modified "double-round" of the stream cipher ChaCha: first, all four columns are updated independently, and thereafter four disjoint diagonals. In the update of each column or diagonal, two message words are input according to a round-dependent permutation. Each round is parametrized by distinct constants to minimize self-similarity. After the sequence of rounds, the state is reduced to half its length with feedforward of the initial value and the salt.

An implementation of BLAKE requires low resources, and is fast in both software and hardware environments. BLAKE can be implemented in hardware in less than 9 900 gates, and reach a throughput of 6 Gbps. In a 8-bit PIC microcontroller, BLAKE hashes at 400 cycles/byte; on our 32-bit Celeron at 22 cycles/byte, and on our 64-bit Core 2 Duo at 12 cycles/byte.

[1] A value that parametrizes the function, and can be either public or secret.

1.3 Expected strength

For all BLAKE hash functions, there should be no attack significantly more efficient than standard bruteforce methods for

- finding collisions, with same or distinct salt
- finding (second) preimages, with arbitrary salt

BLAKE should also be secure for randomized hashing, with respect to the experiment described by NIST in [36, 4.A.ii]. It should be impossible to distinguish a BLAKE instance with an unknown salt (that is, uniformly chosen at random) from a PRF, given blackbox access to the function; more precisely, it shouldn't cost significantly less than $2^{|s|}$ queries to the box, where $|s|$ is the bit length of the salt. BLAKE should have no property that makes its use significantly less secure than an ideal function for any concrete application. (These claims concern the proposed functions with the *recommended* number of rounds, not reduced or modified versions.)

1.4 Advantages and limitations

We summarize the advantages and limitations of BLAKE:

Advantages

Design

- simplicity of the algorithm
- interface for hashing with a salt

Performance

- fast in both software and hardware
- parallelism and throughput/area trade-off for hardware implementation
- simple speed/confidence trade-off with the tunable number of rounds

Security

- based on an intensively analyzed component (ChaCha)
- resistant to generic second-preimage attacks
- resistant to side-channel attacks
- resistant to length-extension

Limitations

- message length limited to respectively 2^{64} and 2^{128} for BLAKE-32 and BLAKE-64
- resistance to Joux's multicollisions similar to that of SHA-2
- fixed-points found in less time than for an ideal function (but not efficiently)

1.5 Notations

Hexadecimal numbers are written in `typewriter` style (for example $\texttt{F0} = 240$). A *word* is either a 32-bit or a 64-bit string, depending on the context. We use the same conventions of big-endianness as NIST does in the SHA-2 specification [34, §3]. In particular, we use (unsigned) big-endian representation for expressing integers, and, e.g. converting data streams into word arrays. Table 1.2 summarizes the basic operations used.

Symbol	Meaning
$\leftarrow$	variable assignment
$+$	addition modulo 2^{32} or (modulo 2^{64})
$\oplus$	Boolean exclusive OR (XOR)
$\ggg k$	rotation of k bits towards less significant bits
$\lll k$	rotation of k bits towards more significant bits
$\langle \ell \rangle_k$	encoding of the integer ℓ over k bits

Table 1.2: Operations symbols used in this document.

If p is a bit string, we view it as a sequence of words and p_i denotes its i^{th} word component; thus $p = p^0 \| p^1 \| \ldots$ For a message m, m^i denotes its i^{th} 16-word block, thus m^i_j is the j^{th} word of the i^{th} block of m. Indices start from zero, for example a N-block message m is decomposed as $m = m^0 m^1 \ldots m^{N-1}$, and the block m^0 is composed of words $m^0_0, m^0_1, m^0_2, \ldots, m^0_{15}$,

The adjective *random* here means uniformly random with respect to the relevant probability space. For example a “random salt” of BLAKE-32 is a random variable uniformly distributed over $\{0,1\}^{128}$, and may also mean “uniformly chosen at random”. The *initial value* is written IV; intermediate hash values in the iterated hash are called *chain values*, and the last one is the *hash value*, or just *hash*.

2 Specification

This chapter defines the hash functions BLAKE-32, BLAKE-64, BLAKE-28, and BLAKE-48.

2.1 BLAKE-32

The hash function BLAKE-32 operates on 32-bit words and returns a 32-byte hash value. This section defines BLAKE-32, going from its constant parameters to its compression function, then to its iteration mode.

2.1.1 Constants

BLAKE-32 starts hashing from the same initial value as SHA-256:

$$\begin{array}{ll}
\mathsf{IV}_0 = \mathtt{6A09E667} & \mathsf{IV}_1 = \mathtt{BB67AE85} \\
\mathsf{IV}_2 = \mathtt{3C6EF372} & \mathsf{IV}_3 = \mathtt{A54FF53A} \\
\mathsf{IV}_4 = \mathtt{510E527F} & \mathsf{IV}_5 = \mathtt{9B05688C} \\
\mathsf{IV}_6 = \mathtt{1F83D9AB} & \mathsf{IV}_7 = \mathtt{5BE0CD19}
\end{array}$$

BLAKE-32 uses 16 constants[1]

$$\begin{array}{ll}
c_0 = \mathtt{243F6A88} & c_1 = \mathtt{85A308D3} \\
c_2 = \mathtt{13198A2E} & c_3 = \mathtt{03707344} \\
c_4 = \mathtt{A4093822} & c_5 = \mathtt{299F31D0} \\
c_6 = \mathtt{082EFA98} & c_7 = \mathtt{EC4E6C89} \\
c_8 = \mathtt{452821E6} & c_9 = \mathtt{38D01377} \\
c_{10} = \mathtt{BE5466CF} & c_{11} = \mathtt{34E90C6C} \\
c_{12} = \mathtt{C0AC29B7} & c_{13} = \mathtt{C97C50DD} \\
c_{14} = \mathtt{3F84D5B5} & c_{15} = \mathtt{B5470917}
\end{array}$$

Ten permutations of $\{0,\ldots,15\}$ are used by all BLAKE functions, defined in Table 2.1.

2.1.2 Compression function

The compression function of BLAKE-32 takes as input four values:

- a chain value $h = h_0, \ldots, h_7$
- a message block $m = m_0, \ldots, m_{15}$
- a salt $s = s_0, \ldots, s_3$
- a counter $t = t_0, t_1$

[1] First digits of π.

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

Table 2.1: Permutations of $\{0, \ldots, 15\}$ used by the BLAKE functions.

These four inputs represent 30 words in total (i.e., 120 bytes = 960 bits). The output of the function is a new chain value $h' = h'_0, \ldots, h'_7$ of eight words (i.e., 32 bytes = 256 bits). We write the compression of h, m, s, t to h' as

$$h' = \textbf{compress}(h, m, s, t)$$

Initialization

A 16-word state $v_0, \ldots, v_{15}$ is initialized such that different inputs produce different initial states. The state is represented as a 4×4 matrix, and filled as follows:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

Round function

Once the state v is initialized, the compression function iterates a series of 10 rounds. A round is a transformation of the state v, which computes

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

where, at round r, $G_i(a, b, c, d)$ sets[2]

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

[2]In the rest of the paper, for statements that don't depend on the index i we shall omit the subscript and write simply G.

The first four calls $G_0, \dots, G_3$ can be computed in parallel, because each of them updates a distinct column of the matrix. We call the procedure of computing $G_0, \dots, G_3$ a *column step*. Similarly, the last four calls $G_4, \dots, G_7$ update distinct diagonals thus can be parallelized as well, which we call a *diagonal step*.

Figures 2.1 and 2.2 illustrate G_i, the column step, and the diagonal step. An example of computation is given in Appendix A.

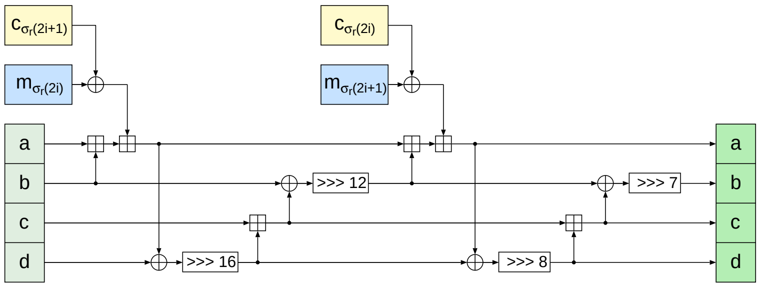

Figure 2.1: The G_i function.

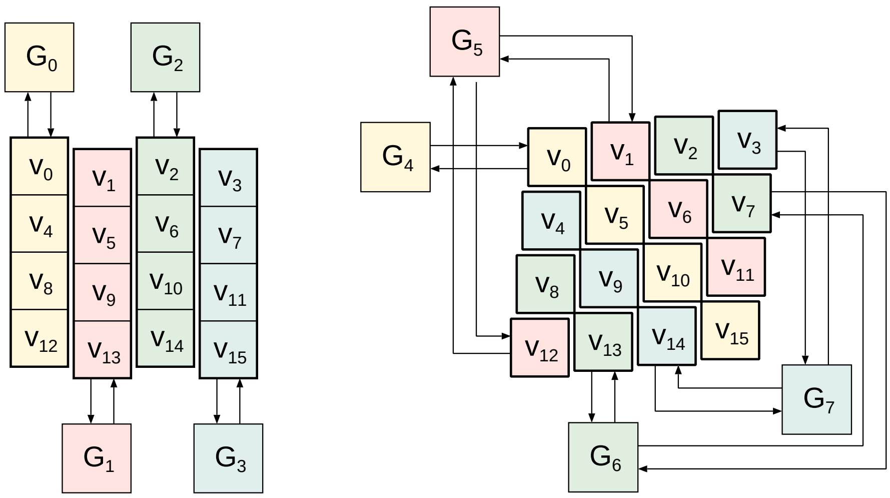

Figure 2.2: Column step and diagonal step.

Finalization

After the rounds sequence, the new chain value $h'_0, \dots, h'_7$ is extracted from the state $v_0, \dots, v_{15}$ with input of the initial chain value $h_0, \dots, h_7$ and the salt $s_0, \dots, s_3$:

$$\begin{aligned}
h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{aligned}$$

2.1.3 Hashing a message

We now describe the procedure for hashing a message m of bit length $\ell < 2^{64}$. As it is usual for iterated hash functions, the message is first *padded* (BLAKE uses a padding rule very similar to that of HAIFA), then it is processed block per block by the compression function.

Padding

First the message is extended so that its length is congruent to 447 modulo 512. Length extension is performed by appending a bit 1 followed by a sufficient number of 0 bits. At least one bit and at most 512 are appended. Then a bit 1 is added, followed by a 64-bit unsigned big-endian representation of ℓ. Padding can be represented as

$$m \leftarrow m \| 1000\dots0001\langle\ell\rangle_{64}$$

This procedure guarantees that the bit length of the padded message is a multiple of 512.

Iterated hash

To proceed to the iterated hash, the padded message is split into 16-word blocks $m^0, \dots, m^{N-1}$. We let ℓ^i be the number of message bits in $m^0, \dots, m^i$, that is, excluding the bits added by the padding. For example, if the original (non-padded) message is 600-bit long, then the padded message has two blocks, and $\ell^0 = 512$, $\ell^1 = 600$. A particular case occurs when the last block contains *no original message bit*; for example a 1020-bit message leads to a padded message with three blocks (which contain respectively 512, 508, and 0 message bits), and we set $\ell^0 = 512$, $\ell^1 = 1020$, $\ell^2 = 0$. The general rule is: if the last block contains no bit from the original message, then the counter is set to zero; this guarantees that if $i \neq j$, then $\ell_i \neq \ell_j$.

The salt s is chosen by the user, and set to the null value when no salt is required (i.e., $s_0 = s_1 = s_2 = s_3 = 0$). The hash of the padded message m is then computed as follows:

$h^0 \leftarrow \mathrm{IV}$
for $i = 0, \dots, N-1$
    $h^{i+1} \leftarrow$ **compress**(h^i, m^i, s, ℓ^i)
return h^N

The procedure of hashing m with BLAKE-32 is aliased BLAKE-32$(m, s) = h^N$, where m is the (non-padded) message, and s is the salt. The notation BLAKE-32(m) denotes the hash of m when no salt is used (i.e., $s = 0$).

2.2 BLAKE-64

BLAKE-64 operates on 64-bit words and returns a 64-byte hash value. All lengths of variables are doubled compared to BLAKE-32: chain values are 512-bit, message blocks are 1024-bit, salt is 256-bit, counter is 128-bit.

2.2.1 Constants

The initial value of BLAKE-64 is the same as for SHA-512:

$$
\begin{array}{ll}
\mathsf{IV}_0 = \texttt{6A09E667F3BCC908} & \mathsf{IV}_1 = \texttt{BB67AE8584CAA73B} \\
\mathsf{IV}_2 = \texttt{3C6EF372FE94F82B} & \mathsf{IV}_3 = \texttt{A54FF53A5F1D36F1} \\
\mathsf{IV}_4 = \texttt{510E527FADE682D1} & \mathsf{IV}_5 = \texttt{9B05688C2B3E6C1F} \\
\mathsf{IV}_6 = \texttt{1F83D9ABFB41BD6B} & \mathsf{IV}_7 = \texttt{5BE0CD19137E2179}
\end{array}
$$

BLAKE-64 uses the constants[3]

$$
\begin{array}{ll}
c_0 = \texttt{243F6A8885A308D3} & c_1 = \texttt{13198A2E03707344} \\
c_2 = \texttt{A4093822299F31D0} & c_3 = \texttt{082EFA98EC4E6C89} \\
c_4 = \texttt{452821E638D01377} & c_5 = \texttt{BE5466CF34E90C6C} \\
c_6 = \texttt{C0AC29B7C97C50DD} & c_7 = \texttt{3F84D5B5B5470917} \\
c_8 = \texttt{9216D5D98979FB1B} & c_9 = \texttt{D1310BA698DFB5AC} \\
c_{10} = \texttt{2FFD72DBD01ADFB7} & c_{11} = \texttt{B8E1AFED6A267E96} \\
c_{12} = \texttt{BA7C9045F12C7F99} & c_{13} = \texttt{24A19947B3916CF7} \\
c_{14} = \texttt{0801F2E2858EFC16} & c_{15} = \texttt{636920D871574E69}
\end{array}
$$

Permutations are the same as for BLAKE-32 (see Table 2.1).

2.2.2 Compression function

The compression function of BLAKE-64 is similar to that of BLAKE-32 except that it makes 14 rounds instead of 10, and that $\mathsf{G}_i(a, b, c, d)$ computes

$$
\begin{array}{rcl}
a & \leftarrow & a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
d & \leftarrow & (d \oplus a) \ggg 32 \\
c & \leftarrow & c + d \\
b & \leftarrow & (b \oplus c) \ggg 25 \\
a & \leftarrow & a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
d & \leftarrow & (d \oplus a) \ggg 16 \\
c & \leftarrow & c + d \\
b & \leftarrow & (b \oplus c) \ggg 11
\end{array}
$$

The only differences with BLAKE-32's G_i are the word length (64 bits instead of 32) and the rotation distances. At round $r > 9$, the permutation used is $\sigma_{r \bmod 10}$ (for example, in the last round $r = 13$ and the permutation $\sigma_{13 \bmod 10} = \sigma_3$ is used).

[3] First digits of π.

2.2.3 Hashing a message

For BLAKE-64, message padding goes as follows: append a bit 1 and as many 0 bits until the message bit length is congruent to 895 modulo 1024. Then append a bit 1, and a 128-bit unsigned big-endian representation of the message bit length:

$$m \leftarrow m \| 1000 \ldots 0001 \langle \ell \rangle_{128}$$

This procedure guarantees that the length of the padded message is a multiple of 1024.

The algorithm for iterated hash is identical to that of BLAKE-32.

2.3 BLAKE-28

BLAKE-28 is similar to BLAKE-32, except that

- it uses the initial value of SHA-224:

$$\begin{aligned} \mathsf{IV}_0 &= \mathtt{C1059ED8} & \mathsf{IV}_1 &= \mathtt{367CD507} \\ \mathsf{IV}_2 &= \mathtt{3070DD17} & \mathsf{IV}_3 &= \mathtt{F70E5939} \\ \mathsf{IV}_4 &= \mathtt{FFC00B31} & \mathsf{IV}_5 &= \mathtt{68581511} \\ \mathsf{IV}_6 &= \mathtt{64F98FA7} & \mathsf{IV}_7 &= \mathtt{BEFA4FA4} \end{aligned}$$

- in the padded data, the 1 bit preceeding the message length is replaced by a 0 bit:

$$m \leftarrow m \| 1000 \ldots 0000 \langle \ell \rangle_{64}$$

- the output is truncated to its first 224 bits, that is, the iterated hash returns $h_0^N, \ldots, h_6^N$ instead of $h^N = h_0^N, \ldots, h_7^N$

2.4 BLAKE-48

BLAKE-48 is similar to BLAKE-64, except that

- it uses the initial value of SHA-384:

$$\begin{aligned} \mathsf{IV}_0 &= \mathtt{CBBB9D5DC1059ED8} & \mathsf{IV}_1 &= \mathtt{629A292A367CD507} \\ \mathsf{IV}_2 &= \mathtt{9159015A3070DD17} & \mathsf{IV}_3 &= \mathtt{152FECD8F70E5939} \\ \mathsf{IV}_4 &= \mathtt{67332667FFC00B31} & \mathsf{IV}_5 &= \mathtt{8EB44A8768581511} \\ \mathsf{IV}_6 &= \mathtt{DB0C2E0D64F98FA7} & \mathsf{IV}_7 &= \mathtt{47B5481DBEFA4FA4} \end{aligned}$$

- in the padded data, the 1 bit preceeding the message length is replaced by a 0 bit:

$$m \leftarrow m \| 1000 \ldots 0000 \langle \ell \rangle_{128}$$

- the output is truncated to its first 384 bits, that is, the iterated hash returns $h_0^N, \ldots, h_5^N$ instead of $h^N = h_0^N, \ldots, h_7^N$

2.5 Alternative descriptions

The round function of BLAKE described in §2.1.2 operates first on columns of the matrix state, second on diagonals (see Fig. 2.2). Another way to view this transformation is

1. make a column-step
2. rotate the i^{th} column up by i positions, for $i = 0, \dots, 3$
3. make a *row-step* (see Fig. 2.3), that is,

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

A similar description was used for the stream cipher Salsa20 [9].

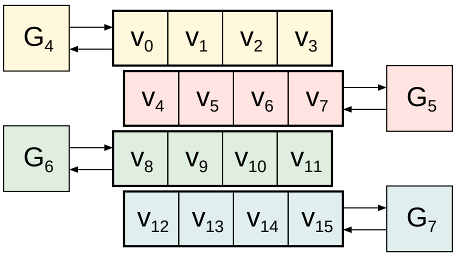

Figure 2.3: Row step of the alternative description.

Similarly, the transformation could be viewed as follows:

1. make a column-step
2. rotate the i^{th} row by i positions left, for $i = 0, \dots, 3$
3. make a column-step again

Finally, another equivalent definition of a round is

$$\begin{array}{llll} G_0\ (v_0\ ,v_4\ ,v_8\ ,v_{12}) & G_2\ (v_1\ ,v_5\ ,v_9\ ,v_{13}) & G_4\ (v_2\ ,v_6\ ,v_{10},v_{14}) & G_6\ (v_3\ ,v_7\ ,v_{11},v_{15}) \\ G_8\ (v_0\ ,v_5\ ,v_{10},v_{15}) & G_{10}(v_1\ ,v_6\ ,v_{11},v_{12}) & G_{12}(v_2\ ,v_7\ ,v_8\ ,v_{13}) & G_{14}(v_3\ ,v_4\ ,v_9\ ,v_{14}) \end{array}$$

where $G_i(a, b, c, d)$ is redefined to

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

This definition may speed up implementations by saving the doublings.

2.6 Tunable parameter

In its call for a new hash function [36], NIST encourages the description of a parameter that allows speed/confidence trade-offs. For BLAKE this parameter is the *number of rounds*. We estimate that 5 rounds are a minimum for BLAKE-32 (and BLAKE-28), and we recommend 10 rounds. For BLAKE-64 (and BLAKE-48), 7 rounds are a minimum and we recommend 14 rounds. Rationales behind these choices appear in Chapter 5.

