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 an 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 .................................................. 3
   1.1 Design principles .................................. 4
   1.2 BLAKE in a nutshell ................................ 5
   1.3 Expected strength .................................. 6
   1.4 Advantages and limitations ......................... 6
   1.5 Notations ........................................... 7

2 Specification .................................................. 8
   2.1 BLAKE-32 ........................................... 8
      2.1.1 Constants ...................................... 8
      2.1.2 Compression function ......................... 8
      2.1.3 Hashing a message ............................. 11
   2.2 BLAKE-64 ........................................... 12
      2.2.1 Constants ...................................... 12
      2.2.2 Compression function ......................... 12
      2.2.3 Hashing a message ............................. 13
   2.3 BLAKE-28 ........................................... 13
   2.4 BLAKE-48 ........................................... 13
   2.5 Alternative descriptions ............................ 14
   2.6 Tunable parameter ..................................... 15

3 Performance .................................................. 16
   3.1 Generalities ......................................... 16
      3.1.1 Complexity ...................................... 16
      3.1.2 Memory/speed tradeoffs ....................... 16
      3.1.3 Parallelism ..................................... 17
   3.2 ASIC and FPGA ....................................... 17
      3.2.1 Architectures .................................... 17
      3.2.2 Implementation results ....................... 18
      3.2.3 Evaluation ...................................... 20
   3.3 8-bit microcontroller ................................ 20
      3.3.1 The PIC18F2525 ................................ 20
      3.3.2 Memory management ............................ 21
      3.3.3 Speed ........................................... 22
   3.4 Large processors ....................................... 23

4 Using BLAKE .................................................. 26
   4.1 Hashing with a salt .................................. 26
   4.2 HMAC and UMAC ...................................... 27
   4.3 PRF ensembles ........................................ 27
   4.4 Randomized hashing ................................... 28

5 Elements of analysis ......................................... 29
   5.1 Permutations ......................................... 29
   5.2 Compression function ................................ 30
      5.2.1 G function ...................................... 30
      5.2.2 Round function ................................ 32
      5.2.3 Compression function ......................... 34
      5.2.4 Fixed-points .................................... 36
   5.3 Iteration mode (HAIFA) ................................ 37
   5.4 Pseudorandomness ..................................... 38
   5.5 Generic attacks ...................................... 39
      5.5.1 Length extension ................................ 39
      5.5.2 Collision multiplication ....................... 39
      5.5.3 Multicollisions .................................. 39
      5.5.4 Second preimages ................................ 40
      5.5.5 Side channels .................................... 41
      5.5.6 SAT solvers ..................................... 41
      5.5.7 Algebraic attacks ................................ 41
   5.6 Dedicated attacks ..................................... 41
      5.6.1 Symmetric differences ......................... 41
      5.6.2 Differential attack ............................. 42
      5.6.3 Slide attack ...................................... 43

6 Acknowledgments ................................................ 44

Bibliography ..................................................... 45

A Round function example ........................................ 48

B Source code ..................................................... 50
   B.1 VHDL .................................................. 50
   B.2 PIC assembly .......................................... 59
   B.3 ANSI C .................................................. 64

C Intermediate values ............................................ 67
   C.1 BLAKE-32 ............................................ 67
   C.2 BLAKE-28 ............................................ 69
   C.3 BLAKE-64 ............................................ 71
   C.4 BLAKE-48 ............................................ 74
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word</th>
<th>Message Blocks</th>
<th>Digest</th>
<th>Salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE-28</td>
<td>32</td>
<td>$&lt;2^{64}$</td>
<td>512</td>
<td>224</td>
</tr>
<tr>
<td>BLAKE-32</td>
<td>32</td>
<td>$&lt;2^{64}$</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>BLAKE-48</td>
<td>64</td>
<td>$&lt;2^{128}$</td>
<td>1024</td>
<td>384</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>64</td>
<td>$&lt;2^{128}$</td>
<td>1024</td>
<td>512</td>
</tr>
</tbody>
</table>