3 Performance

We implemented BLAKE in several environments (software and hardware). This chapter reports results from our implementations.

3.1 Generalities

This section gives general facts about the complexity of BLAKE, independently of any implementation.

3.1.1 Complexity

Number of operations

A single G makes 6 XOR's, 6 additions and 4 rotations, so 16 arithmetic operations in total[1]. Hence a round makes 48 XOR's, 48 additions and 32 rotations, so 128 operations. BLAKE-32's compression function thus counts 480 XOR's, 480 additions, 320 rotations, plus 4 XOR's for the initialization and 24 XOR's for the finalization, thus a total of 1312 operations. BLAKE-64's compression function counts 672 XOR's, 672 additions, 448 rotations, plus 4 XOR's and 24 XOR's, thus a total of 1824 operations. We omit the overhead for initializing the hash structure, padding the message, etc., whose cost is negligible compared to that of a compression function.

Memory

BLAKE-32 needs to store in ROM 64 bytes for the constants, and 80 bytes to describe the permutations (144 bytes in total). In RAM, the storage m, h, s, t and v requires 184 bytes. In practice, however, more space might be required. For example, our implementation on the PIC18F2525 microcontroller (see §3.3) stores the 8-bit addresses of the permutation elements, not the 4-bit elements directly, thus using 160 bytes for storing the 80 bytes of information of the message permutations.

3.1.2 Memory/speed tradeoffs

A memory/speed tradeoff for a hash function implementation consists in storing a larger amount of data, in order to reduce the number of computation steps. This is relevant, for example, for hash functions that use a a large set of constants generated from a smaller set of constants. BLAKE, however, requires a fixed and small set of constants, which is not trivially compressible.

[1] The values in this paragraph should *not* be interpreted in terms of clock cycles.

Therefore, the algorithm of BLAKE admits no memory/speed tradeoff; the implementations reported in §3.2, 3.3, and 3.4 thus do not consider memory/speed tradeoffs. The tradeoffs made in the hardware implementations (§3.2) are rather space/speed than memory/speed.

3.1.3 Parallelism

When hashing a message, most of the time spent by the computing unit will be devoted to computing rounds of the compression function. Each round is composed of eight calls to the G function: $G_0, G_1, \ldots, G_7$. Simplifying:

- on a *serial* machine, the speed of a round is about eight times the speed of a G
- on a *parallel* machine, G_0, G_1, G_2 and G_3 can be computed in four parallel branches, and then G_4, G_5, G_6 and G_7 can be computed in four branches again. The speed of a round is thus about twice the speed of a G

Since parallelism is generally a trade-off, the gain in speed may increase the consumption of other resources (area, etc.). An example of trade-off is to split a round into two branches, resulting in a speed of four times that of a G.

3.2 ASIC and FPGA

We propose four hardware architectures of the BLAKE compression function and report the performances of the corresponding ASIC and FPGA implementations. Similar architectures have been considered by Henzen et al. for VLSI implementations of ChaCha, in [26].

3.2.1 Architectures

The HAIFA iteration mode forces a straightforward hardware implementation of the BLAKE compression function based on a single round unit and a memory to store the internal state variables $v_0, v_1, \ldots, v_{15}$. No pipeline circuits have been designed, due to the enormous resource requirements of such solutions. Nonetheless, several architectures of the compression function have been investigated to evaluate the relation between speed and area. Every implemented circuit reports to the basic block diagram of Fig 3.1.

Besides memory, the four main block components of BLAKE are

- the *initialization* and *finalization* blocks, which are pure combinational logic; initialization contains eight 32/64-bit XOR logic gates to compute the initial state v, while finalization consists of 24 XOR gates to generate the next chain value.
- the *round function*, which is essentially one or more G functions; G is composed of six modulo $2^{32}/2^{64}$ adders and six XOR gates. Rotations are implemented as a straight rerouting of the internal word bits without any additional logic and without affecting the propagation delay of the circuit.
- the *control unit*, which controls the computation of the compression function, aided by IO enable signals.

Four architectures with different round units have been investigated:

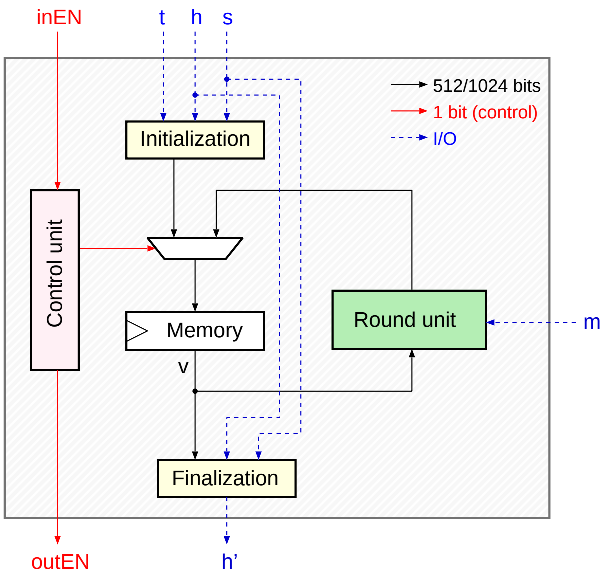

Figure 3.1: Block diagram of the BLAKE compression function. The signals `inEn` and `outEN` define the input and output enables.

- [8G]-BLAKE: This design corresponds to the isomorphic implementation of the round function. Eight G function units are instantiated; the first four units work in parallel to compute the column step, while the last four compute the diagonal step.

- [4G]-BLAKE: The round module consists of four parallel G units, which, at a given cycle, compute either the column step or the diagonal step.

- [1G]-BLAKE: The iterative decomposition of the compression function leads to the implementation of a single G function. Thus, one G unit processes the full round in eight cycles.

- [$\frac{1}{2}$G]-BLAKE: This lightweight implementation consists of a single half G unit. During one cycle, only a single update of the inputs a, b, c, d is processed (i.e., half a G).

In the last three architectures, additional multiplexers and demultiplexers driven by the control unit preserve the functionality of the algorithm, selecting the correct v elements inside and outside the round unit.

3.2.2 Implementation results

Based on functional VHDL coding (see Appendix B.1), the four designs have been synthesized using a 0.18 µm CMOS technology with the aid of the Synopsys Design Compiler Tool. Table 3.1 summarizes the final values of area, frequency, and throughput[2]. In addition, the hardware efficiency computes the ratio between speed and area of the circuits. The [8G]

[2]The unit Gbps means Gigabits per second, where a Gigabit is 1000^3 bits, not 1024^3. Similar rule applies to Mbps and Kbps in Tables 3.1 and 3.2.

and [4G]-BLAKE architectures maximize the throughput, so they were synthesized with speed optimization options at the maximal clock frequency. The target applications of [1G] and [$\frac{1}{2}$G]-BLAKE are resource-restricted environments, where a compact chip size is the main constraint. Hence, these designs have been synthesized at low frequencies to achieve minimum-area requirements.

Arch.	Function	Area [kGE]	Freq. [MHz]	Latency [cycles]	Throughput [Mbps]	Efficiency [Kbps/GE]
[8G]	BLAKE-32	58.30	114	11	5295	90.8
	BLAKE-64	132.47	87	15	5910	44.6
[4G]	BLAKE-32	41.31	170	21	4153	100.5
	BLAKE-64	82.73	136	29	4810	58.1
[1G]	BLAKE-32	10.54	40	81	253	24.0
	BLAKE-64	20.61	20	113	181	8.8
[$\frac{1}{2}$G]	BLAKE-32	9.89	40	161	127	12.9
	BLAKE-64	19.46	20	225	91	4.7

Table 3.1: ASIC synthesis results. One gate equivalent (GE) corresponds to the area of a two-input drive-one NAND gate of size 9.7 μm^2.

Three architectures have been implemented on FPGA silicon devices: the Xilinx Virtex-5, Virtex-4, and Virtex-II Pro[3]. We used SynplifyPro and Xilinx ISE for synthesis and place & route. Table 3.2 reports resulting circuit performances.

	XC2VP50			XC4VLX100			XC5VLX110		
Function	Area [slices]	Freq. [MHz]	Thr. [Mbps]	Area [slices]	Freq. [MHz]	Thr. [Mbps]	Area [slices]	Freq. [MHz]	Thr. [Mbps]
[8G]-BLAKE architecture									
BLAKE-32	3091	37	1724	3087	48	2235	1694	67	3103
BLAKE-64	11122	17	1177	11483	25	1707	4329	35	2389
[4G]-BLAKE architecture									
BLAKE-32	2805	53	1292	2754	70	1705	1217	100	2438
BLAKE-64	6812	31	1104	6054	40	1413	2389	50	1766
[1G]-BLAKE architecture									
BLAKE-32	958	59	371	960	68	430	390	91	575
BLAKE-64	1802	36	326	1856	42	381	939	59	533

Table 3.2: FPGA post place & route results [overall effort level: standard]. A single Virtex-5 slice contains twice the number of LUTs and FFs.

For the ASIC and the FPGA implementations, the memory of the internal state consists of 16 32/64-bit registers, which are updated every round with the output words of the round unit. No RAM or ROM macro cells are used to store the 16 constants $c_0, \ldots, c_{15}$. In the same

[3] Data sheets available at `http://www.xilinx.com/support/documentation/`

way, the ten permutations $\sigma_0, \ldots, \sigma_9$ have been hard-coded in VHDL. In ASIC, this choice has been motivated by the insufficient memory requirement of these variables. In FPGA, constants and permutations can be stored in dedicated block RAMs. This solution decreases slightly the number of slices needed, but does not speed-up the circuits.

A complete implementation of BLAKE (to include memory storing intermediate values, counter, and circuits to finalize the message, etc.) leads to an increase of about 1.8 kGE or 197 slices for ASIC and FPGA, respectively.

Minimizing the area

An ASIC architecture even smaller than $[\frac{1}{2}G]$ can be reached, by making a circuit only for a quarter (rather than a half) of the G function, and serializing the finalization block. Latency and throughput deteriorate much, but we can reach an area of 8.4 kGE. We omit an extensive description of this architecture because the area reduction from $[\frac{1}{2}G]$ is not worth its cost, in general.

3.2.3 Evaluation

The scalable structure of the round function allows the implementation of distinct architectures, where the trade-off between area and speed differs. Fast circuits are able to achieve throughput about 6 Gbps in ASIC and 3 Gbps in modern FPGA chips, while lightweight architectures require less than 10 kGE or 1000 Slices. BLAKE turns out to be an extremely flexible function, that can be integrated in a wide range of applications, from modern high-speed communication security protocols to low-area RFID systems.

3.3 8-bit microcontroller

The compression function of BLAKE-32 was implemented in a PIC18F2525 microcontroller. About 1800 assembly lines were written, using Microchip's MPLAB Integrated Development Environment v7.6. This section reports results of this implementation, starting with a presentation of the device used. Sample assembly code computing the round function is given in Appendix B.2.

3.3.1 The PIC18F2525

The PIC18F2525 is a member of the PIC family of microcontrollers made by Microchip Technology. PIC's are very popular for embedded systems (more than 6 billions sold). The PIC18F2525 works with 8-bit words, but has an instruction width of 16 bits; it makes up to 10 millions of instructions per second (MIPS).

Following the Harvard architecture, the PIC18F2525 separates program memory and data memory:

- *program memory* is where the program resides, and can store 48 Kb in flash memory (that is, 24576 instructions)
- *data memory* is reserved to the data used by the program. It can store 3986 bytes in RAM and 1024 bytes in EEPROM.

Program memory will contain the code of our BLAKE implementation, including the permutations' look-up tables, while variables will be stored in the data memory.

Our PIC processor runs at up to 40 MHz, and a single-cycle instruction takes four clock cycles (10 MIPS). In the following we give cost estimates in terms of instruction cycles, not clock cycles.

Operating frequency	DC – 40 MHz
Program memory (bytes)	49152
Program memory (instructions)	24576
Data memory (bytes)	3968
Data EEPROM (bytes)	1024
Interrupt sources	19
I/O ports	Ports A, B, C, (E)
Timers	4
Serial communication	MSSP, enhanced USART
Parallel communications	no
Instruction set	75 instructions (83 with extended IS)

Table 3.3: Main features of the PIC18F2525

Features of the PIC18F2525 are summarized in Table 3.3. All details can be found on Wikpedia[4] and in Microchip's datasheet[5].

3.3.2 Memory management

Our implementation requires 2470 bytes of program memory (including the look-up tables for the permutations), out of 48 Kb available. Data memory stores 274 bytes in RAM for the input variables, constants, and temporary variables, that is:

- message block m (64 bytes)
- chain value h (32 bytes)
- salt s (16 bytes)
- counter t (8 bytes)
- constants $c_0, \dots, c_{15}$ (64 bytes)
- internal state v (64 bytes)
- temporary variables (a, b, c, d) for G (16 bytes)
- other temporary variables (10 bytes)

To summarize, BLAKE-32 uses 5% of the program memory, 7% of the RAM, and no EEPROM.

[4]http://en.wikipedia.org/wiki/PIC_micro
[5]http://ww1.microchip.com/downloads/en/DeviceDoc/39626b.pdf

3.3.3 Speed

BLAKE-32 only uses the three operations XOR, 32-bit integer addition, and 32-bit rotation. In the PIC18F2525 the basic unit is a byte, not a 32-bit word, hence 32-bit operations have to be simulated with 8-bit instructions:

- 32-bit XOR is simulated by four independent 8-bit XOR's
- 32-bit addition is simulated by four 8-bit additions with manual transfer of the carry between each addition
- 32-bit rotation is simulated using byte swaps and 1-bit rotate instructions

Rotations are the most complicated operations to implement, because a different code has to be written for each rotation distance; rotation of 8 or 16 positions requires no rotate instruction, while one is needed for 7-bit rotation, and four for 12-bit rotation. For example, the code for a 8-bit rotation of `x=x_hi`‖`x_mh`‖`x_ml`‖`x_lo` looks like

```
movFF x_hi,tmp
movFF x_mh,x_hi
movFF x_ml,x_mh
movFF x_lo,x_ml
movFF tmp,x_lo
```

while the code for a 7-bit rotation looks like

```
bcf STATUS, C
btfsc x_lo,0
bsf STATUS, C
rrcF x_hi
rrcF x_mh
rrcF x_ml
rrcF x_lo
movFF x_lo,tmp
movFF x_hi,x_lo
movFF x_mh,x_hi
movFF x_ml,x_mh
movFF tmp,x_ml
```

In terms of cycles, counting all the instructions needed (rotate, move, etc.), we have that

- $\ggg$ 16 needs 6 cycles
- $\ggg$ 12 needs 22 cycles
- $\ggg$ 8 needs 5 cycles
- $\ggg$ 7 needs 12 cycles

Below we detail the maximum cost of each line of the G_i function:

$$\begin{array}{rlcl}
\text{(76 cycles)} & a & \leftarrow & a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
\text{(24 cycles)} & d & \leftarrow & (d \oplus a) \ggg 16 \\
\text{(24 cycles)} & c & \leftarrow & c + d \\
\text{(34 cycles)} & b & \leftarrow & (b \oplus c) \ggg 12 \\
\text{(67 cycles)} & a & \leftarrow & a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
\text{(22 cycles)} & d & \leftarrow & (d \oplus a) \ggg 8 \\
\text{(24 cycles)} & c & \leftarrow & c + d \\
\text{(29 cycles)} & b & \leftarrow & (b \oplus c) \ggg 7
\end{array}$$

The cycle count is different for $(b \oplus c) \ggg 12$ and $(b \oplus c) \ggg 7$ because of the different rotation distances. The fifth line needs fewer cycles than the first because of the proximity of the indices (though not of the addresses).

In addition, preparing G_i's inputs costs 18 cycles, and calling it 4 cycles, thus in total 322 cycles are needed for computing a G_i. Counting the initialization of v (at most 161 cycles) and the overhead of 8 cycles per round, the compression function needs 26001 cycles (that is, 406 cycles per byte). With a 32 MHz processor (8 MIPS), it takes about 3.250 ms to hash a single message block (a single instruction is 125 ns long); with a 40 MHz processor (10 MIPS), it takes about 2.6 ms.

No precomputation is required to set up the algorithm (BLAKE does not require building internal tables before hashing a message, neither it requires the initialization of a particular data structure, for example). On the PIC18F2525, the only setup cost is for preparing the device, i.e. loading data into the data memory; this cost cannot be expressed (solely) in terms of clock cycles, because of interrupt routines and waiting time, which depend on the data source considered.

For sufficiently large messages (say, a few blocks), the cost of preparing the device and of padding the message is negligible, compared to the cost of computing the compression functions. In this case, generating one message digest with BLAKE-28 or BLAKE-32 on a PIC18F2525 requires about 406 cycles per byte.

3.4 Large processors

BLAKE is easily implemented on 32- and 64-bit processors: it works on words of 32 or 64 bits, and only makes wordwise operations (XOR, rotation, addition) that are implemented in most of the processors. It is based on ChaCha, one of the fastest stream ciphers. The speed-critical code portion is short and thus is relatively easy to optimize. Because the core of BLAKE is just the G function (16 operations), implementations are simple and compact.

As requested by NIST, we wrote a reference implementation and optimized implementations in ANSI C. Here we report speed benchmarks based on the optimized implementation, which will be used by NIST for comparing BLAKE with other candidates. On specific processors, faster implementations can be obtained by programming BLAKE in assembly; one may directly reuse the assembly programs of ChaCha available[6].

We compiled our program with `gcc 4.1.0` with options `-O3 -fomit-frame-pointer -Wall -ansi`. We report speeds for various lengths of (aligned) messages, and give the median measurement over a hundred trials. We measured the time of a call to the function `Hash` specified in NIST's API, which includes

[6] See `http://cr.yp.to/chacha.html`

1. function `Init`: initialization of the function parameters, copy of the instance's IV
2. function `Update`: iterated hash of the message
3. function `Final`: padding of the message, compression (at most two) of the remaining data

Table 3.4 reports the number of clock cycles required to generate one message digest with the full versions of BLAKE-32 and BLAKE-64 and for reduced-round versions, depending on the message length. BLAKE-28 and BLAKE-48 show performance similar to BLAKE-32 and BLAKE-64, respectively. The "Core 2 Duo" platform corresponds to the *NIST SHA-3 Reference Platform*, except that our computer was running Linux instead of Windows Vista.

For any digest length, a negligible number of cycles is required to setup the algorithm. This is because no precomputation is necessary, and the only preparation consists in loading data in memory.

Data length [bytes]	10	100	1000	10000
Celeron M (32-bit mode)				
BLAKE-32 (10 rounds)	≈1500	50.1	24.5	22.2
BLAKE-32 (8 rounds)	≈1500	56.5	21.7	18.5
BLAKE-32 (5 rounds)	≈1500	43.2	13.9	12.5
BLAKE-64 (14 rounds)	≈2000	126.4	64.4	58.8
BLAKE-64 (10 rounds)	≈2000	99.7	47.7	43.1
BLAKE-64 (7 rounds)	≈2000	93.5	32.5	30.8
Core 2 Duo (32-bit mode)				
BLAKE-32 (10 rounds)	≈2900	51.5	27.4	28.3
BLAKE-32 (8 rounds)	≈2900	45.2	22.6	24.2
BLAKE-32 (5 rounds)	≈2900	35.0	15.9	14.0
BLAKE-64 (14 rounds)	≈4400	94.0	61.3	61.7
BLAKE-64 (10 rounds)	≈4400	74.0	45.4	57.6
BLAKE-64 (7 rounds)	≈4400	58.9	32.5	41.0
Core 2 Duo (64-bit mode)				
BLAKE-32 (10 rounds)	≈1600	36.4	18.4	16.7
BLAKE-32 (8 rounds)	≈1600	32.2	15.4	13.8
BLAKE-32 (5 rounds)	≈1600	26.9	10.9	9.6
BLAKE-64 (14 rounds)	≈1900	33.7	13.8	12.3
BLAKE-64 (10 rounds)	≈1900	29.9	11.6	9.3
BLAKE-64 (7 rounds)	≈1900	26.8	8.5	7.2

Table 3.4: Performance of our optimized C implementation of BLAKE (in cycles/byte), on a 900 MHz Intel Celeron M and a 2.4 GHz Intel Core 2 Duo.

In terms of bytes-per-second, the top speed is achieved by BLAKE-64 in 64-bit mode, with about 317 Mbps. For very small messages (10 bytes) the overhead is due to the compression of 64 (respectively 128) bytes, and to the cost of initializing and padding the message. The cost per byte quickly decreases, and stabilizes after 1000-byte messages. Although different processors were used, our estimates can be compared with the fastest C implementation of SHA-256, by Gladman[7]: in 64-bit mode on a AMD processor, SHA-256 runs at 20.4 cycles-per-byte, and SHA-512 at 13.4 cycles-per-byte.

[7] http://fp.gladman.plus.com/cryptography_technology/sha/index.htm

4 Using BLAKE

BLAKE is intended to replace SHA-2 with a minimal engineering effort, and to be used wherever SHA-2 is. BLAKE provides the same interface as SHA-2, with the optional input of a salt. BLAKE is suitable whenever a cryptographic hash function is needed, be it for digital signatures, MAC's, commitment, password storage, key derivation, etc.

This chapter explains how the salt input should (not) be used, and construction details based on BLAKE for HMAC and UMAC, PRF ensembles, and randomized hashing.

4.1 Hashing with a salt

The BLAKE hash functions take as input a message and a salt. The aim of hashing with distinct salts is to hash with different functions but using the same algorithm. Depending on the application, the salt can be chosen randomly (thus reusing a same salt twice can occur, though with small probability), or derived from a counter (nonce).

For applications in which no salt is required, it is set to the null value ($s = 0$). In this case the initialization of the state v simplifies to

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ c_0 & c_1 & c_2 & c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

and the finalization of the compression function becomes

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

The salt input may contain a nonce or a random seed, for example. A typical application is for password storage. However, the salt input is not intended to contain the secret key for a MAC construction. We recommend using HMAC or UMAC for MAC functionality, which are much more efficient.

4.2 HMAC and UMAC

HMAC [6] can be built on BLAKE similarly to SHA-2. The salt input is not required, and should thus be set to zero (see 4.1). BLAKE has no property that limits its use for HMAC, compared to SHA-2. For example, HMAC based on BLAKE-32 takes as input a key k and a message m and computes

$$\text{HMAC}_k(m) = \text{BLAKE-32}\big(k \oplus \text{opad} \| \text{BLAKE-32}(k \oplus \text{ipad} \| m)\big).$$

All details on the HMAC construction are given in the NIST standardization report [35] or in the original publication [6].

UMAC is a MAC construction "faster but more complex" [13] than HMAC: it is based on the "$\text{PRF}(\text{hash}, \text{nonce})$" approach, where the value "hash" is a universal hash of the message authenticated. UMAC authors propose to instanciate the PRF with HMAC based on SHA-1, computing $\text{HMAC}_k(\text{nonce}\|\text{hash})$.

For combining BLAKE with UMAC, the same approach can be used, namely using HMAC based on BLAKE. It is however more efficient to use BLAKE's salt, and thus compute $\text{HMAC}(\text{hash})$ with $s = \text{nonce}$:

$$\text{HMAC}_k(\text{hash}) = \text{BLAKE-32}\big(k \oplus \text{opad} \| \text{BLAKE-32}(k \oplus \text{ipad} \| \text{hash}, \text{nonce}), \text{nonce}\big)$$

In the best case, setting $s = \text{nonce}$ saves one compression compared to the original construction, while in the worst case performance is unchanged. UMAC authors suggest a nonce of 64 bits [13], which fits in the salt input of all BLAKE functions. We recommend this construction for UMAC based on BLAKE.

4.3 PRF ensembles

To construct pseudorandom functions (PRF) ensembles from hash functions, a common practice is to append or prepend the index data to the message. For example, for an arbitrary message m one can define the i^{th} function of the ensemble as

$$\text{BLAKE-32}(m\|i) \text{ or } \text{BLAKE-32}(i\|m)$$

where i is encoded over a fixed number of bits. These techniques pose no problem specific to BLAKE. The second construction is even more secure than with SHA-2, because it makes some length-extension attacks impossible (cf. [6, §6] and §5.5.1).

Another technique proposed for constructing PRF ensembles is to modify the IV according to the index data. That is, the i^{th} function of the ensemble has an IV equal to (some representation of) i. A concrete construction that exploits this technique is NMAC [6], which computes a MAC as

$$\text{NMAC}_{k_1\|k_2}(m) = H_{k_1}\left(H_{k_2}(m)\right)$$

where H_k is a hash function with initial value k.

For combining BLAKE with NMAC, we recommend not to set directly $\text{IV} \leftarrow k_i$, $i = 1, 2$, but instead $\text{IV} \leftarrow \textbf{compress}(\text{IV}, i, 0, 0)$, starting from the IV specific to the function used. This makes the effective IV dependent on the function instance (cf. §2.1 and §2.3).

A last choice for constructing PRF's based on BLAKE is to use the salt for the index data, giving ensembles of 2^{128} and 2^{256} for BLAKE-32 and BLAKE-64, respectively.

4.4 Randomized hashing

Randomized hashing is mainly used for digital signatures (cf. [24, 37]): instead of sending the signature $\mathsf{Sign}(\mathsf{H}(m))$, the signer picks a random r and sends $(\mathsf{Sign}(\mathsf{H}_r(m)), r)$ to the verifier. The advantage of randomized hashing is that it relaxes the security requirements of the hash function [24]. In practice, random data is either appended/prepended to the message or combined with the message; for example the RMX transform [24], given a random r, hashes m to the value

$$\mathsf{H}\big(r\|(m^1 \oplus r)\| \ldots \|(m^{N-1} \oplus r)\big).$$

BLAKE offers a dedicated interface for randomized hashing, not a modification of a non-randomized mode: the input s, 128 or 256 bits long, should be dedicated for the salt of randomized hashing. This avoids the potential computation overhead of other methods, and allows the use of the function as a blackbox, rather than a special mode of operation of a classical hash function. BLAKE remains compatible with previous generic constructions, including RMX.

5 Elements of analysis

This chapter presents a preliminary analysis of BLAKE, with a focus on BLAKE-32. We study properties of the function's components, resistance to generic attacks, and dedicated attack strategies.

5.1 Permutations

The permutations $\sigma_0, \ldots, \sigma_9$ were chosen to match several security criteria: First we ensure that a same input difference doesn't appear twice at the same place (to complicate "correction" of differences in the state). Second, for a random message all values $(m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$ and $(m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$ should be distinct with high probability. For chosen messages, this guarantees that each message word will be XOR'd with different constants, and thus apply distinct transformations to the state through rounds. It also implies that no pair (m_i, m_j) is input twice in the same G_i. Finally, the position of the inputs should be balanced: in a round, a given message word is input either in a column step or in a diagonal step, and appears either first or second in the computation of G_i. We ensure that each message word appears as many times in a column step as in a diagonal step, and as many times first as second within a step. To summarize:

1. no message word should be input twice at the same point
2. no message word should be XOR'd twice with the same constant
3. each message word should appear exactly 5 times in a column step and 5 times in a diagonal step
4. each message word should appear exactly 5 times in first position in G and 5 times in second position

This is equivalent to say that, in the representation of permutations in §2.1.1 (also see Table 5.1):

1. for all $i = 0, \ldots, 15$, there should exist no distinct permutations σ, σ' such that $\sigma(i) = \sigma'(i)$
2. no pair (i, j) should appear twice at an offset of the form $(2k, 2k+1)$, for all $k = 0, \ldots, 7$
3. for all $i = 0, \ldots, 15$, there should be 5 distinct permutations σ such that $\sigma(i) < 8$, and 5 such that $\sigma(i) > 8$
4. for all $i = 0, \ldots, 15$, there should be 5 distinct permutations σ such that $\sigma(i)$ is even, and 5 such that $\sigma(i)$ is odd

In BLAKE-64, four of the permutations are repeated because it makes 14 rounds instead of 10. The above criteria thus just apply to the first ten rounds. The slight loss of balance in the four last rounds seems unlikely to affect security.

Round	G_0		G_1		G_2		G_3		G_4		G_5		G_6		G_7	
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Table 5.1: Input of message words.

5.2 Compression function

This section reports a bottom-up analysis of BLAKE's compression function.

5.2.1 G function

Given (a, b, c, d) and message block(s) m_j, $j \in \{0, \dots, 15\}$; a function G_i computes

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

The G function is inspired from the "quarter-round" function of the stream cipher ChaCha, which transforms (a, b, c, d) as follows:

$$\begin{aligned}
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \lll 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \lll 12 \\
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \lll 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \lll 7
\end{aligned}$$

To build BLAKE's compression function based on this algorithm, we add input of two message words and constants, and let the function be otherwise unchanged. We keep the rotation distances of ChaCha, which provide a good trade-off security/efficiency: 16- and 8-bit rotations preserve byte alignment, so are fast on 8-bit processors (no rotate instruction is needed), while 12- and 7-bit rotations break up the byte structure, and are reasonably fast.

ChaCha's function is itself an improvement of the "quarter round" of the stream cipher Salsa20. The idea of a 4×4 state with four parallel mappings for rows and columns goes back to the cipher Square [18], and was then successfuly used in Rijndael [19], Salsa20 and ChaCha. Detailed design rationale and preliminary analysis of ChaCha and Salsa20 can be found in [7, 9], and cryptanalysis in [3, 17, 27, 39].

Bijectivity

Given a message m, and a round index r, the inverse function of G_i is defined as follows:

$$\begin{array}{rcl} b & \leftarrow & c \oplus (b \lll 7) \\ c & \leftarrow & c - d \\ d & \leftarrow & a \oplus (d \lll 8) \\ a & \leftarrow & a - b - (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\ b & \leftarrow & c \oplus (b \lll 12) \\ c & \leftarrow & c - d \\ d & \leftarrow & a \oplus (d \lll 16) \\ a & \leftarrow & a - b - (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \end{array}$$

Hence for any (a', b', c', d'), one can efficiently compute the unique (a, b, c, d) such that $G_i(a, b, c, d) = (a', b', c', d')$, given i and m. In other words, G_i is a permutation of $\{0, 1\}^{128}$.

Linear approximations

We found several linear approximations of differentials; the notation $(\Delta_0, \Delta_1, \Delta_2, \Delta_3) \mapsto (\Delta'_0, \Delta'_1, \Delta'_2, \Delta'_3)$ means that the two inputs with the leftmost difference lead to outputs with the rightmost difference, when $(m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) = (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) = 0$. For random inputs we have for example

- $(80000000, 00000000, 80000000, 80008000) \mapsto (80000000, 0, 0, 0)$ with probability 1
- $(00000800, 80000800, 80000000, 80000000) \mapsto (0, 0, 80000000, 0)$, with probability 1/2
- $(80000000, 80000000, 80000080, 00800000) \mapsto (0, 0, 0, 80000000)$, with probability 1/2

Many high probability differentials can be identified for G, and one can use standard message modification techniques (linearization, neutral bits) to identify a subset of inputs for which the probability is much higher than for the whole domain. Similar linear differentials exist in the Salsa20 function, and were exploited [3] to attack the compression function Rumba [8], breaking 3 rounds out of 20.

Particular properties of G are

1. the only fixed-point in G is the zero input
2. no preservation of differences can be obtained by linearization

The first observation is straightforward when writing the corresponding equations. The second point means that there exist no pair of inputs whose difference (according to XOR) is preserved in the corresponding pair of outputs, in the linearized model. This follows from the fact that, if an input difference gives the same difference in the output, then this difference must be a fixed-point for G; since the only fixed-point is the null value, there exists no such difference.

Diffusion

Diffusion is the ability of the function to quickly spread a small change in the input through the whole internal state. For example, G inputs message words such that any difference in a message word affects the four words output. Tables 5.2.1 and 5.3 give the average number of bits modified by G, given a random one-bit difference in the input, for each input word.

in\out	a	b	c	d
a	4.6	11.7	10.0	6.5
b	6.6	14.0	11.5	8.4
c	2.4	6.6	4.8	2.4
d	2.4	8.4	6.7	3.4

Table 5.2: Average number of changes in each output word given a random bit flip in each input word.

in\out	a	b	c	d
a	4.4	9.9	8.2	6.3
b	6.3	12.4	9.8	8.1
c	1.9	3.9	2.9	1.9
d	1.9	4.9	3.9	2.9

Table 5.3: Average number of changes in each output word given a random bit flip in each input word, in the XOR-linearized model.

5.2.2 Round function

The round function of BLAKE is

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

Bijectivity

Because G is a permutation, a round is a permutation of the inner state v for any fixed message. In other words, given a message and the value of v after r rounds, one can determine the value of v at rounds $r-1$, $r-2$, etc., and thus the initial value of v. Therefore, for a same initial state a sequence of rounds is a permutation of the message. That is, one cannot find two messages that produce the same internal state, after any number of rounds.

Diffusion and low-weight differences

After one round, all 16 words are affected by a modification of one bit in the input (be it the message, the salt, or the chain value). Here we illustrate diffusion through rounds with a concrete example, for the *null message* and the *null initial state*. The matrices displayed below represent the *differences* in the state after each step of the first two rounds (column step, diagonal step, column step, diagonal step), for a difference in the least significant bit of v_0:

$$\text{column step} \quad \begin{pmatrix} \texttt{00000037} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{E06E0216} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{37010B00} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{37000700} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \quad (\text{weight } 34)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{0000027F} & \texttt{10039015} & \texttt{5002B070} & \texttt{C418A7D4} \\ \texttt{66918CC7} & \texttt{1CBEEE25} & \texttt{F1A8535F} & \texttt{C111AD29} \\ \texttt{F8D104F0} & \texttt{6F08C6F9} & \texttt{5F77131E} & \texttt{E4291FE7} \\ \texttt{151703A7} & \texttt{705002B0} & \texttt{F2C22207} & \texttt{7F001702} \end{pmatrix} \quad (\text{weight } 219)$$

$$\text{column step} \quad \begin{pmatrix} \texttt{944F85FD} & \texttt{A044CCB3} & \texttt{9476A6BC} & \texttt{24B6ADAC} \\ \texttt{A729BBE9} & \texttt{6549BC3D} & \texttt{3A330361} & \texttt{7318B20D} \\ \texttt{7BF5F768} & \texttt{7831614B} & \texttt{CF44C968} & \texttt{53D886E2} \\ \texttt{5A1642B3} & \texttt{41B00EA0} & \texttt{A7115A95} & \texttt{7AC791D1} \end{pmatrix} \quad (\text{weight } 249)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{DFC2D878} & \texttt{F9FAAE7A} & \texttt{2D804D9A} & \texttt{3EF58B7F} \\ \texttt{FC91AF81} & \texttt{D78E2315} & \texttt{55048021} & \texttt{0811CC46} \\ \texttt{FB98AF71} & \texttt{DC27330E} & \texttt{47A19B59} & \texttt{EDDE442E} \\ \texttt{F042BB72} & \texttt{1C7A59AB} & \texttt{AC2EFFA4} & \texttt{2E76390B} \end{pmatrix} \quad (\text{weight } 264)$$

In comparison, in the linearized model (i.e., where all additions are replaced by XOR's), we have:

$$\text{column step} \quad \begin{pmatrix} \texttt{00000011} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{20220202} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{11010100} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{11000100} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \quad (\text{weight } 14)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{00000101} & \texttt{10001001} & \texttt{10011010} & \texttt{02202000} \\ \texttt{40040040} & \texttt{22022220} & \texttt{00202202} & \texttt{00222020} \\ \texttt{01110010} & \texttt{20020222} & \texttt{01111101} & \texttt{00111101} \\ \texttt{01110001} & \texttt{10100110} & \texttt{22002200} & \texttt{01001101} \end{pmatrix} \quad (\text{weight } 65)$$

$$\text{column step} \quad \begin{pmatrix} \texttt{54500415} & \texttt{13012131} & \texttt{02002022} & \texttt{20331103} \\ \texttt{2828A0A8} & \texttt{46222006} & \texttt{04006046} & \texttt{64646022} \\ \texttt{00045140} & \texttt{30131033} & \texttt{12113132} & \texttt{10010011} \\ \texttt{00551045} & \texttt{23203003} & \texttt{03121212} & \texttt{01311212} \end{pmatrix} \quad (\text{weight } 125)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{35040733} & \texttt{67351240} & \texttt{24050637} & \texttt{B1300980} \\ \texttt{27472654} & \texttt{8AE6CA08} & \texttt{EE4A6286} & \texttt{E08264A8} \\ \texttt{03531247} & \texttt{1AB89238} & \texttt{54132765} & \texttt{55051040} \\ \texttt{14360705} & \texttt{73540643} & \texttt{89128902} & \texttt{70030514} \end{pmatrix} \quad (\text{weight } 186)$$

The higher weight in the original model is due to the addition carries induced by the constants $c_0, \ldots, c_{15}$. A technique to avoid carries at the first round and get a low-weight output difference is to choose a message such that $m_0 = c_0, \ldots, m_{15} = c_{15}$. At the subsequent rounds, however, nonzero words are introduced because of the different permutations.

Diffusion can be delayed a few steps by combining high-probability and low-weight differentials of G, using initial conditions, neutral bits, etc. For example, applying directly the differential

$$(80000000, 00000000, 80000000, 80008000) \mapsto (80000000, 0, 0, 0)$$

the diffusion is delayed one step, as illustrated below:

$$\text{column step} \quad \begin{pmatrix} \texttt{80000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \quad (\text{weight } 1)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{800003E8} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{0B573F03} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{AB9F819D} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{E8800083} \end{pmatrix} \quad (\text{weight } 49)$$

$$\text{column step} \quad \begin{pmatrix} \texttt{8007E4A0} & \texttt{2075B261} & \texttt{18E78828} & \texttt{9800099E} \\ \texttt{5944FE53} & \texttt{F178A22F} & \texttt{86B0A65B} & \texttt{936C73CB} \\ \texttt{A27F0D24} & \texttt{98D6929A} & \texttt{4088A5FB} & \texttt{2E39EDA3} \\ \texttt{A08FFF64} & \texttt{2AD374B7} & \texttt{2818E788} & \texttt{1E9883E1} \end{pmatrix} \quad (\text{weight } 236)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{4B3CBDD2} & \texttt{0290847F} & \texttt{B4FF78F9} & \texttt{F1E71BA3} \\ \texttt{3A023C96} & \texttt{49908E86} & \texttt{F13BC1D7} & \texttt{ADC2020A} \\ \texttt{9DCA344A} & \texttt{827BF1E5} & \texttt{B20A8825} & \texttt{FE575BE3} \\ \texttt{FC81FE81} & \texttt{D676FFC9} & \texttt{80740480} & \texttt{52570CB2} \end{pmatrix} \quad (\text{weight } 252)$$

In comparison, for a same input difference in the linearized model we have

$$\text{column step} \quad \begin{pmatrix} \texttt{80000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix} \quad (\text{weight } 1)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{80000018} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{10310101} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{18808080} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{18800080} \end{pmatrix} \quad (\text{weight } 18)$$

$$\text{column step} \quad \begin{pmatrix} \texttt{80000690} & \texttt{E1101206} & \texttt{0801B818} & \texttt{B8000803} \\ \texttt{1D217176} & \texttt{600FC064} & \texttt{60111212} & \texttt{22167121} \\ \texttt{90B8B886} & \texttt{16E12133} & \texttt{00888138} & \texttt{83389890} \\ \texttt{90803886} & \texttt{17E01122} & \texttt{180801B8} & \texttt{83B88010} \end{pmatrix} \quad (\text{weight } 155)$$

$$\text{diagonal step} \quad \begin{pmatrix} \texttt{44E4E456} & \texttt{133468BD} & \texttt{DBBDA164} & \texttt{0F649833} \\ \texttt{4E20F629} & \texttt{563A9099} & \texttt{A62F3969} & \texttt{7773C0BE} \\ \texttt{FEB6F508} & \texttt{AABDCBF9} & \texttt{3262E291} & \texttt{87A10D6A} \\ \texttt{3C2B867B} & \texttt{B603B05C} & \texttt{DA695123} & \texttt{F88E8007} \end{pmatrix} \quad (\text{weight } 251)$$

These examples show that even in the linearized model, after two rounds about half of the state bits have changed when different initial states are used (similar figures can be given for a difference in the message). Using clever combinations of low-weight differentials and message modifications one may attack reduced versions with two or three rounds. However, differences after more than four steps seem very difficult to control.