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.](image)

The inner state of the compression function is represented as a $4 \times 4$ 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 brute-force 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 \( \text{F0} = 240 \)). A word is either a 32-bit or a 64-bit string, depending on the context. We use the same conventions of big-endianess 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leftarrow )</td>
<td>variable assignment</td>
</tr>
<tr>
<td>(+)</td>
<td>addition modulo (2^{32}) or (modulo (2^{64}))</td>
</tr>
<tr>
<td>(\oplus)</td>
<td>Boolean exclusive OR (XOR)</td>
</tr>
<tr>
<td>(\gg k)</td>
<td>rotation of (k) bits towards less significant bits</td>
</tr>
<tr>
<td>(\ll k)</td>
<td>rotation of (k) bits towards more significant bits</td>
</tr>
<tr>
<td>(\langle \ell \rangle_k)</td>
<td>encoding of the integer (\ell) over (k) bits</td>
</tr>
</tbody>
</table>

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 \(p = p_0 \parallel p_1 \parallel \ldots\). For a message \(m\), \(m^i\) denotes its \(i^{th}\) 16-word block, thus \(m^i\) 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:

\[
\begin{align*}
IV_0 &= 6A09E667 \\
IV_1 &= BB67AE85 \\
IV_2 &= 3C6EF372 \\
IV_3 &= A54FF53A \\
IV_4 &= 510E527F \\
IV_5 &= 9B05688C \\
IV_6 &= 1F83D9AB \\
IV_7 &= 5BE0CD19 \\
\end{align*}
\]

BLAKE-32 uses 16 constants:\footnote{First digits of \(\pi\).}

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

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 \(t = t_0, t_1\)
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

\[h' = \text{compress}(h, m, s, t)\]

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 \(4 \times 4\) 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

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

where, at round \(r\), \(G_l(a, b, c, d)\) sets

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

\[\text{In the rest of the paper, for statements that don’t depend on the index } i \text{ we shall omit the subscript and write simply } G.\]
The first four calls $G_0, \ldots, G_3$ can be computed in parallel, because each of them updates a distinct column of the matrix. We call the procedure of computing $G_0, \ldots, G_3$ a *column step*. Similarly, the last four calls $G_4, \ldots, G_7$ 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.

![Diagram](image-url)
Finalization

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

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

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 \( \ell \). Padding can be represented as

\[ m \leftarrow m \parallel 1000 \ldots 0001 \langle \ell \rangle_{64} \]

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 \( m^0, \ldots, m^{N-1} \). We let \( \ell^i \) be the number of message bits in \( m^0, \ldots, m^i \), 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:

\[
\begin{align*}
    h^0 & \leftarrow IV \\
    \text{for } i = 0, \ldots, N - 1 \\
    & \quad h^{i+1} \leftarrow \text{compress}(h^i, m^i, s, \ell^i) \\
    \text{return } h^N
\end{align*}
\]
The procedure of hashing $m$ with BLAKE-32 is aliased $\text{BLAKE-32}(m, s) = h^N$, where $m$ is the (non-padded) message, and $s$ is the salt. The notation $\text{BLAKE-32}(m)$ 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:

$$
\begin{array}{ll}
IV_0 &= 6A09E667F3BCC908 \\
IV_1 &= BB67AE8584CAA73B \\
IV_2 &= 3C6EF372FE94F82B \\
IV_3 &= A54FF53A5F1D36F1 \\
IV_4 &= 510E527FADE682D1 \\
IV_5 &= 9B05688C2B3E6C1F \\
IV_6 &= 1F83D9ABFB41BD6B \\
IV_7 &= 5BE0CD19137E2179 \\
\end{array}
$$