5.2.3 Compression function

BLAKE's compression function is the combination of an initialization, a sequence of rounds, and a finalization. Contrary to ChaCha, BLAKE breaks self-similarity by using a round-dependent permutation of the message and the constants. This prevents attacks that exploit the similarity among round functions (cf. slide attacks in §5.6.3). Particular properties of the compression function are summarized below.

Initialization

At the initialization stage, constants and redundancy of t impose a nonzero initial state (and a non "all-one" state). The disposition of inputs implies that after the first column step the initial value h is directly mixed with the salt s and the counter t.

The double input of t_0 and t_1 in the initial state suggests the notion of *valid* initial state: we shall call an initial state $v_0, \dots, v_{15}$ valid if and only there exists t_0, t_1 such that $v_{12} = t_0 \oplus c_4$ and $v_{13} = t_0 \oplus c_5$, and $v_{14} = t_1 \oplus c_6$ and $v_{15} = t_1 \oplus c_7$. Non-valid states are thus impossible initial states.

Number of rounds

The choice of 10 rounds for BLAKE-32 was determined by

1. the cryptanalytic results on Salsa20, ChaCha, and Rumba (one BLAKE-32 round is essentially two ChaCha rounds, so the initial conservative choice of 20 rounds for ChaCha corresponds to 10 rounds for BLAKE-32): truncated differentials were observed for up to 4 Salsa20 rounds and 3 ChaCha rounds, and the Rumba compression function has shortcut attacks for up to 3 rounds; the eSTREAM project chose a version of Salsa20 with 12 rounds in its portfolio, and 12-round ChaCha is arguably as strong as 12-round Salsa20.

2. our results on early versions of BLAKE, which had similar high-level structure, but a round function different from the present one: for the worst version, we could find collisions for up to 5 rounds.

3. our results on the final BLAKE: full diffusion is achieved after two rounds, and the best differentials found can be used to attack two rounds only.

BLAKE-64 has 14 rounds, i.e., 4 more than BLAKE-32; this is because the larger state requires more rounds for achieving similar security (in comparison, SHA-512 has 1.25 times more rounds than SHA-256).

We believe that the choice of 10 and 14 rounds provides a high security margin, without sacrificing performance. The number of rounds may later be adjusted according to the future results on BLAKE (for example, 8 rounds for BLAKE-32 may be fine if the best attack breaks 4 rounds, while 12 rounds may be chosen if an attack breaks, say, 6 rounds).

Finalization

At the finalization stage, the state is compressed to half its length, in a way similar to that of the cipher Rabbit [14]. The feedforward of h and s makes each word of the hash value dependent on two words of the inner state, one word of the initial value, and one word of the salt. The goal is to make the function non-invertible when the initial value and/or the salt are unknown.

Our approach of "permutation plus feedforward" is similar to that of SHA-2, and can be seen as a particular case of Davies-Meyer-like constructions: denoting E the blockcipher defined by the round sequence, BLAKE's compression function computes

$$\mathsf{E}_{m\|s}(h) \oplus h \oplus (s\|s)$$

which, for a null salt, gives the Davies-Meyer construction $E_m(h) \oplus h$. We use XOR's and not additions (as in SHA-2), because here additions don't increase security, and are much more expensive in circuits and 8-bit processors.

If the salt s was unknown and not fedforward, then one would be able to recover it given a one-block message, its hash value, and the IV. This would be a critical property. The counter t is not input in the finalization, because its value is always known and never chosen by the users.

Local collisions

A *local collision* happens when, for two distinct messages, the internal states after a same number of rounds are identical. For BLAKE hash functions, there exists no local collisions for a same initial state (i.e., same IV, salt, and counter). This result directly follows from the fact that the round function is a permutation of the message, for fixed initial state v (and so different inputs lead to different outputs). This property generalizes to any number of rounds. The requirement of a same initial state does not limit much the result: for most of the applications, no salt is used, and a collision on the hash function implies a collision on the compression function with same initial state [10].

Full diffusion

Full diffusion is achieved when each input bit has a chance to affect each output bit. BLAKE-32 and BLAKE-64 achieve full diffusion after two rounds, given a difference in the IV, m, or s.

5.2.4 Fixed-points

A fixed-point for BLAKE's compression function is a tuple (m, h, s, t) such that

$$\textbf{compress}(m, h, s, t) = h$$

Functions of the form $E_m(h) \oplus h$ (like SHA-2) allow the finding of fixed-points for chosen messages by computing $h = E^{-1}(0)$, which gives $E_m(h) \oplus h = h$.

BLAKE's structure is a particular case of the Davies-Meyer-like constructions mentioned in §5.2.3; consider the case when no salt is used ($s = 0$), without loss of generality; for finding fixed-points, we have to choose the final v such that

$$\begin{aligned}
h_0 &= h_0 \oplus v_0 \oplus v_8 \\
h_1 &= h_1 \oplus v_1 \oplus v_9 \\
h_2 &= h_2 \oplus v_2 \oplus v_{10} \\
h_3 &= h_3 \oplus v_3 \oplus v_{11} \\
h_4 &= h_4 \oplus v_4 \oplus v_{12} \\
h_5 &= h_5 \oplus v_5 \oplus v_{13} \\
h_6 &= h_6 \oplus v_6 \oplus v_{14} \\
h_7 &= h_7 \oplus v_7 \oplus v_{15}
\end{aligned}$$

That is, we need $v_0 = v_8, v_1 = v_9, \dots, v_7 = v_{15}$, so there are 2^{256} possible choices for v. From this v we compute the round function backward to get the initial state, and we find a fixed-point when

- the third line of the state is $c_0, \dots, c_3$, and
- the fourth line of the state is valid, that is, $v_{12} = v_{13} \oplus c_4 \oplus c_5$ and $v_{14} = v_{15} \oplus c_6 \oplus c_7$

Thus we find a fixed-point with effort $2^{128} \times 2^{64} = 2^{192}$, instead of 2^{256} ideally. This technique also allows to find several fixed-points for a same message (up to 2^{64} per message) in less time than expected for an ideal function.

BLAKE's fixed-point properties do not give a distinguisher between BLAKE and a PRF, because we use here the internal mechanisms of the compression function, and not blackbox queries.

Fixed-point collisions

A fixed-point collision for BLAKE is a tuple (m, m', h, s, s', t, t') such that

$$\textbf{compress}(m, h, s, t) = \textbf{compress}(m', h, s', t') = h,$$

that is, a pair of fixed-points for the same hash value. This notion was introduced in [2], which shows that fixed-point collisions can be used to build multicollisions at a reduced cost. For BLAKE-32, however, a fixed-point collision costs about $2^{192} \times 2^{128} = 2^{320}$ trials, which is too high to exploit for an attack.

5.3 Iteration mode (HAIFA)

HAIFA [10, 23] is a general iteration mode for hash functions, which can be seen as "Merkle-Damgård with a salt and a counter". HAIFA offers an interface for input of the salt and the counter, and provides resistance to several generic attacks (herding, long-message second preimages, length extension). HAIFA was used for the LAKE hash functions [5], and analyzed in [1, 15].

Below we comment on BLAKE's use of HAIFA:

- HAIFA has originally a single IV for a family of functions, and computes the effective IV of a specific instance with k-bit hashes by setting IV $\leftarrow$ **compress**$(\text{IV}, k, 0, 0)$. This allows variable-length hashing, but complicates the function and requires an additional compression. BLAKE has only two different instances for each function, so we directly specify their proper IV to simplify the definition. Each instance has a distinct effective IV, but no extra compression is needed.
- HAIFA defines a padding data that includes the encoding of the hash value length; again, because we only have two different lengths, one bit suffices to encode the identity of the instance (i.e., 1 encodes 256, and 0 encodes 224). We preserve the instance-dependent padding, but reduce the data overhead, and in the best case save one call to the compression function. Padding the binary encoding of the hash bit length wouldn't increase security.

On the role of the counter

We will highlight some facts that underlie HAIFA's resistance to length extension and second preimage attacks. Suppose that **compress**$(\cdot, \cdot, \cdot, t)$ defines a family of pseudorandom functions (PRF's); to make clear the abstraction we'll write $\{F_t\}_t$ the PRF family, such that

$F_t(m, h, s) = h'$, i.e. F is an ideal compression function, and F_t an ideal compression function with counter set to t. In the process of iteratively hashing a message, all compression functions F_t are different, because the counter is different at each compression. For example, when hashing a 1020-bit message with BLAKE-32, we first use F_{512}, then F_{1020}, and finally F_0.

Now observe that the family $\{F_t\}$ can be split into two disjoint sets (considering BLAKE-32's parameters):

1. the *intermediate* compressions, called to compress message blocks containing no padding data (only original message bits):
$$\mathcal{I} = \{F_t, \exists k \in \mathbb{N}^\star, t = 512 \cdot k \leq 2^{64} - 512\}$$
2. the *final* compressions, called to compress message blocks containing padding data:
$$\mathcal{F} = \{F_0\} \cup \{F_t, \exists k \in \mathbb{N}^\star, p \in \{1, \dots, 511\}, t = 512 \cdot k + p < 2^{64}\}$$

A function in $\mathcal{I}$ is never the last in a chain of iterations. A function in $\mathcal{F}$ appears either in last or penultimate position, and its inputs are restricted to message blocks with consistent padding (for example F_{10} in BLAKE-32 needs messages of the form $\langle 10 \text{ bits}\rangle 10\dots01\langle 10\rangle_{64}$). Clearly, $|\mathcal{I}| = 2^{55} - 1$ and $|\mathcal{F}| = 511 \cdot |\mathcal{I}|$. Functions in $\mathcal{F}$ can be seen as playing a role of output filter, in the same spirit as the NMAC hash construction [16].

The above structure is behind the original security properties of HAIFA, including its resistance to second-preimage attacks [23].

5.4 Pseudorandomness

One expects from a good hash function to "look like a random function". Notions of indistinguishability, unpredictability, indifferentiability [31] and seed-incompressibility [25] define precise notions related to "randomness" for hash functions, and are used to evaluate generic constructions or dedicated designs. However they give no clue on how to construct primitives' algorithms.

Roughly speaking, the algorithm of the compression function should simulate a "complicated function", with no apparent structure—i.e., it should have no property that a random function has not. In terms of structure, "complicated" means for example that the algebraic normal form (ANF) of the function, as a vector of Boolean functions, should contain each possible monomial with probability 1/2; generalizing, it means that when any part of the input is random, then the ANF obtained by fixing this input is also (uniform) random. Put differently, the truth table of the hash function when part of the input is random should "look like" a random bit string. In terms of input/output, "complicated" means for example that a small difference in the input doesn't imply a small difference in the input; more generally, any difference or relation between two inputs should be statistically independent of any relation of the corresponding outputs.

Pseudorandomness is particularly critical for stream ciphers, and no distinguishing attack—or any other non-randomness property—has been identified on Salsa20 or ChaCha. These ciphers construct a complicated function by making a long chain of simple operations. Non-randomness was observed for reduced versions with up to three ChaCha rounds (which correspond to one and a half BLAKE round). BLAKE inherits ChaCha's pseudorandomness, and in addition avoids the self-similarity of the function by having round-dependent constants. Although there is no formal reduction of BLAKE's security to ChaCha's, we can reasonably conjecture that BLAKE's compression function is "complicated enough" with respect to pseudorandomness.

5.5 Generic attacks

This section reports on the resistance of BLAKE to the most important generic attacks, that is, attacks that exploit to broad class of functions: for example a generic attack can exploit the iteration mode, or weak algebraic properties of the compression function.

5.5.1 Length extension

Length extension is a forgery attack against MAC's of the form $H_k(m)$ or $H(k\|m)$, i.e. where the key k is respectively used as the IV or prepended to the message. The attack can be applied when H is an iterated hash with "MD-strengthening" padding: given $h = H_k(m)$ and m, determine the padding data p, and compute $v' = H_h(m')$, for an arbitrary m'. It follows from the iterated construction that $v' = H_k(m\|p\|m')$. That is, the adversary forged a MAC of the message $m\|p\|m'$.

The length extension attack doesn't apply to BLAKE, because of the input of the number of bits hashed so far to the compression function, which simulates a specific output function for the last message block (cf. §5.3). For example, let m be a 1020-bit message; after padding, the message is composed of three blocks m^0, m^1, m^2; the final chain value will be $h^3 = \textbf{compress}(h^2, m^2, s, 0)$, because counter values are respectively 512, 1020, and 0 (see §2.1.3). If we extend the message with a block m^3, with convenient padding bits, and hash $m^0\|m^1\|m^2\|m^3$, then the chain value between m^2 and m^3 will be $\textbf{compress}(h^2, m^2, s, 1024)$, and thus be different from $\textbf{compress}(h^2, m^2, s, 0)$. The knowledge of BLAKE-32$(m^0\|m^1\|m^2)$ cannot be used to compute the hash of $m^0\|m^1\|m^2\|m^3$.

5.5.2 Collision multiplication

We coin the term "collision multiplication" to define the ability, given a collision (m, m'), to derive an arbitrary number of other collisions. For example, Merkle-Damgård hash functions allow to derive collisions of the form $(m\|p\|u, m'\|p'\|u)$, where p and p' are the padding data, and u an arbitrary string; this technique can be seen as a kind of length extension attack. And for the same reasons that BLAKE resists length extension, it also resists this type of collision multiplication, when given a collision of minimal size (that is, when the collision only occurs for the hash value, not for intermediate chain values).

5.5.3 Multicollisions

A multicollision is a set of messages that map to the same hash value. We speak of a k-collision when k distinct colliding messages are known.

Joux's technique

The technique proposed by Joux [28] finds a k-collision for Merkle-Damgård hash functions with n-bit hash values in $\lceil \log_2 k \rceil \cdot 2^{n/2}$ calls to the compression function (see Fig. 5.1). The colliding messages are long of $\lceil \log_2 k \rceil$ blocks. This technique applies as well for the BLAKE hash functions, and to all hash functions based on HAIFA. For example, a 32-collision for BLAKE-32 can be found within 2^{133} compressions.

Joux's attack is clearly not a concrete threat, which is demonstrated *ad absurdum*: to be applicable, it requires the knowledge of at least two collisions, but any function (resistant or not

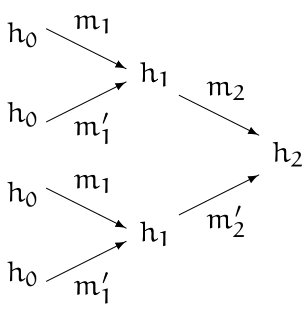

Figure 5.1: Illustration of Joux's technique for 2-collisions, where **compress**(h_0, m_1) = **compress**$(h_0, m'_1) = h_1$, etc. This technique can apply to BLAKE.

to Joux's attack) for which collisions can be found is broken anyway. Hence this attack only damages non-collision-resistant hash functions.

Kelsey/Schneier's technique

The technique presented by Kelsey and Schneier [29] works only when the compression function admits easily found fixed-points. An advantage over Joux's attack is that the cost of finding a k-collision no longer depends on k. Specifically, for a Merkle-Damgård hash function with n-bit hash values, it makes $3 \cdot 2^{n/2}$ compressions and needs storage for $2^{n/2}$ message blocks (see Fig. 5.2). Colliding messages are long of k blocks. This technique does not apply to BLAKE, because fixed-points cannot be found efficiently, and the counter t foils fixed-point repetition.

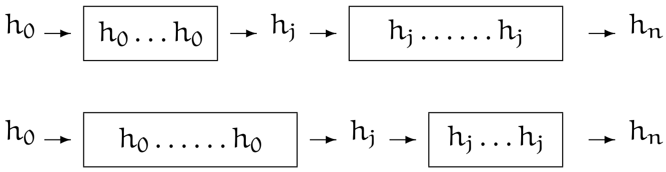

Figure 5.2: Schematic view of the Kelsey/Schneier multicollision attack on Merkle-Damgård functions. This technique does not apply to BLAKE.

Faster multicollisions

When an iterated hash admits fixed-points and the IV is chosen by the attacker, this technique [2] finds a k-collision in time $2^{n/2}$ and negligible memory, with colliding messages of size $\lceil \log_2 k \rceil$ (see Fig. 5.3. Like the Kelsey/Schneier technique, it is based on the repetition of fixed-points, thus does not apply to BLAKE.

5.5.4 Second preimages

Dean [22, §5.6.3] and subsequently Kelsey and Schneier [29] showed generic attacks on n-bit iterated hashes that find second preimages in significantly less than 2^n compressions. HAIFA was proven to be resistant to these attacks [23], assuming a strong compression function; this result applies to BLAKE, as a HAIFA-based design. Therefore, no attack on n-bit BLAKE can find second-preimages in less than 2^n trials, unless exploiting the structure of the compression function.

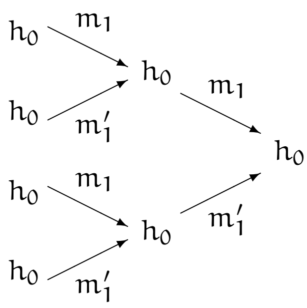

Figure 5.3: Illustration of the faster multicollision, for 2-collisions on Merkle-Damgård hash functions. This technique does not apply to BLAKE.

5.5.5 Side channels

All operations in the BLAKE functions are independent of the input and can be implemented to run in constant time on all platforms (and still be fast). The ChaCha core function was designed to be immune to all kind of side-channel attacks (timing, power analysis, etc.), and BLAKE inherits this property. Side-channel analysis of the eSTREAM finalists also suggests that Salsa20 and ChaCha are immune to side-channel attacks [40].

5.5.6 SAT solvers

Attacks using SAT-solvers consist in describing a security problem in terms of a SAT instance, then solving this instance with an efficient solver. These attacks were used for finding collisions [32] and preimages for (reduced) for MD4 and MD5 [20]. The high complexity of BLAKE and the absence of SAT-solver-based attacks on ChaCha and Salsa20 argues for the resistance of BLAKE to these methods.

5.5.7 Algebraic attacks

Algebraic attacks consist in reducing a security problem to solving a system of equations, then solving this system. The approach is similar to that of SAT-solver attacks, and for similar reasons is unlikely to break BLAKE.

5.6 Dedicated attacks

This section describes several strategies for attacking BLAKE, and justifies their limitations.

5.6.1 Symmetric differences

A sufficient (but not necessary) condition to find a collision on BLAKE is to find two message blocks for which, given same IV's and salts, the corresponding internal states v and v' after the sequence of rounds satisfy the relation

$$v_i \oplus v_{i+8} = v'_i \oplus v'_{i+8},\ i = 0, \dots, 7.$$

Put differently, it suffices to find a message difference that leads after the rounds sequence to a difference of the form

$$\begin{pmatrix} v_0 \oplus v_0' & v_1 \oplus v_1' & v_2 \oplus v_2' & v_3 \oplus v_3' \\ v_4 \oplus v_4' & v_5 \oplus v_5' & v_6 \oplus v_6' & v_7 \oplus v_7' \\ v_8 \oplus v_8' & v_9 \oplus v_9' & v_{10} \oplus v_{10}' & v_{11} \oplus v_{11}' \\ v_{12} \oplus v_{12}' & v_{13} \oplus v_{13}' & v_{14} \oplus v_{14}' & v_{15} \oplus v_{15}' \end{pmatrix} = \begin{pmatrix} \Delta_0 & \Delta_1 & \Delta_2 & \Delta_3 \\ \Delta_4 & \Delta_5 & \Delta_6 & \Delta_7 \\ \Delta_0 & \Delta_1 & \Delta_2 & \Delta_3 \\ \Delta_4 & \Delta_5 & \Delta_6 & \Delta_7 \end{pmatrix}.$$

We say that the state has *symmetric* differences. This condition is not necessary for collisions, because there may exist collisions for different salts.

Birthday attack

A birthday attack on v can be used to find two messages with symmetric differences, that is, a collision for the "top" and "bottom" differences. Since for each pair of messages the collision occurs with probability 2^{-256}, a birthday attack requires about 2^{128} messages. This approach is likely to be a bit faster than a direct birthday attack on the hash function, because here one never computes the finalization of the compression function. The attack may be improved if one finds message differences that give, for example, $v_0 \oplus v_0' = v_8 \oplus v_8'$ with probability noticeably higher than 2^{-32} (for BLAKE-32). Such correlations between differences are however very unlikely with the recommended number of rounds.

Backward attack

One can pick two random v and v' having symmetric differences, and compute rounds backward for two arbitrary distinct messages. In the end the initial states obtained need

1. to have an IV and salt satisfying $h_i \oplus s_{i \bmod 4} = h_i' \oplus s_{i \bmod 4}'$, for $i = 0, \dots, 7$, which occurs with probability 2^{-256}
2. to be valid initial states for a counter $0 < t \leq 512$, which occurs with probability 2^{-128}

Using a birthday strategy, running this attack requires about 2^{256} trials, and finds collisions with different IV's and different salts. If we allow different counters of arbitrary values, then the initial state obtained is valid with probability 2^{-64}, and the attacks runs within $2^{128} \times 2^{64} = 2^{192}$ trials, which is still slower than a direct birthday attack.

5.6.2 Differential attack

BLAKE functions can be attacked if one finds a message difference that gives certain output difference with significantly higher probability than ideally expected. A typical differential attack uses high-probability differentials for the sequence of round functions. An argument against the existence of such differentials is that BLAKE's round function is essentially ChaCha's "double-round", whose differential behavior has been intensively studied without real success; in [3].

Attacks on ChaCha are based on the existence of truncated differentials after three steps (that is, one and a half BLAKE round) [3]. These differentials have a 1-bit input difference and a 1-bit output difference; namely, flipping certain bits gives non-negligible biases in certain output bits. No truncated differential was found through four steps (two BLAKE rounds). This suggests that differentials in BLAKE with input difference in the IV or the salt cannot be found for more than two rounds. An input difference in the message spreads even more, because the difference affects the state through each round of the function.

Rumba [8] is a compression function based on the stream cipher Salsa20; contrary to BLAKE, the message is put in the initial state and no data is input during the rounds iteration. Attacks on Rumba in [3] are based on the identification of a linear approximation through three steps, and the use of message modification techniques to increase the probability of finding compliant messages. Rumba is based on Salsa20, not on ChaCha, and thus such differentials are likely to have much lower probability with ChaCha. With its ten rounds (20 steps), BLAKE is very unlikely to be attacked with such techniques.

5.6.3 Slide attack

Slide attacks were originally proposed to attack block ciphers [11,12], and recently were applied in some sense to hash functions [38]. Here we show how to apply the idea to attack a modified variant of BLAKE's compression function.

Suppose all the permutations σ_i are equal (to, say, the identity). Then for a message such that $m_0 = \cdots = m_{15}$, the sequence of rounds is a repeated application of the same permutation on the internal state, because for each G_i, the value $(m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$ is now independent of the round index r. The idea of the attack is to use 256 bits of freedom of the message to have, after one round, an internal state v' such that $h_i \oplus s_{i \bmod 4} = h'_i \oplus s'_{i \bmod 4}$, for h' and s' derived from v' according to the initialization rule. The state obtained will be valid with probability 2^{-64}. Then, for the same message and the $(r-1)$-round function, we get a collision after the finalization process, with different IV, salt, and counter. Runtime is 2^{64} trials, to find collisions with two different versions of the compression function. For the full version (with nontrivial permutations), this attack cannot work for more than two rounds.

6 Acknowledgments

We thank Dan Bernstein for allowing us to build on Chacha's design. We also thank Florian Mendel and Martin Schläffer for their analysis of BLAKE's predecessor LAKE, and Orr Dunkelman for communicating us new results on HAIFA. Finally, we thank Peter Steigmeier for implementing BLAKE on the 8-bit PIC processor.

Jean-Philippe Aumasson is supported by the Swiss National Science Foundation under project no. 113329. Willi Meier is supported by GEBERT RÜF STIFTUNG under project no. GRS-069/07.

Bibliography

[1] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In *ASIACRYPT*, 2007.

[2] Jean-Philippe Aumasson. Faster multicollisions. In *INDOCRYPT 2008*, 2008.

[3] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of Latin dances: analysis of Salsa, ChaCha, and Rumba. In *FSE*, 2008.

[4] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage attacks on 3-pass HAVAL and step-reduced MD5. In *SAC*, 2008.

[5] Jean-Philippe Aumasson, Willi Meier, and Raphael C.-W. Phan. The hash function family LAKE. In *FSE*, 2008.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[7] Daniel J. Bernstein. ChaCha, a variant of Salsa20. `cr.yp.to/chacha.html`.

[8] Daniel J. Bernstein. The Rumba20 compression function. `cr.yp.to/rumba20.html`.

[9] Daniel J. Bernstein. Salsa20. Technical Report 2005/25, ECRYPT eSTREAM, 2005. `cr.yp.to/snuffle.html`.

[10] Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. ePrint report 2007/278, 2007.

[11] Alex Biryukov and David Wagner. Slide attacks. In *FSE*, 1999.

[12] Alex Biryukov and David Wagner. Advanced slide attacks. In *EUROCRYPT*, 2000.

[13] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO*, 1999.

[14] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: A new high-performance stream cipher. In *FSE*, 2003.

[15] Charles Bouillaguet, Pierre-Alain Fouque, Adi Shamir, and Sébastien Zimmer. Second preimage attacks on dithered hash functions. ePrint report 2007/395, 2007.

[16] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *CRYPTO*, 2005.

[17] Paul Crowley. Truncated differential cryptanalysis of five rounds of Salsa20. In *SASC 2006*. ECRYPT, 2006.

[18] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In *FSE*, 1997.

[19] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, 2000.

[20] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using satSolvers. In *SAT*, 2007.

[21] Christophe de Cannière and Christian Rechberger. Preimage attacks for (reduced) SHA-0 and SHA-1. In *CRYPTO*, 2008.

[22] Richard Drews Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, 1999.

[23] Orr Dunkelman. Re-visiting HAIFA. Talk at the workshop *Hash functions in cryptology: theory and practice*, 2008.

[24] Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In *CRYPTO*, 2006.

[25] Shai Halevi, Steven Myers, and Charles Rackoff. On seed-incompressible functions. In *TCC*, 2008.

[26] Luca Henzen, Flavio Carbognani, Norbert Felber, and Wolfgang Fichtner. VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha, and the compression function Rumba. In *IEEE International Conference on Signals, Circuits and Systems (SCS'08)*, 2008.

[27] Julio Cesar Hernandez-Castro, Juan M. E. Tapiador, and Jean-Jacques Quisquater. On the Salsa20 hash function. In *FSE*, 2008.

[28] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[29] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *EUROCRYPT*, 2005.

[30] Stefan Lucks. A failure-friendly design principle for hash functions. In *ASIACRYPT*, 2005.

[31] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, 2004.

[32] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *SAT*, 2006.

[33] Ivica Nikolic and Alex Biryukov. Collisions for step-reduced SHA-256. In *FSE*, 2008.

[34] NIST. FIPS 180-2 secure hash standard, 2002.

[35] NIST. FIPS 198 the keyed-hash message authentication code, 2002.

[36] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(112), November 2007.

[37] NIST. SP 800-106, randomized hashing digital signatures, 2007.

[38] Thomas Peyrin. Security analysis of extended sponge functions. Talk at the workshop *Hash functions in cryptology: theory and practice*, 2008.

[39] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential cryptanalysis of Salsa20/8. In *SASC 2007*. ECRYPT, 2007.

[40] Erik Zenner. Cache timing analysis of HC-256. In *SASC 2008 – The State of the Art of Stream Ciphers*. ECRYPT, 2008.

A Round function example

We give an example of computation by the BLAKE-32 round function.

At the first round $G_0(v_0, v_4, v_8, v_{12})$ computes

$$
\begin{aligned}
v_0 &\leftarrow v_0 + v_4 + (m_0 \oplus \texttt{85A308D3}) \\
v_{12} &\leftarrow (v_{12} \oplus v_0) \ggg 16 \\
v_8 &\leftarrow v_8 + v_{12} \\
v_4 &\leftarrow (v_4 \oplus v_8) \ggg 12 \\
v_0 &\leftarrow v_0 + v_4 + (m_1 \oplus \texttt{243F6A88}) \\
v_{12} &\leftarrow (v_{12} \oplus v_0) \ggg 8 \\
v_8 &\leftarrow v_8 + v_{12} \\
v_4 &\leftarrow (v_4 \oplus v_8) \ggg 7
\end{aligned}
$$

where $\texttt{85A308D3} = c_{\sigma_0(2\times 0+1)} = c_1$ and $\texttt{243F6A88} = c_{\sigma_0(2\times 0)} = c_0$.

Then $G_1(v_1, v_5, v_9, v_{13})$ computes

$$
\begin{aligned}
v_1 &\leftarrow v_1 + v_5 + (m_2 \oplus \texttt{03707344}) \\
v_{13} &\leftarrow (v_{13} \oplus v_1) \ggg 16 \\
v_9 &\leftarrow v_9 + v_{13} \\
v_5 &\leftarrow (v_5 \oplus v_9) \ggg 12 \\
v_1 &\leftarrow v_1 + v_5 + (m_3 \oplus \texttt{13198A2E}) \\
v_{13} &\leftarrow (v_{13} \oplus v_1) \ggg 8 \\
v_9 &\leftarrow v_9 + v_{13} \\
v_5 &\leftarrow (v_5 \oplus v_9) \ggg 7
\end{aligned}
$$

and so on until $G_7(v_3, v_4, v_9, v_{14})$, which computes

$$
\begin{aligned}
v_3 &\leftarrow v_3 + v_4 + (m_{14} \oplus \texttt{B5470917}) \\
v_{14} &\leftarrow (v_{14} \oplus v_3) \ggg 16 \\
v_9 &\leftarrow v_9 + v_{14} \\
v_4 &\leftarrow (v_4 \oplus v_9) \ggg 12 \\
v_3 &\leftarrow v_3 + v_4 + (m_{15} \oplus \texttt{3F84D5B5}) \\
v_{14} &\leftarrow (v_{14} \oplus v_3) \ggg 8 \\
v_9 &\leftarrow v_9 + v_{14} \\
v_4 &\leftarrow (v_4 \oplus v_9) \ggg 7
\end{aligned}
$$

After $G_7(v_3, v_4, v_9, v_{14})$, the second round starts. Because of the round-dependent permutations, $G_0(v_0, v_4, v_8, v_{12})$ now uses the permutation σ_1 instead of σ_0, and thus computes

$$\begin{aligned}
v_0 &\leftarrow v_0 + v_4 + (m_{14} \oplus \texttt{BE5466CF}) \\
v_{12} &\leftarrow (v_{12} \oplus v_0) \ggg 16 \\
v_8 &\leftarrow v_8 + v_{12} \\
v_4 &\leftarrow (v_4 \oplus v_8) \ggg 12 \\
v_0 &\leftarrow v_0 + v_4 + (m_{10} \oplus \texttt{3F84D5B5}) \\
v_{12} &\leftarrow (v_{12} \oplus v_0) \ggg 8 \\
v_8 &\leftarrow v_8 + v_{12} \\
v_4 &\leftarrow (v_4 \oplus v_8) \ggg 7
\end{aligned}$$

Above, $14 = \sigma_1(2\times 0) = \sigma_1(0)$, $10 = \sigma_1(2\times 0+1) = \sigma_1(1)$, $\texttt{BE5466CF} = c_{10}$, and $\texttt{3F84D5B5} = c_{14}$. Applying similar rules, column steps and diagonal steps continue until the tenth round, which uses the permutation σ_9.

B Source code

B.1 VHDL

We give our VHDL code computing the compression function of BLAKE-32 with the [8G] architecture. We split the implementation into 7 `vhd` files: `blake32`, `blake32Pkg`, `initialization`, `roundreg`, `gcomp`, `finalization`, and `controller`:

File `blake32.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;

entity blake32 is
    port (
      CLKxCI : in std_logic;
      RSTxRBI : in std_logic;
      MxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
      HxDI : in std_logic_vector(WWIDTH*8-1 downto 0);
      SxDI : in std_logic_vector(WWIDTH*4-1 downto 0);
      TxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
      HxDO : out std_logic_vector(WWIDTH*8-1 downto 0);
      InENxSI : in std_logic;
      OutENxSO : out std_logic
      );
end blake32;

architecture hash of blake32 is
    component controller
      port (
        CLKxCI : in std_logic;
        RSTxRBI : in std_logic;
        VALIDINxSI : in std_logic;
        VALIDOUTxSO : out std_logic;
        ROUNDxSO : out unsigned(3 downto 0)
        );
    end component;

    component initialization
      port (
        HxDI : in std_logic_vector(WWIDTH*8-1 downto 0);
        SxDI : in std_logic_vector(WWIDTH*4-1 downto 0);
        TxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
        VxDO : out std_logic_vector(WWIDTH*16-1 downto 0)
        );
    end component;

    component roundreg
```

```
    port (
      CLKxCI : in std_logic;
      RSTxRBI : in std_logic;
      WEIxSI : in std_logic;
      ROUNDxSI : in unsigned(3 downto 0);
      VxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
      MxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
      VxDO : out std_logic_vector(WWIDTH*16-1 downto 0)
      );
  end component;

  component finalization
    port (
      VxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
      HxDI : in std_logic_vector(WWIDTH*8-1 downto 0);
      SxDI : in std_logic_vector(WWIDTH*4-1 downto 0);
      HxDO : out std_logic_vector(WWIDTH*8-1 downto 0)
      );
  end component;

  signal VxD, VFINALxD : std_logic_vector(WWIDTH*16-1 downto 0);
  signal ROUNDxS : unsigned(3 downto 0);

begin -- hash
  -----------------------------------------------------------------------------
  -- CONTROLLER
  -----------------------------------------------------------------------------
  u_controller:  controller
    port map (
      CLKxCI => CLKxCI,
      RSTxRBI => RSTxRBI,
      VALIDINxSI => InENxSI,
      VALIDOUTxSO => OutENxSO,
      ROUNDxSO => ROUNDxS
      );

  -----------------------------------------------------------------------------
  -- INITIALIZATION
  -----------------------------------------------------------------------------
  u_initialization:  initialization
    port map (
      HxDI => HxDI,
      SxDI => SxDI,
      TxDI => TxDI,
      VxDO => VxD
      );

  -----------------------------------------------------------------------------
  -- ROUND
  -----------------------------------------------------------------------------
  u_roundreg:  roundreg
    port map (
      CLKxCI => CLKxCI,
      RSTxRBI => RSTxRBI,
      WEIxSI => InENxSI,
      ROUNDxSI => ROUNDxS,
      VxDI => VxD,
      MxDI => MxDI,
      VxDO => VFINALxD
      );

  -----------------------------------------------------------------------------
  -- FINALIZATION
```

```
    ----------------------------------------------------------------------------
    u_finalization:  finalization
      port map (
        VxDI => VFINALxD,
        HxDI => HxDI,
        SxDI => SxDI,
        HxDO => HxDO
        );

end hash;
```

File blake32Pkg.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

package blake32Pkg is

    constant WWIDTH : integer := 32; -- WORD WIDTH
    constant NROUND : integer := 10; -- ROUND NUMBER

    ----------------------------------------------------------------------------
    -- c Constants
    ----------------------------------------------------------------------------
    type c_const is array (0 to 15) of std_logic_vector(31 downto 0);

    constant C : c_const :=  ((x"243F6A88"), (x"85A308D3"),
                              (x"13198A2E"), (x"03707344"),
                              (x"A4093822"), (x"299F31D0"),
                              (x"082EFA98"), (x"EC4E6C89"),
                              (x"452821E6"), (x"38D01377"),
                              (x"BE5466CF"), (x"34E90C6C"),
                              (x"C0AC29B7"), (x"C97C50DD"),
                              (x"3F84D5B5"), (x"B5470917"));

    ----------------------------------------------------------------------------
    -- o Permutations
    ----------------------------------------------------------------------------
    type perm is array (0 to 9, 0 to 15) of integer;

    constant PMATRIX : perm :=  ((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),
                                 (13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10),
                                 (6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5),
                                 (10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0));

end blake32Pkg;
```

File initialization.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;
```

```
entity initialization is
  port (
    HxDI : in std_logic_vector(WWIDTH*8-1 downto 0);
    SxDI : in std_logic_vector(WWIDTH*4-1 downto 0);
    TxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
    VxDO : out std_logic_vector(WWIDTH*16-1 downto 0)
    );
end initialization;

architecture hash of initialization is

begin -- hash

  VxDO(WWIDTH*16-1 downto WWIDTH*8) <= HxDI;

  VxDO(WWIDTH*8-1 downto WWIDTH*7) <= SxDI(WWIDTH*4-1 downto WWIDTH*3) xor C(0);
  VxDO(WWIDTH*7-1 downto WWIDTH*6) <= SxDI(WWIDTH*3-1 downto WWIDTH*2) xor C(1);
  VxDO(WWIDTH*6-1 downto WWIDTH*5) <= SxDI(WWIDTH*2-1 downto WWIDTH) xor C(2);
  VxDO(WWIDTH*5-1 downto WWIDTH*4) <= SxDI(WWIDTH-1 downto 0) xor C(3);