BLAKE-64 uses the constants

$$
\begin{array}{ll}
c_0 &= 243F6A8885A308D3 \\
c_1 &= 13198A2E03707344 \\
c_2 &= A4903822299F31D0 \\
c_3 &= 082EFA98EC6C89 \\
c_4 &= 452821E638D01377 \\
c_5 &= BE5466CF34E90C6C \\
c_6 &= COAC29B7C97C50DD \\
c_7 &= 3F84D5D98979FB1B \\
c_8 &= 9216D5D98979FB1B \\
c_9 &= D1310BA698DFB5AC \\
c_{10} &= 2FFD72DBD01ADFB7 \\
c_{11} &= B8E1AFED6A267E96 \\
c_{12} &= BA7C9045F12C7F99 \\
c_{13} &= 2A419947B3916CF7 \\
c_{14} &= 0801F2E2858EFC16 \\
c_{15} &= 636920D871574E69 \\
\end{array}
$$

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 $G_i(a, b, c, d)$ computes

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

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 $\sigma_r \mod 10$ (for example, in the last round $r = 13$ and the permutation $\sigma_{13 \mod 10} = \sigma_3$ is used).

$^3$First digits of $\pi$. 
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:

\[ m \leftarrow m \| 1000 \ldots 001(\ell)_{128} \]

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:
  
  \[
  \begin{align*}
  IV_0 &= C1059ED8 & IV_1 &= 367CD507 \\
  IV_2 &= 3070DD17 & IV_3 &= F70E5939 \\
  IV_4 &= FFC00B31 & IV_5 &= 68581511 \\
  IV_6 &= 64F98FA7 & IV_7 &= BEFA4FA4 \\
  \end{align*}
  \]

- in the padded data, the 1 bit preceeding the message length is replaced by a 0 bit:
  
  \[ m \leftarrow m \| 1000 \ldots 0000(\ell)_{64} \]

- 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:
  
  \[
  \begin{align*}
  IV_0 &= CBBB9D5DC1059ED8 & IV_1 &= 629A292A367CD507 \\
  IV_2 &= 9159015A3070DD17 & IV_3 &= 152FECD8F70E5939 \\
  IV_4 &= 67332667FFC00B31 & IV_5 &= 8EB44A8768581511 \\
  IV_6 &= DB0C2E0D64F98FA7 & IV_7 &= 47B5481DBEFA4FA4 \\
  \end{align*}
  \]

- in the padded data, the 1 bit preceeding the message length is replaced by a 0 bit:
  
  \[ m \leftarrow m \| 1000 \ldots 0000(\ell)_{128} \]

- the output is truncated to its first 384 bits, that is, the iterated hash returns \( h_0^N, \ldots, h_6^N \) instead of \( h^N = h_0^N, \ldots, h_7^N \)
2.5 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 $i = 0, \ldots, 3$
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].

![Diagram](image.png)

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 $i = 0, \ldots, 3$
3. make a column-step again

Finally, another equivalent definition of a round is

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

$$G_8(v_0, v_5, v_{10}, v_{15}) \quad G_{10}(v_1, v_6, v_{11}, v_{12}) \quad G_{12}(v_2, v_7, v_8, v_{13}) \quad G_{14}(v_3, v_4, v_9, v_{14})$$

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

$$a \leftarrow a + b + (m_{\sigma_i(l)} \oplus c_{\sigma_i(l+1)})$$
$$d \leftarrow (d \oplus a) \gg 16$$
$$c \leftarrow c + d$$
$$b \leftarrow (b \oplus c) \gg 12$$
$$a \leftarrow a + b + (m_{\sigma_i(l+1)} \oplus c_{\sigma_i(l)})$$
$$d \leftarrow (d \oplus a) \gg 8$$
$$c \leftarrow c + d$$
$$b \leftarrow (b \oplus c) \gg 7$$

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 \textit{serial} machine, the speed of a round is about eight times the speed of a \( G \)
- on a \textit{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 \textit{initialization} and \textit{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 \textit{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 \textit{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.