  VxDO(WWIDTH*4-1 downto WWIDTH*3) <= TxDI(WWIDTH*2-1 downto WWIDTH) xor C(4);
  VxDO(WWIDTH*3-1 downto WWIDTH*2) <= TxDI(WWIDTH*2-1 downto WWIDTH) xor C(5);
  VxDO(WWIDTH*2-1 downto WWIDTH) <= TxDI(WWIDTH-1 downto 0) xor C(6);
  VxDO(WWIDTH-1 downto 0) <= TxDI(WWIDTH-1 downto 0) xor C(7);

end hash;
```

File roundreg.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;

entity roundreg is
  port (
    CLKxCI : in std_logic;
    RSTxRBI : in std_logic;
    WEIxSI : in std_logic;
    ROUNDxSI : in unsigned(3 downto 0);
    VxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
    MxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
    VxDO : out std_logic_vector(WWIDTH*16-1 downto 0)
    );
end roundreg;

architecture hash of roundreg is

  component gcomp
    port (
      AxDI : in std_logic_vector(WWIDTH-1 downto 0);
      BxDI : in std_logic_vector(WWIDTH-1 downto 0);
      CxDI : in std_logic_vector(WWIDTH-1 downto 0);
      DxDI : in std_logic_vector(WWIDTH-1 downto 0);
      MxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
      KxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
      AxDO : out std_logic_vector(WWIDTH-1 downto 0);
      BxDO : out std_logic_vector(WWIDTH-1 downto 0);
      CxDO : out std_logic_vector(WWIDTH-1 downto 0);
      DxDO : out std_logic_vector(WWIDTH-1 downto 0)
      );
  end component;
```

```
    type SUBT16 is array (15 downto 0) of std_logic_vector(WWIDTH-1 downto 0);

    signal VxDN, VxDP, MxD : SUBT16;

    signal G0MxD, G0KxD, G4MxD, G4KxD : std_logic_vector(WWIDTH*2-1 downto 0);
    signal G1MxD, G1KxD, G5MxD, G5KxD : std_logic_vector(WWIDTH*2-1 downto 0);
    signal G2MxD, G2KxD, G6MxD, G6KxD : std_logic_vector(WWIDTH*2-1 downto 0);
    signal G3MxD, G3KxD, G7MxD, G7KxD : std_logic_vector(WWIDTH*2-1 downto 0);

    signal G0AOxD, G0BOxD, G0COxD, G0DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G1AOxD, G1BOxD, G1COxD, G1DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G2AOxD, G2BOxD, G2COxD, G2DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G3AOxD, G3BOxD, G3COxD, G3DOxD : std_logic_vector(WWIDTH-1 downto 0);

    signal G4AOxD, G4BOxD, G4COxD, G4DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G5AOxD, G5BOxD, G5COxD, G5DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G6AOxD, G6BOxD, G6COxD, G6DOxD : std_logic_vector(WWIDTH-1 downto 0);
    signal G7AOxD, G7BOxD, G7COxD, G7DOxD : std_logic_vector(WWIDTH-1 downto 0);

begin -- hash

    p_unform:  for i in 15 downto 0 generate
      MxD(15-i) <= MxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
    end generate p_unform;

    VxDO <= VxDP(0) &  VxDP(1) & VxDP(2) & VxDP(3) &
              VxDP(4) & VxDP(5) & VxDP(6) & VxDP(7) &
              VxDP(8) & VxDP(9) & VxDP(10) & VxDP(11) &
              VxDP(12) & VxDP(13) & VxDP(14) & VxDP(15);

    -----------------------------------------------------------------------------
    -- MEMORY INPUTS
    -----------------------------------------------------------------------------
    p_inmem:  process ( G4AOxD, G4BOxD, G4COxD, G4DOxD, G5AOxD, G5BOxD, G5COxD,
                        G5DOxD, G6AOxD, G6BOxD, G6COxD, G6DOxD, G7AOxD, G7BOxD,
                        G7COxD, G7DOxD, VxDI, VxDP, WEIxSI)
    begin -- process p_inmem

      VxDN <= VxDP;

      if WEIxSI = '1' then
        for i in 15 downto 0 loop
          VxDN(15-i) <= VxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);

        end loop;

      else
        VxDN(0) <= G4AOxD;
        VxDN(5) <= G4BOxD;
        VxDN(10) <= G4COxD;
        VxDN(15) <= G4DOxD;

        VxDN(1) <= G5AOxD;
        VxDN(6) <= G5BOxD;
        VxDN(11) <= G5COxD;
        VxDN(12) <= G5DOxD;

        VxDN(2) <= G6AOxD;
        VxDN(7) <= G6BOxD;
        VxDN(8) <= G6COxD;
        VxDN(13) <= G6DOxD;

        VxDN(3) <= G7AOxD;
        VxDN(4) <= G7BOxD;
        VxDN(9) <= G7COxD;
        VxDN(14) <= G7DOxD;

      end if;
```

```
end process p_inmem;

-------------------------------------------------------------------------------
-- G INPUTS
-------------------------------------------------------------------------------
p_outmem:  process (MxD, ROUNDxSI)
begin -- process p_outmem

  G0MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 0)) & MxD(PMATRIX(to_integer(ROUNDxSI), 1));
  G1MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 2)) & MxD(PMATRIX(to_integer(ROUNDxSI), 3));
  G2MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 4)) & MxD(PMATRIX(to_integer(ROUNDxSI), 5));
  G3MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 6)) & MxD(PMATRIX(to_integer(ROUNDxSI), 7));
  G4MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 8)) & MxD(PMATRIX(to_integer(ROUNDxSI), 9));
  G5MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 10)) & MxD(PMATRIX(to_integer(ROUNDxSI), 11));
  G6MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 12)) & MxD(PMATRIX(to_integer(ROUNDxSI), 13));
  G7MxD <= MxD(PMATRIX(to_integer(ROUNDxSI), 14)) & MxD(PMATRIX(to_integer(ROUNDxSI), 15));

  G0KxD <= C(PMATRIX(to_integer(ROUNDxSI), 1)) & C(PMATRIX(to_integer(ROUNDxSI), 0));
  G1KxD <= C(PMATRIX(to_integer(ROUNDxSI), 3)) & C(PMATRIX(to_integer(ROUNDxSI), 2));
  G2KxD <= C(PMATRIX(to_integer(ROUNDxSI), 5)) & C(PMATRIX(to_integer(ROUNDxSI), 4));
  G3KxD <= C(PMATRIX(to_integer(ROUNDxSI), 7)) & C(PMATRIX(to_integer(ROUNDxSI), 6));
  G4KxD <= C(PMATRIX(to_integer(ROUNDxSI), 9)) & C(PMATRIX(to_integer(ROUNDxSI), 8));
  G5KxD <= C(PMATRIX(to_integer(ROUNDxSI), 11)) & C(PMATRIX(to_integer(ROUNDxSI), 10));
  G6KxD <= C(PMATRIX(to_integer(ROUNDxSI), 13)) & C(PMATRIX(to_integer(ROUNDxSI), 12));
  G7KxD <= C(PMATRIX(to_integer(ROUNDxSI), 15)) & C(PMATRIX(to_integer(ROUNDxSI), 14));

end process p_outmem;

-------------------------------------------------------------------------------
-- G BLOCKS
-------------------------------------------------------------------------------
u_gcomp0:  gcomp
  port map (
    AxDI => VxDP(0), BxDI => VxDP(4), CxDI => VxDP(8), DxDI => VxDP(12), MxDI => G0MxD,
    KxDI => G0KxD, AxDO => G0AOxD, BxDO => G0BOxD, CxDO => G0COxD, DxDO => G0DOxD
    );

u_gcomp1:  gcomp
  port map (
    AxDI => VxDP(1), BxDI => VxDP(5), CxDI => VxDP(9), DxDI => VxDP(13), MxDI => G1MxD,
    KxDI => G1KxD, AxDO => G1AOxD, BxDO => G1BOxD, CxDO => G1COxD, DxDO => G1DOxD
    );

u_gcomp2:  gcomp
  port map (
    AxDI => VxDP(2), BxDI => VxDP(6), CxDI => VxDP(10), DxDI => VxDP(14), MxDI => G2MxD,
    KxDI => G2KxD, AxDO => G2AOxD, BxDO => G2BOxD, CxDO => G2COxD, DxDO => G2DOxD
    );

u_gcomp3:  gcomp
  port map (
    AxDI => VxDP(3), BxDI => VxDP(7), CxDI => VxDP(11), DxDI => VxDP(15), MxDI => G3MxD,
    KxDI => G3KxD, AxDO => G3AOxD, BxDO => G3BOxD, CxDO => G3COxD, DxDO => G3DOxD
    );

-----------------------

u_gcomp4:  gcomp
  port map (
    AxDI => G0AOxD, BxDI => G1BOxD, CxDI => G2COxD, DxDI => G3DOxD, MxDI => G4MxD,
    KxDI => G4KxD, AxDO => G4AOxD, BxDO => G4BOxD, CxDO => G4COxD, DxDO => G4DOxD
    );

u_gcomp5:  gcomp
  port map (
    AxDI => G1AOxD, BxDI => G2BOxD, CxDI => G3COxD, DxDI => G0DOxD, MxDI => G5MxD,
```

```
      KxDI => G5KxD, AxDO => G5AOxD, BxDO => G5BOxD, CxDO => G5COxD, DxDO => G5DOxD
      );

  u_gcomp6: gcomp
    port map (
      AxDI => G2AOxD, BxDI => G3BOxD, CxDI => G0COxD, DxDI => G1DOxD, MxDI => G6MxD,
      KxDI => G6KxD, AxDO => G6AOxD, BxDO => G6BOxD, CxDO => G6COxD, DxDO => G6DOxD
      );

  u_gcomp7: gcomp
    port map (
      AxDI => G3AOxD, BxDI => G0BOxD, CxDI => G1COxD, DxDI => G2DOxD, MxDI => G7MxD,
      KxDI => G7KxD, AxDO => G7AOxD, BxDO => G7BOxD, CxDO => G7COxD, DxDO => G7DOxD
      );

  -----------------------------------------------------------------------------
  -- v MEMORY
  -----------------------------------------------------------------------------
  p_mem: process (CLKxCI, RSTxRBI)
  begin -- process p_vmem
    if RSTxRBI = '0' then -- asynchronous reset (active low)
      VxDP <= (others => (others => '0'));

    elsif CLKxCI'event and CLKxCI = '1' then -- rising clock edge
      VxDP <= VxDN;

    end if;
  end process p_mem;

end hash;
```

File gcomp.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;

entity gcomp is
  port (
    AxDI : in std_logic_vector(WWIDTH-1 downto 0);
    BxDI : in std_logic_vector(WWIDTH-1 downto 0);
    CxDI : in std_logic_vector(WWIDTH-1 downto 0);
    DxDI : in std_logic_vector(WWIDTH-1 downto 0);
    MxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
    KxDI : in std_logic_vector(WWIDTH*2-1 downto 0);
    AxDO : out std_logic_vector(WWIDTH-1 downto 0);
    BxDO : out std_logic_vector(WWIDTH-1 downto 0);
    CxDO : out std_logic_vector(WWIDTH-1 downto 0);
    DxDO : out std_logic_vector(WWIDTH-1 downto 0)
    );
end gcomp;

architecture hash of gcomp is

  signal T1, T4, T7, T10 : unsigned(WWIDTH-1 downto 0);
  signal T2, T3, T5, T6 : std_logic_vector(WWIDTH-1 downto 0);
  signal T8, T9, T11, T12 : std_logic_vector(WWIDTH-1 downto 0);
  signal TK1, TK2 : std_logic_vector(WWIDTH-1 downto 0);

begin -- hash

  TK1 <= MxDI(WWIDTH*2-1 downto WWIDTH) xor KxDI(WWIDTH*2-1 downto WWIDTH);
  T1 <= unsigned(AxDI) + unsigned(BxDI) + unsigned(TK1);
```

```
    T2 <= std_logic_vector(T1) xor DxDI;
    T3 <= T2(15 downto 0) & T2(WWIDTH-1 downto 16);

    T4 <= unsigned(CxDI) + unsigned(T3);
    T5 <= std_logic_vector(T4) xor BxDI;
    T6 <= T5(11 downto 0) & T5(WWIDTH-1 downto 12);

    -------------------------------------------------------------------------

    TK2 <= MxDI(WWIDTH-1 downto 0) xor KxDI(WWIDTH-1 downto 0);
    T7 <= T1 + unsigned(T6) + unsigned(TK2);
    T8 <= std_logic_vector(T7) xor T3;
    T9 <= T8(7 downto 0) & T8(WWIDTH-1 downto 8);

    T10 <= T4 + unsigned(T9);
    T11 <= std_logic_vector(T10) xor T6;
    T12 <= T11(6 downto 0) & T11(WWIDTH-1 downto 7);

    AxDO <= std_logic_vector(T7);
    BxDO <= T12;
    CxDO <= std_logic_vector(T10);
    DxDO <= T9;

end hash;
```

File finalization.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;

entity finalization is
    port (
      VxDI : in std_logic_vector(WWIDTH*16-1 downto 0);
      HxDI : in std_logic_vector(WWIDTH*8-1 downto 0);
      SxDI : in std_logic_vector(WWIDTH*4-1 downto 0);
      HxDO : out std_logic_vector(WWIDTH*8-1 downto 0)
      );
end finalization;

architecture hash of finalization is

    type SUB4 is array (3 downto 0) of std_logic_vector(WWIDTH-1 downto 0);
    type SUB8 is array (7 downto 0) of std_logic_vector(WWIDTH-1 downto 0);
    type SUB16 is array (15 downto 0) of std_logic_vector(WWIDTH-1 downto 0);

    signal SINxD : SUB4;
    signal HINxD, HOUTxD : SUB8;
    signal VxD : SUB16;

begin -- hash

    p_unform4:  for i in 0 to 3 generate
      SINxD(i) <= SxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
    end generate p_unform4;

    p_unform8:  for i in 0 to 7 generate
      HINxD(i) <= HxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
      HxDO(WWIDTH*(i+1)-1 downto WWIDTH*i) <= HOUTxD(i);
    end generate p_unform8;

    p_unform16:  for i in 0 to 15 generate
      VxD(i) <= VxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
    end generate p_unform16;
```

```
    HOUTxD(0) <= HINxD(0) xor VxD(0) xor VxD(8) xor SINxD(0);
    HOUTxD(1) <= HINxD(1) xor VxD(1) xor VxD(9) xor SINxD(1);
    HOUTxD(2) <= HINxD(2) xor VxD(2) xor VxD(10) xor SINxD(2);
    HOUTxD(3) <= HINxD(3) xor VxD(3) xor VxD(11) xor SINxD(3);
    HOUTxD(4) <= HINxD(4) xor VxD(4) xor VxD(12) xor SINxD(0);
    HOUTxD(5) <= HINxD(5) xor VxD(5) xor VxD(13) xor SINxD(1);
    HOUTxD(6) <= HINxD(6) xor VxD(6) xor VxD(14) xor SINxD(2);
    HOUTxD(7) <= HINxD(7) xor VxD(7) xor VxD(15) xor SINxD(3);

end hash;
```

File `controller.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.blake32Pkg.all;

    entity controller is
    port (
      CLKxCI : in std_logic;
      RSTxRBI : in std_logic;
      VALIDINxSI : in std_logic;
      VALIDOUTxSO : out std_logic;
      ROUNDxSO : out unsigned(3 downto 0)
      );
end controller;

architecture hash of controller is

    type state is (idle, round, fin);

    signal STATExDP, STATExDN : state;
    signal ROUNDxDP, ROUNDxDN : unsigned(3 downto 0);

begin -- hash

    ROUNDxSO <= ROUNDxDP;

    fsm:  process (ROUNDxDP, STATExDP, VALIDINxSI)
    begin -- process fsm

      VALIDOUTxSO <= '0';
      ROUNDxDN <= (others => '0');

      case STATExDP is

        -----------------------------------------------------------------------
        when idle =>

          if VALIDINxSI = '1' then
            STATExDN <= round;

          else
            STATExDN <= idle;

          end if;

        -----------------------------------------------------------------------
        when round =>

          if ROUNDxDP < NROUND-1 then
            ROUNDxDN <= ROUNDxDP + 1;
            STATExDN <= round;

          else
```

```
          STATExDN <= fin;

        end if;
      ------------------------------------------------------------------------
      when fin =>

        VALIDOUTxSO <= '1';
        STATExDN <= idle;

      ------------------------------------------------------------------------
      when others =>

        STATExDN <= idle;

      end case;

  end process fsm;

  process (CLKxCI, RSTxRBI)
  begin -- process
    if RSTxRBI = '0' then -- asynchronous reset (active low)
      STATExDP <= idle;
      ROUNDxDP <= (others => '0');

    elsif CLKxCI'event and CLKxCI = '1' then -- rising clock edge
      STATExDP <= STATExDN;
      ROUNDxDP <= ROUNDxDN;

      end if;
  end process;

end hash;
```

B.2 PIC assembly

We give our assembly code computing the round function of BLAKE-32.

```
                                            ; round function of BLAKE32
                                            ; indirect adress register FSR0 used for accessing m
                                            ; FSR1 used for accessing c
     do_Gi
                    clrf FSR1H              ; stays zero al the time
                                            ; only lower adress range is used for cts address
                    movlw h'01'             ; table m starts at equ H'110'
                    movWF FSR0H             ; so using FSR0 we need to set highbyte correct
                    movFF i,pointer2mc      ; use i
                    bcf STATUS, C           ; prepare CARRYbit for *2
                    rlcF pointer2mc         ; 2*i
                    movF pointer2mc         ; load pointer into w
                    addWF r,w               ; ADD r (permutation offset in table)
                    movWF pointer2mc        ; ..save it back, is now r(2i)
                    movlw high permut_table_m  ; ..and use it here to find adress of current m
                    movwf TBLPTRH
                    rlncf pointer2mc, w
                    movwf TBLPTRL
                    tblrd*                  ; table read here into TABLAT
                    movff TABLAT, FSR0L     ; move adress to pointer
                    movFF INDF0,tmpXOR_lo   ; access content of m signum_r(2i) low byte loaded
                    movFF PREINC0,tmpXOR_ml ; preincrement pointer, access midlowbyte
                    movFF PREINC0,tmpXOR_mh ; preincrement pointer, access midhighbyte
                    movFF PREINC0,tmpXOR_hi ; preincrement pointer, access highbyte
```

```
term_a1_lowbyte
                    incF pointer2mc            ; pointer now (2i+1)
                    movF pointer2mc            ; load pointer into w
                    movlw high permut_table_c  ; find c signum_r (2i+1)lowbyte adress
                    movwf TBLPTRH
                    rlncf pointer2mc, w
                    movwf TBLPTRL
                    tblrd*                     ; table read here into TABLAT
                    movff TABLAT, FSR1L        ; move adress to pointer

                    movF INDF1                 ; content of c signum_r(2i+1) now in working reg
                    xorWF tmpXOR_lo,w          ; lowest byte [m signum_r (2i) XOR c signum_r (2i+1)]

                    addWFC b_lo,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_ml             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

                    addWFC a_lo,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_ml             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

term_a1_midlowbyte
                    movF PREINC1               ; content of c signum_r (2i+1) midlow byte loaded in w
                    xorWF tmpXOR_ml,w          ; midlow byte [m signum_r (2i) XOR c signum_r (2i+1)]
                    addWFC b_ml,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry


                    addWFC a_ml,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

term_a1_midhighbyte
                    movF PREINC1               ; content of c signum_r (2i+1) midhigh byte loaded in w
                    xorWF tmpXOR_mh,w          ; midhigh byte [m signum (2i) XOR c signum (2i+1)]
                    addWFC b_mh,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

                    addWFC a_mh,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry
```

```
term_a1_highbyte
                 movF PREINC1               ; content of c signum_r (2i+1) high byte loaded in w
                 xorWF tmpXOR_hi,w          ; highest byte [m signum (2i) XOR c signum (2i+1)]
                 addWFC b_hi,w              ; ADD b with carry, but carry disapears in black hole
                 addWFC a_hi,f              ; ADD a, place result in a

term_d1                                     ;...  next is d = d xor a <<< 16
                 call compute_dxora
                 movFF d_hi,tmpXOR_hi       ; rotate 16 is actually only swapping
                 movFF d_ml,d_hi
                 movFF tmpXOR_hi,d_ml
                 movFF d_mh,tmpXOR_mh
                 movFF d_lo,d_mh
                 movFF tmpXOR_mh,d_lo

term_c1
                 call compute_c

term_b1                                     ;...  next is b = b xor c <<< 12
                 call compute_bxorc
                                            ; now rotate left 12 positions
                 bcf STATUS, C              ; prepare Carry flag with 0
                 btfsc b_ml,7               ; IF bit 7 of ml-byte
                 bsf STATUS, C              ; THEN prepare Carry with 1
                 rlcF b_hi
                 rlcF b_ml
                 rlcF b_hi
                 rlcF b_ml
                 rlcF b_hi
                 rlcF b_ml
                 rlcF b_hi
                 rlcF b_ml
                 bcf STATUS, C              ; prepare Carry flag with 0
                 btfsc b_lo,7               ; IF bit 7 of ml-byte
                 bsf STATUS, C              ; THEN prepare Carry with 1
                 rlcF b_mh
                 rlcF b_lo
                 rlcF b_mh
                 rlcF b_lo
                 rlcF b_mh
                 rlcF b_lo
                 rlcF b_mh
                 rlcF b_lo

term_a2
                 movF pointer2mc            ; load pointer into w [now (2i+1)]
                 movlw high permut_table_m  ; ..and use it here to find adress of current m
                 movwf TBLPTRH
                 rlncf pointer2mc, w
                 movwf TBLPTRL
                 tblrd*                     ; table read here into TABLAT
                 movff TABLAT, FSR0L        ; move adress to pointer

                 movFF INDF0,tmpXOR_lo      ; access content of m signum_r(2i) low byte loaded
                 movFF PREINC0,tmpXOR_ml    ; preincrement pointer, access midlowbyte
                 movFF PREINC0,tmpXOR_mh    ; preincrement pointer, access midhighbyte
                 movFF PREINC0,tmpXOR_hi    ; preincrement pointer, access highbyte
```

```
term_a2_lowbyte
                    decF pointer2mc            ; pointer now (2i)
                    movF pointer2mc            ; load pointer into w
                    movlw high permut_table_c  ; find c signum_r (2i)lowbyte adress
                    movwf TBLPTRH
                    rlncf pointer2mc, w
                    movwf TBLPTRL
                    tblrd*                     ; table read here into TABLAT
                    movff TABLAT, FSR1L        ; move adress to pointer, points now to c signum_r(2i)

                    movF INDF1                 ; content of c signum_r(2i+1) now in working reg


                    xorWF tmpXOR_lo,w          ; lowest byte [m signum_r (2i+1) XOR c signum_r (2i)]

                    addWFC b_lo,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_ml             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

                    addWFC a_lo,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_ml             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

term_a2_midlowbyte
                    movF PREINC1               ; content of c signum_r (2i) midlow byte loaded in w
                    xorWF tmpXOR_ml,w          ; midlow byte [m signum_r (2i+1) XOR c signum_r (2i)]

                    addWFC b_ml,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

                    addWFC a_ml,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_mh             ; then ...  add carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

term_a2_midhighbyte
                    movF PREINC1               ; content of c signum_r (2i) midhigh byte loaded in w
                    xorWF tmpXOR_mh,w          ; midhigh byte [m signum_r (2i+1) XOR c signum_r (2i)]

                    addWFC b_mh,w              ; ADD b with carry
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry

                    addWFC a_mh,f              ; ADD a, place result in a
                    btfsc STATUS, C            ; IF carrybit =1 ...
                    incF tmpXOR_hi             ; then ...  add carry
```

```
term_a2_highbyte
                    movF PREINC1              ; content of c signum_r (2i) high byte loaded in w
                    xorWF tmpXOR_hi,w         ; highest byte [m signum_r (2i+1) XOR c signum_r (2i)]
                    addWFC b_hi,w             ; ADD b with carry, but carry disapears in black hole
                    addWFC a_hi,f             ; ADD a, place result in a

term_d2                                       ;...  next is d = d xor a <<< 8
                    call compute_dxora
                    movFF d_hi,tmpXOR_hi      ; rotate 8 is actually swapping
                    movFF d_mh,d_hi
                    movFF d_ml,d_mh
                    movFF d_lo,d_ml
                    movFF tmpXOR_hi,d_lo

term_c2
call compute_c

term_b2                                       ;...  next is b = b xor c <<< 7
                    call compute_bxorc
                                              ; now rotate left 7 positions
                                              ; which can be seen as rotate right 1 and byte-wapping
                    bcf STATUS, C             ; prepare Carry flag with 0
                    btfsc b_lo,0              ; IF bit 0 of lo-byte
                    bsf STATUS, C             ; THEN prepare Carry with 1
                    rrcF b_hi                 ; rotate through carry
                    rrcF b_mh
                    rrcF b_ml
                    rrcF b_lo
                    movFF b_lo,tmpXOR_lo      ; temporarly save low
                    movFF b_hi,b_lo           ; swap byte high -> low
                    movFF b_mh,b_hi           ; midhigh to high
                    movFF b_ml,b_mh           ; midlow to midhigh
                    movFF tmpXOR_lo,b_ml      ; low to midlow

                    return

                                              ; function d <- d XOR a
compute_dxora
                    movF a_lo                 ; load a
                    xorWF d_lo,f              ; d XOR a, result in d
                    movF a_ml
                    xorWF d_ml,f
                    movF a_mh
                    xorWF d_mh,f
                    movF a_hi
                    xorWF d_hi,f
                    return
```

```
                                        ; function c <- c + d
compute_c
                    movF d_lo           ; load d
                    addWFC c_lo,f       ; ADD c, place result in c
                    btfsc STATUS, C     ; IF carrybit =1 ...
                    incF d_ml           ; then ...  add carry
                    btfsc STATUS, C     ; IF carrybit =1 ...
                    incF d_mh           ; then ...  add carry
                    btfsc STATUS, C     ; IF carrybit =1 ...
                    incF d_hi           ; then ...  add carry

                    movF d_ml
                    addWFC c_ml,f
                    btfsc STATUS, C
                    incF d_mh
                    btfsc STATUS, C
                    incF d_hi

                    movF d_mh
                    addWFC c_mh,f
                    btfsc STATUS, C
                    incF d_hi

                    movF d_hi
                    addWFC c_hi,f

                    return

                                        ; function b <- b XOR c
compute_bxorc
                    movF c_lo           ; load c
                    xorWF b_lo,f        ; b XOR c, result in b
                    movF c_ml
                    xorWF b_ml,f
                    movF c_mh
                    xorWF b_mh,f
                    movF c_hi
                    xorWF b_hi,f
                    return
```

B.3 ANSI C

In the C code provided with the submission, we added a function `AddSalt( hashState * state, const BitSequence * salt)`, whose arguments are:

- an initialized state (`state`)
- a salt (`salt`) of type `BitSequence`, long of 128 bits for BLAKE-28 and BLAKE-32, and long of 256 bits for BLAKE-48 or BLAKE-64

The function `AddSalt` extends the initialization of the hash state by adding a salt as extra parameter. Calling `AddSalt` is not compulsory; applications that don't use a salt should not call `AddSalt`. When a salt is required, `AddSalt` should be called after the call `Init`, and before any call to `Update`.

We give our optimized C code computing the compression function of BLAKE-32.

```
static HashReturn compress32( hashState * state, const BitSequence * datablock ) {
#define ROT32(x,n) (((x)<<(32-n))|( (x)>>(n)))
#define ADD32(x,y) ((u32)((x) + (y)))
#define XOR32(x,y) ((u32)((x) ^ (y)))
#define G32(a,b,c,d,i) do {\
      v[a] = XOR32(m[sigma[round][i]], c32[sigma[round][i+1]])+ADD32(v[a],v[b]);\
      v[d] = ROT32(XOR32(v[d],v[a]),16);\
      v[c] = ADD32(v[c],v[d]);\
      v[b] = ROT32(XOR32(v[b],v[c]),12);\
      v[a] = XOR32(m[sigma[round][i+1]], c32[sigma[round][i]])+ADD32(v[a],v[b]);\
      v[d] = ROT32(XOR32(v[d],v[a]), 8);\
      v[c] = ADD32(v[c],v[d]);\
      v[b] = ROT32(XOR32(v[b],v[c]), 7);\
    } while (0)

    u32 v[16];
    u32 m[16];
    int round;

    /* get message */
    m[ 0] = U8TO32_BE(datablock + 0);
    m[ 1] = U8TO32_BE(datablock + 4);
    m[ 2] = U8TO32_BE(datablock + 8);
    m[ 3] = U8TO32_BE(datablock +12);
    m[ 4] = U8TO32_BE(datablock +16);
    m[ 5] = U8TO32_BE(datablock +20);
    m[ 6] = U8TO32_BE(datablock +24);
    m[ 7] = U8TO32_BE(datablock +28);
    m[ 8] = U8TO32_BE(datablock +32);
    m[ 9] = U8TO32_BE(datablock +36);
    m[10] = U8TO32_BE(datablock +40);
    m[11] = U8TO32_BE(datablock +44);
    m[12] = U8TO32_BE(datablock +48);
    m[13] = U8TO32_BE(datablock +52);
    m[14] = U8TO32_BE(datablock +56);
    m[15] = U8TO32_BE(datablock +60);

    /* initialization */
    v[ 0] = state->h32[0];
    v[ 1] = state->h32[1];
    v[ 2] = state->h32[2];
    v[ 3] = state->h32[3];
    v[ 4] = state->h32[4];
    v[ 5] = state->h32[5];
    v[ 6] = state->h32[6];
    v[ 7] = state->h32[7];
    v[ 8] = state->salt32[0];
    v[ 8] ^= 0x243F6A88;
    v[ 9] = state->salt32[1];
    v[ 9] ^= 0x85A308D3;
    v[10] = state->salt32[2];
    v[10] ^= 0x13198A2E;
    v[11] = state->salt32[3];
    v[11] ^= 0x03707344;
    v[12] = 0xA4093822;
    v[13] = 0x299F31D0;
    v[14] = 0x082EFA98;
    v[15] = 0xEC4E6C89;

    if (state->nullt == 0) {
      v[12] ^= state->t32[0];
      v[13] ^= state->t32[0];
```

```
  v[14] ^= state->t32[1];
  v[15] ^= state->t32[1];
}

for(round=0; round<NB_ROUNDS32; ++round) {

  G32( 0, 4, 8,12, 0);
  G32( 1, 5, 9,13, 2);
  G32( 2, 6,10,14, 4);
  G32( 3, 7,11,15, 6);

  G32( 3, 4, 9,14,14);
  G32( 2, 7, 8,13,12);
  G32( 0, 5,10,15, 8);
  G32( 1, 6,11,12,10);
}

state->h32[0] ^= v[ 0];
state->h32[1] ^= v[ 1];
state->h32[2] ^= v[ 2];
state->h32[3] ^= v[ 3];
state->h32[4] ^= v[ 4];
state->h32[5] ^= v[ 5];
state->h32[6] ^= v[ 6];
state->h32[7] ^= v[ 7];
state->h32[0] ^= v[ 8];
state->h32[1] ^= v[ 9];
state->h32[2] ^= v[10];
state->h32[3] ^= v[11];
state->h32[4] ^= v[12];
state->h32[5] ^= v[13];
state->h32[6] ^= v[14];
state->h32[7] ^= v[15];
state->h32[0] ^= state->salt32[0];
state->h32[1] ^= state->salt32[1];
state->h32[2] ^= state->salt32[2];
state->h32[3] ^= state->salt32[3];
state->h32[4] ^= state->salt32[0];
state->h32[5] ^= state->salt32[1];
state->h32[6] ^= state->salt32[2];
state->h32[7] ^= state->salt32[3];

return SUCCESS;
}
```

C Intermediate values

As required by NIST, we provide intermediate values for hashing a one-block and a two-block message, for each of the required message sizes. For the one-block case, we hash the 8-bit message 00000000. For the two-block case we hash the 576-bit message 000...000 with BLAKE-32 and BLAKE-28, and we hash the 1152-bit message 000...000 with BLAKE-64 and BLAKE-48. Values are given left to right, top to bottom. For example

00800000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000001	00000000	00000008

represents

m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
m_8	m_9	m_{10}	m_{11}	m_{12}	m_{13}	m_{14}	m_{15}

C.1 BLAKE-32

One-block message

IV:

6A09E667	BB67AE85	3C6EF372	A54FF53A	510E527F	9B05688C	1F83D9AB	5BE0CD19

Message block after padding:

00800000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000001	00000000	00000008

Salt and counter

00000000	00000000	00000000	00000000			00000008	00000000

Initial state of v:

6A09E667	BB67AE85	3C6EF372	A54FF53A	510E527F	9B05688C	1F83D9AB	5BE0CD19
243F6A88	85A308D3	13198A2E	03707344	A409382A	299F31D8	082EFA98	EC4E6C89

State v after 1 round:

E78B8DFE	150054E7	CABC8992	D15E8984	0669DF2A	084E66E3	A516C4B3	339DED5B
26051FB7	09D18B27	3A2E8FA8	488C6059	13E513E6	B37ED53E	16CAC7B9	75AF6DF6

State v after 2 rounds:

9DE875FD	8286272E	ADD20174	F1B0F1B7	37A1A6D3	CF90583A	B67E00D2	943A1F4F
E5294126	43BD06BF	B81ECBA2	6AF5CEAF	4FEB3A1F	0D6CA73C	5EE50B3E	DC88DF91

State v after 5 rounds:

5AF61049	FD4A2ADC	5C1DBBD8	5BA19232	9A685791	2B3DD795	A84DF8D6	A1D50A83
E3C8D94A	86CCC20A	B4000CA4	596AC140	9D159377	A6374FFA	F00C4787	767CE962

State v after 10 rounds:

BC04B9A6	C340C7AC	4AA36DAA	FDB53079	0D85D1BE	14500FCD	E8A133E1	788F54AE
07EEC484	0505399D	837CCC3F	19AD3EE7	9D3FA079	FA1C772A	F0DFD074	5C25729F

Hash value output:

D1E39B45	7D2250B4	F5B152E7	4157FBA4	C1B423B8	7549106B	07FD3A3E	7F4AEB28

Two-block message

IV:

6A09E667	BB67AE85	3C6EF372	A54FF53A	510E527F	9B05688C	1F83D9AB	5BE0CD19

First compression Message block after padding:

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Salt and counter

00000000	00000000	00000000	00000000			00000200	00000000

Initial state of v:

6A09E667	BB67AE85	3C6EF372	A54FF53A	510E527F	9B05688C	1F83D9AB	5BE0CD19
243F6A88	85A308D3	13198A2E	03707344	A4093A22	299F33D0	082EFA98	EC4E6C89

State v after 1 round:

CC8704B8	14AF5E97	448BD7A4	7D5ED80F	88D88192	8DF5C28F	B11E631F	0AC6CEAB
01A455BA	43BAAEC3	C07C7DEC	4C912C63	6F8CDFEC	87FD02E0	D969B7B1	B74125B6

State v after 2 rounds:

D7ED8FC3	CC0A55F2	24014945	38A9D033	8DA19E93	9B91D76A	18E0448C	C10A0DF6
FB350B3C	D894B64E	F1B35175	D0DFF837	54E0DF8F	B3131C53	64BCB7A4	819FDFEA

State v after 5 rounds:

6BB8EAA1	FB2D35B9	F1C87115	8CCED083	C3CCF47F	EC295B60	18CF9A21	DC2AC833
1F87FBA1	759AE5F0	EE2F791D	11410F9F	46C442D0	EC5BE440	DC9ED226	97E6E8BC

State v after 10 rounds:

58B76F7A	24300259	EA5BAEE6	7ABECB5C	BEAA0C3C	38251BB6	F0D337AF	FF985D99
527E3C0C	4EBFC5FA	BF73D485	8B538346	03C56421	D1B9147E	63662E6C	70E9E8B2

Intermediate hash value

60C0B511	D1E86926	69468911	54A2BD20	EC613A62	72996744	8C36C068	D4917832

Second compression Message block after padding:

00000000	00000000	80000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000001	00000000	00000240

Salt and counter

00000000	00000000	00000000	00000000			00000240	00000000

Initial state of v:

60C0B511	D1E86926	69468911	54A2BD20	EC613A62	72996744	8C36C068	D4917832
243F6A88	85A308D3	13198A2E	03707344	A4093A62	299F3390	082EFA98	EC4E6C89

State v after 1 round:

2A12A61C	97455E40	71CEADC4	910B1078	420B2A13	EB18D4FC	179C8D8F	32115CDC
09A6088F	6698DD12	B7CD9DED	29E4EBE7	660D3499	75061D15	52848DFD	FC099457

State v after 2 rounds:

F4C6263D	7327094B	D139C80D	18A95331	6211D241	1BA339FA	4F059AB4	AA1580E9
211995BC	CE94B418	5391B476	6D480D9D	70988FB3	114F5AF1	8648B874	4F87AF38

State v after 5 rounds:

ECFEE77A	1F878081	339A7A59	D4CED068	73649B08	A3ACE1DA	A0B085A5	22CCBB53
27B8D497	30FB68D3	0ACF6405	524F093A	14E97D67	DCC7C7B0	98EA099A	A41ECBCA

State v after 10 rounds:

74CBFCFA	BC46AECD	8835BA12	FA9767EE	E1AAF6A5	2394033A	D433008D	897E05BB
9E68CD63	AEB60243	C3592B10	B979EC7A	B6AD289C	58A2B983	272EEF06	4BF407E4

Hash value output:

8A638488	C318C5A8	222A1813	174C36B4	BB66E45B	09AFDDFD	7F2B2FE3	161B7A6D

C.2 BLAKE-28

One-block message

IV:

C1059ED8	367CD507	3070DD17	F70E5939	FFC00B31	68581511	64F98FA7	BEFA4FA4

Message block after padding:

00800000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000008

Salt and counter

00000000	00000000	00000000	00000000			00000008	00000000

Initial state of v:

C1059ED8	367CD507	3070DD17	F70E5939	FFC00B31	68581511	64F98FA7	BEFA4FA4
243F6A88	85A308D3	13198A2E	03707344	A409382A	299F31D8	082EFA98	EC4E6C89

State v after 1 round:

04027914	24CFDD6B	7D33F394	12CBCC67	2DE38C62	6664F3D3	1D8D68FC	D6CD0B0B
481423A7	2F45B4F9	21C35492	50FB35FE	1255AE24	DFF2A626	9240D453	E8530B9D

State v after 2 rounds:

9FB36742	31BC5AC2	064D4095	4A2260B2	C12165D2	00D0EE58	AD1D8245	4F7B0F17
36EF0086	38DFA9E5	A67CC4B5	20963EEB	F2821838	D01907D2	7D15E12D	9B9EF864

State v after 5 rounds:

AAB629F7	16DE3E4A	5E78A622	257EBE3C	8669EA65	99D687FD	A632EA5E	511B1C46
93068AB9	67EA727C	5EC4C9A9	7212CD6A	7F90526F	6E8952F4	70E30791	16C1EBD8

State v after 10 rounds:

C9E1652F	BA9E5BDE	660E702E	67FC6579	BE6B4C7F	F5F0749A	1DFE158F	3B49131F
62A1B43D	E2D6F00A	67AAA716	E006A66D	95556F38	8145A426	1EC4DE7E	FC75FF74

Hash value output:

6A454FCA	6E347ED3	31D40A2F	70F49A2D	D4FE2876	1CEDC5AD	67C34456

Two-block message

IV:

C1059ED8	367CD507	3070DD17	F70E5939	FFC00B31	68581511	64F98FA7	BEFA4FA4

First compression Message block after padding:

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Salt and counter

00000000	00000000	00000000	00000000			00000200	00000000

Initial state of v:

C1059ED8	367CD507	3070DD17	F70E5939	FFC00B31	68581511	64F98FA7	BEFA4FA4
243F6A88	85A308D3	13198A2E	03707344	A4093A22	299F33D0	082EFA98	EC4E6C89

State v after 1 round:

E5B52991	1FBB7ECB	F7350E64	0C8D11C6	148B1E94	7C688FED	C8FEEE1B	4046AC6E
8BC4F63C	C1C7FE8C	1FA6AE53	EE4DC034	87863887	2D70805B	4FA9A232	D9860F12

State v after 2 rounds:

2F3A90E3	EBBBC331	5737A2D1	6480F282	DB471183	43014ABD	88924F03	5160CB72
6E8F7EEB	115D1FD6	43387C5F	FFB59797	F8663D1A	D5FA0EC9	0C0ED9E5	8579D4A6

State v after 5 rounds:

F729608D	8119B461	E62F4D54	7889D045	838FBD7D	1A1E5618	8728C02B	E973E337
06F32665	23B502C7	FEDC26FC	CEFD14A6	DAD6B58F	4DCA0D19	31D904CB	3C7E2160

State v after 10 rounds:

D3465C90	9AF58DB6	77044D06	8782E7B8	F5C3F50A	78A3A751	D7923EF6	647B8D32
7B80826F	21577A7A	CE253568	1B6A082B	D5E512E2	E213D8E0	F39651A7	F9FDAE6E

Intermediate hash value

69C34027	8DDE22CB	8951A579	6BE6B6AA	DFE6ECD9	F2E86AA0	40FDE0F6	237C6CF8

Second compression Message block after padding:

00000000	00000000	80000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000240

Salt and counter

00000000	00000000	00000000	00000000			00000240	00000000

Initial state of v:

69C34027	8DDE22CB	8951A579	6BE6B6AA	DFE6ECD9	F2E86AA0	40FDE0F6	237C6CF8
243F6A88	85A308D3	13198A2E	03707344	A4093A62	299F3390	082EFA98	EC4E6C89

State v after 1 round:

215AEB86	8A40E284	8889C5CF	3A7A93F9	3ECC4417	4EB11689	3B06106F	0092D184
7F047CFA	BCBFA0C8	8E907E6C	582C5CC4	C7C016E8	696F917E	0AF46854	929FD9AB

State v after 2 rounds:

998F9380	6D6C16FD	79CE8034	65B3E4A4	459C22CC	3B8EA998	35638BB5	D9F54BB2
A3C7177D	A3E59D0B	A059BBAF	C62D9E5A	B1A2808E	9032CCCB	B36DB002	ECDC6D0D

State v after 5 rounds:

2E967A8A	56885CE5	8218AB56	CFBA4356	32627515	913CB1C0	F480A1AE	B524AE3A
643AE882	419A50AA	74CDF767	CFC40BDF	2FDDA24A	42651292	2B4A4CE2	B7B83356

State v after 10 rounds:

2C975117	5D90EBA5	78A0F5C4	FB0EDE6F	E88CE2F8	03206935	CD05A414	05F47C03
2B9CC580	2EE07DFA	A110229E	DCE37F4B	4E31D239	23EC233D	D697DF5B	86F74FCC

Hash value output:

6EC8D4B0	FEAEB494	50E17223	4C0B178E	795BDC18	D22420A8	5B6F9BB9

C.3 BLAKE-64

One-block message

Message block after padding:

0080000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000001	0000000000000000	0000000000000008

IV:

6A09E667F3BCC908	BB67AE8584CAA73B	3C6EF372FE94F82B	A54FF53A5F1D36F1
510E527FADE682D1	9B05688C2B3E6C1F	1F83D9ABFB41BD6B	5BE0CD19137E2179

Salt and counter

0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000008	0000000000000000		

Initial state of v:

6A09E667F3BCC908	BB67AE8584CAA73B	3C6EF372FE94F82B	A54FF53A5F1D36F1
510E527FADE682D1	9B05688C2B3E6C1F	1F83D9ABFB41BD6B	5BE0CD19137E2179
243F6A8885A308D3	13198A2E03707344	A4093822299F31D0	082EFA98EC4E6C89
452821E638D0137F	BE5466CF34E90C64	C0AC29B7C97C50DD	3F84D5B5B5470917

State v after 1 round:

98957863D61905B3	2064357139454E43	391FB64BD757FB63	A77C0E00BBE362B5
86D4B6C41F60C7E1	823F30053BEB147C	68E6FC038D3B0B70	D93165F3477733DF
DED9D48A51DDE68F	3B73BB8B500C22B1	03F92332A668036B	E2F0B698EA636BB9
A40103908A3FD2AE	016613AD1A47C604	BFBC229C63E28B76	02A5DDF1AFF95A3A

State v after 2 rounds:

84DAC4B310F8B76B	01CE15A3AA8D8B2E	F12C708C9D10A8B0	778C288779642198
13D4F878F30C3F5E	5B049744B1932015	0FCFC0DEE2C0F4A0	80B67926A85E5AD8
8D0E3FB6C987BE2B	A1E68630BE9171C7	06D755881837E80F	B8729CFE5D112FA0
9226C2A7D8AD1F76	8265C86D8C126BC1	C0BFC6FEE0CFF19B	E48FA8828EEC436A

State v after 5 rounds:

EFD689A66BDC0A95	2253DDE0CB058FFC	886B8A405AE244FA	CA317DFE42522691
FB5123461DF359E7	17EFB7C5FD09F586	8E07FE0BD4918C29	E3AE0ACDF25D6303
6D4719E51F4A0833	27218B65BD7D4BC0	9227B3EA1497AD64	72B2C922552B72F9
855C5D1C44DD57A4	FC1340AE55773E39	03B57F827BE2F1CD	B43F42F4AA368791

State v after 14 rounds:

1C803AADBC03622B	055EB72E5A0615B3	4624E5B1391E8A33	7B2A7AA93E27710A
F7EA864E4D591DF7	34E2FF788DBD71A7	01D13A3673488668	390D346D5CB82ECF
00D6AC4E1B3D8DE0	58CD6E304B8AD357	33E864217D9C1147	C9C686A43790D49F
8C76318C3B9E3C07	20952009E26AE7A1	E63865AEC6B7E10C	2FAFFDCB74ADE2DE

Hash value output:

765F7084548226C3	E6F4779B954661DF	49A272E2BA16635F	17A3093756AA9364
2A92E5BDDB21A321	8F72B7FD44E9FA19	F86A86334EBEDA0F	4D4204BF3B6BED68

Two-block message

IV:

6A09E667F3BCC908	BB67AE8584CAA73B	3C6EF372FE94F82B	A54FF53A5F1D36F1
510E527FADE682D1	9B05688C2B3E6C1F	1F83D9ABFB41BD6B	5BE0CD19137E2179

First compression Message block after padding:

0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000000	0000000000000000	0000000000000000	0000000000000000

Salt and counter

0000000000000000	0000000000000000	0000000000000000	0000000000000000
0000000000000400	0000000000000000		

Initial state of v:

6A09E667F3BCC908	BB67AE8584CAA73B	3C6EF372FE94F82B	A54FF53A5F1D36F1
510E527FADE682D1	9B05688C2B3E6C1F	1F83D9ABFB41BD6B	5BE0CD19137E2179
243F6A8885A308D3	13198A2E03707344	A4093822299F31D0	082EFA98EC4E6C89
452821E638D01777	BE5466CF34E9086C	C0AC29B7C97C50DD	3F84D5B5B5470917

State v after 1 round:

1BE45837F23BAEE5	2111F54A79AD333D	F51F6F4BDBDACC64	BFD3AF47522BA647
3CBD1A03BABEE0B1	4C1679E18847BED0	65375DDA217AF370	FC804555EA9C61C0
13DCA8E50FCBEEA2	A028A1030A7F2907	A8486683A019458C	6F50BBC1BAAD52D1
26FF0C474E8A8E46	3661DBA5D8ADCE89	FB6E1530F3FA0CD2	29F3D982476D1C5B

State v after 2 rounds:

078A7F4AB38B51A3	3CC938D334F088AE	C9688433013EB5F4	963A2028D731F262
A2E4F2F9127A623E	7DF540DFFEC115F7	539403CCFF3E7EDA	4039A268638B91E7
6DE0D9BF908EF408	D9747550EADAF1B2	5CBEB17148553D5C	CC40FD3E15DD6C42
528F6D54B521156E	CE320314E7255341	C374721DDC0FEEB2	F64047D64AED39A9

State v after 5 rounds:

7CE663EFB2F3997D	CA831A13AE1ADEA2	1B489B08D9C77613	8449E1F48BF74A4A
D7F36F5DAD19B6F0	1B79A03B9DADCC93	0C5A6120750E5B4A	4D74C0055FEA4D29
91ECB03DDFB95F46	D12929425D257265	4436F30BA8FDA059	8F5EA5D22A3CFC07
1591886653094950	A98739E101B44D3A	78556C535F2905F2	E5BC8EDDAC0176DF

State v after 14 rounds:

BAE5B20438EBD1AE	FB9EB556D67BE6CD	1DD32AA12CB2C411	42374BFECE90FA65
807E55B199234ECC	7FC73B526FADC9D8	760B6B884BA1B098	B77D0E14CCB094DD
FB079B4D09CDA172	EE56FD3B622F28AC	A4C9C6924B60C4B9	244E57A15B596644
7C86CAACE54A8E3E	71782EF1771E5ABA	5FCE8F0139CBA368	D3F1A57A2BD841F4

Intermediate hash value:

```
2BEBCF2EC29AB9D4  AEAFE6E8309E695A  85741F419946F883  C336E965CAD4AAD0
ADF6CD62D18F4223  95BA7D2F338DFF7D  36463D22892BAE9B  3F6C6677F416F450
```

Second compression Message block after padding:

```
0000000000000000  0000000000000000  8000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000001  0000000000000000  0000000000000480
```

Salt and counter

```
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000480  0000000000000000
```

Initial state of v:

```
2BEBCF2EC29AB9D4  AEAFE6E8309E695A  85741F419946F883  C336E965CAD4AAD0
ADF6CD62D18F4223  95BA7D2F338DFF7D  36463D22892BAE9B  3F6C6677F416F450
243F6A8885A308D3  13198A2E03707344  A4093822299F31D0  082EFA98EC4E6C89
452821E638D017F7  BE5466CF34E908EC  C0AC29B7C97C50DD  3F84D5B5B5470917
```

State v after 1 round:

```
97B7744F66047D30  EFAF6C7255A85A64  18269E18102C7DF0  5845FCE8352347AA
33945C40520094E4  BF2E239191F3FB2A  F52AA83F401E1C94  03D39EF6D699D428
C9C5F695FF595911  BA2CB996500E645A  043F4721E6185DC6  F06941D9A4AE3838
45F73F26426EDC75  9C1FDEEE8C3B71F6  E362AD2A84BA1C65  972A9B18D218E63C
```

State v after 2 rounds:

```
77DDF1D318481AF5  0E5BB7B53A077AB3  52AC32E7E020E8C4  9F2D720DFA259B0E
AA8C0FD13D1AC0EC  85AC17EB7D90CB3C  45C7BAC2500D182B  F70ABDEDE7FBE95B
4B8145BC80391D37  8CE035CEB9332CCE  5160F2E0762575C9  5F14547FC0B45158
B8033BABB00BB947  4690BE32AC7037B1  A8841F193796A0AF  E0C40F4CDA85533A
```

State v after 5 rounds:

```
4E1FE57385697E55  815DEE13C3C990D8  AAB9BC1621BFDB4C  24308C06892728BE
C72F23D392287B05  5CAB7BB581F70C1F  ADA9296C20920C02  32CBBC0000666FE8
AAE844890FC188D6  FFAA213A3CC310DF  ECA6E32297722DC8  9C5D5C2CCF01C274
AF2A09A3721A949B  6C3461C3134774B6  48C942D2E0B00355  D5BF25B37AA44AE1
```

State v after 14 rounds:

```
60EF7F97E6FE03C5  EB78F18831CFFFE8  1207B65336348F8E  D380A238CA002C04
87CFE47BA3E06881  568BE33B0D9007D7  5D4147CBD6987380  504CD06EA90E16AF
A1B38091204C9B14  3424EFBEE7293F03  2C9CF1CDFA356568  A7A86D768E2B3CF1
19F87A0EB186D235  7158735578B32859  C99F5DEF5FE6170F  0A07E5F6BF1273C1
```

Hash value output:

```
EAB7302804282105  71F3F8DEE678A9B1  BBEF58DF55471265  B71E262B8EFFBA25
33C15317C3E9F897  B269ED4146AED0F3  A29827060055CA14  652753EFE20A913E
```

C.4 BLAKE-48

One-block message

Message block after padding:

```
0080000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000008
```

IV:

```
CBBB9D5DC1059ED8  629A292A367CD507  9159015A3070DD17  152FECD8F70E5939
67332667FFC00B31  8EB44A8768581511  DB0C2E0D64F98FA7  47B5481DBEFA4FA4
```

Salt and counter

```
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000008  0000000000000000
```

Initial state of v:

```
CBBB9D5DC1059ED8  629A292A367CD507  9159015A3070DD17  152FECD8F70E5939
67332667FFC00B31  8EB44A8768581511  DB0C2E0D64F98FA7  47B5481DBEFA4FA4
243F6A8885A308D3  13198A2E03707344  A4093822299F31D0  082EFA98EC4E6C89
452821E638D0137F  BE5466CF34E90C64  C0AC29B7C97C50DD  3F84D5B5B5470917
```

State v after 1 round:

```
5B063A05F1A479BB  82CA717B7A4F6F94  4F58DFBDAB593FFB  F826C578573BEC7E
C0836949C0FA750A  99FD9AA2E726BF09  32F52E2CBFC45A64  80686C4AE126CDA9
5EB10A738BF891EE  3DF23E84618C549F  F2C230E414F34299  9191632BEE7EE45E
C83CF461EDC79B6D  8FF3FB919A781656  9BE2FD02DFE1B98A  5B64934E1FE8370D
```

State v after 2 rounds:

```
5B2B57C1586FEEA6  7413D0FE48C32BE2  535CA6F699C38D80  BBEE0C0CBD530269
9E3CD39F1C1868DA  A4D8C74D2A7AA0F5  7524F4211494EF12  A94A548795A319EC
B9F9689AFC6AEDA6  EBC0E49C45A1E9AA  260D24A2D818CB43  BA3914617A2D98EC
F7BA66DC1AEB284C  9C362FBCE59789D9  74B3B2650C513D2C  D53EB118A489C053
```

State v after 5 rounds:

```
4292009F26C4CAA5  17DF7CF80E7A6542  24CA7FE6607B8393  C91DDCA2AFECD146
7ECAF3B6BC20CFD7  00D47510478C61B9  F1A2F95870EAF7B0  52AD845DA7D26918
A0E941F5B18548FA  BFCB96FC91F31717  4B9F4584075D75C4  BF9C0EE7E53657FF
CB09E853BA91C13D  FD46E7FE45AA85E3  CE6E1C891FFAAEF9  2C9E50427598264A
```

State v after 14 rounds:

```
1DD69F386C168B30  EB4B1AD311C7C265  42044AA20151C2A0  1BD8CBE637DFB25D
94ABF0918D4B9749  6A59118B73AB159B  56EE21C11395B066  00BB340A4C94C03B
2EC5D56650765851  B84BF78188E22A8D  5149DF33128FAAC1  8E52CD242ADB8EA8
88EA30691A1873AA  DABF685D0556D4AF  51168CA096930C62  E42652FFB6D559CF
```

Hash value output:

```
F8A8D703FD654DB9  319AC478AF593DEF  821494CB23AEB576  80A5EA1AEA0A65CC
7B72E69F6893EFD2  3E5233511EA5D425
```

Two-block message

IV:

```
CBBB9D5DC1059ED8  629A292A367CD507  9159015A3070DD17  152FECD8F70E5939
67332667FFC00B31  8EB44A8768581511  DB0C2E0D64F98FA7  47B5481DBEFA4FA4
```

First compression Message block after padding:

```
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
```

Salt and counter

```
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000400  0000000000000000
```

Initial state of v:

```
CBBB9D5DC1059ED8  629A292A367CD507  9159015A3070DD17  152FECD8F70E5939
67332667FFC00B31  8EB44A8768581511  DB0C2E0D64F98FA7  47B5481DBEFA4FA4
243F6A8885A308D3  13198A2E03707344  A4093822299F31D0  082EFA98EC4E6C89
452821E638D01777  BE5466CF34E9086C  C0AC29B7C97C50DD  3F84D5B5B5470917
```

State v after 1 round:

```
3BBF567D6D8E7C9A  826AB1796F4B2F2A  D3589AB1A73A76FB  7FFB66FFAAA078B4
1F7BFE2284B78162  E1F997F6B243CD2A  70B6BA23B832F52D  B5418F66EC6D2031
ADA82F0DD0769947  C23086272083F261  F6A871C70393F9FA  8D515B125606EADA
C802F0CF294F6269  C6F36399DF7E1E35  8F20EDDF0BA7D74A  DE4472F1D1506E6F
```

State v after 2 rounds:

```
EA85A242A7F6CFCE  89A54C23487CA8BF  5C8893D38EF63BF3  46B087AA28D56BE5
5D085C4433F1929C  8134381EEE29381F  36505EC762DAB50C  D71519E8814D4E39
F4A2235795910F0F  58AD370D224CB9B0  47D1E79A61966B91  0563F8E3BA681DBD
48D6E244313C9D0C  D079DE27CBA8F3C8  DD134C5A6384EFAC  7E27A4AC04CF472D
```

State v after 5 rounds:

```
802C1F2E2198AE80  EE5B58BB836A1D70  8157B2DA7FB7781D  9295E0C42DC728FC
D88DF0E4BFC0ADAB  7871BB15B4555CAB  F89864B706E11F5F  F01F54F3CB2B4E5F
014C1C71F0918E4D  EA826F742DAA21D0  33C03F7DFB0166DC  11442F58CFC88765
0D2FB5DCD1ADE0AE  7C972BBFEF957FB5  7D874F206DD2E3FB  8CFE8958C6233803
```

State v after 14 rounds:

```
48D2ABEEC2D71CC5  453ACF7BB753BBF1  8AD951B5121E15F2  6D70D249D39A715A
AF9FDE1EE3CAD40D  C661F45A89950ADC  843A9EE5D8169BD5  C74BC1121B511E1A
12D0217D0E74E5B1  CC7BD5E254C52B17  8636BF1D9B6E636B  E5FDF466195146E0
16DAC45878471174  CDAE5B050C98E92A  121004668DBAB665  AEF35F816CEA29F2
```

Intermediate hash value:

```
91B917CE0DA667AC  EBDB33B3D5EA45E1  9DB6EFF2B900AB8E  9DA2CAF73DC56E83
DE763C21644DCE48  857BE5D8ED55F6E7  4D26B48E3155A217  2E0DD68EC941784C
```

Second compression Message block after padding:

```
0000000000000000  0000000000000000  8000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000000  0000000000000000  0000000000000000  0000000000000480
```

Salt and counter

```
0000000000000000  0000000000000000  0000000000000000  0000000000000000
0000000000000480  0000000000000000
```

Initial state of v:

```
91B917CE0DA667AC  EBDB33B3D5EA45E1  9DB6EFF2B900AB8E  9DA2CAF73DC56E83
DE763C21644DCE48  857BE5D8ED55F6E7  4D26B48E3155A217  2E0DD68EC941784C
243F6A8885A308D3  13198A2E03707344  A4093822299F31D0  082EFA98EC4E6C89
452821E638D017F7  BE5466CF34E908EC  C0AC29B7C97C50DD  3F84D5B5B5470917
```

State v after 1 round:

```
EB5305AF9C675316  B04F4367EF5BCB01  C5ACFFF4A502B3AC  7B1494BE21EA8AEC
EFC2114AF5B89E14  8C5D51A5085E8343  FB3871A4E93CDC4C  730A928E549F309C
EB5B62A3636B5994  380D6D5F3BE6DE51  0C3A9D08903CE741  C89B96FA0C4FE476
5406B1EE5E8E0B04  DE7BCC2A14B5687D  189291CFE98DD45F  CC0AEDF772238F5A
```

State v after 2 rounds:

```
EDDD82F01BAC0561  4CFDCDFFA77330A4  A3BEC55427F66DD0  E61E7A01F5B44065
697C76A05841756C  C4238D4E0E4F480C  1924ABE4F334FCE1  4410660CC930607C
4CD8F10D348336C4  8A2C792E6B6607D4  FC362721166BF27C  BF00A632885CC7ED
1B470C101AA73F07  F21D3F3E6C497536  C6A24BD8C6E548A5  0C9F27FDC4AA89C1
```

State v after 5 rounds:

```
0B54F86A35B74457  A4315CE1B09ADE8D  E3078EA3D51F8EA4  453748C8FEDC0071
EADF5CBCC39D038D  D9763C0B677A4587  B0EBA224DEE4E974  AFEA28B0B8AAE56C
BB57DFD78B8B4D38  7B04C7FDE21E1FB1  5FE3B5E55E53DA1D  9483FC16047631D4
437715901F3DBBAF  34AC592C780C505B  0475414152111284  80397DEF4F32B2BF
```

State v after 14 rounds:

```
757F77BF12F5C1B0  223D06220DB9BFF2  330D25F2DA9321D4  CC96ECE63FA108EA
3BCB8A5F730CE929  9A760947DC3E64DE  5790F83BFC764982  ADDDA3E22C5AB3C5
2CC451168EAEDA0F  5C335D58949ACF0D  89406B7B3F0A86C0  F814998AD3057F48
073E119817B69FF5  83F7A1741EE1F446  15A3F6053FD0B9F0  0B2BE29A95030597
```

Hash value output:

```
C802316791FD7C13  95D568C94CC9351E  27FBA17B5C990C9A  A920BF9BD1611921
E283A7E600F7B894  9CFA4DEB2F8A667F
```