- [½G]-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. In addition, the hardware efficiency computes the ratio between speed and area of the circuits. The [8G]

---

---

---

---

---
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 [4G]-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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[8G] BLAKE-32</td>
<td>58.30</td>
<td>114</td>
<td>11</td>
<td>5295</td>
<td>90.8</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>132.47</td>
<td>87</td>
<td>15</td>
<td>5910</td>
<td>44.6</td>
</tr>
<tr>
<td>[4G] BLAKE-32</td>
<td>41.31</td>
<td>170</td>
<td>21</td>
<td>4153</td>
<td>100.5</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>82.73</td>
<td>136</td>
<td>29</td>
<td>4810</td>
<td>58.1</td>
</tr>
<tr>
<td>[1G] BLAKE-32</td>
<td>10.54</td>
<td>40</td>
<td>81</td>
<td>253</td>
<td>24.0</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>20.61</td>
<td>20</td>
<td>113</td>
<td>181</td>
<td>8.8</td>
</tr>
<tr>
<td>[1/2G] BLAKE-32</td>
<td>9.89</td>
<td>40</td>
<td>161</td>
<td>127</td>
<td>12.9</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>19.46</td>
<td>20</td>
<td>225</td>
<td>91</td>
<td>4.7</td>
</tr>
</tbody>
</table>

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 µm².

Three architectures have been implemented on FPGA silicon devices: the Xilinx Virtex-5, Virtex-4, and Virtex-II Pro³. We used SynplifyPro and Xilinx ISE for synthesis and place & route. Table 3.2 reports resulting circuit performances.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[8G]-BLAKE architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[8G]-BLAKE architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLAKE-32</td>
<td>3091</td>
<td>37</td>
<td>1724</td>
<td>3087</td>
<td>48</td>
<td>2235</td>
<td>1694</td>
<td>67</td>
<td>3103</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>11122</td>
<td>17</td>
<td>1177</td>
<td>11483</td>
<td>25</td>
<td>1707</td>
<td>4329</td>
<td>35</td>
<td>2389</td>
</tr>
<tr>
<td></td>
<td>[4G]-BLAKE architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[4G]-BLAKE architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLAKE-32</td>
<td>2805</td>
<td>53</td>
<td>1292</td>
<td>2754</td>
<td>70</td>
<td>1705</td>
<td>1217</td>
<td>100</td>
<td>2438</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>6812</td>
<td>31</td>
<td>1104</td>
<td>6054</td>
<td>40</td>
<td>1413</td>
<td>2389</td>
<td>50</td>
<td>1766</td>
</tr>
<tr>
<td></td>
<td>[1G]-BLAKE architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[1G]-BLAKE architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLAKE-32</td>
<td>958</td>
<td>59</td>
<td>371</td>
<td>960</td>
<td>68</td>
<td>430</td>
<td>390</td>
<td>91</td>
<td>575</td>
</tr>
<tr>
<td>BLAKE-64</td>
<td>1802</td>
<td>36</td>
<td>326</td>
<td>1856</td>
<td>42</td>
<td>381</td>
<td>939</td>
<td>59</td>
<td>533</td>
</tr>
</tbody>
</table>

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

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.

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

Table 3.3: Main features of the PIC18F2525

Features of the PIC18F2525 are summarized in Table 3.3. All details can be found on Wikipedia\(^4\) 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 \(c_0, \ldots, c_{15}\) (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
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

\[
\text{movFF } x_{hi}, \text{tmp} \\
\text{movFF } x_{mh}, x_{hi} \\
\text{movFF } x_{ml}, x_{mh} \\
\text{movFF } x_{lo}, x_{ml} \\
\text{movFF } \text{tmp}, x_{lo}
\]

while the code for a 7-bit rotation looks like

\[
\text{bcf } \text{STATUS}, \text{C} \\
\text{btfsc } x_{lo}, 0 \\
\text{bsf } \text{STATUS}, \text{C} \\
\text{rrcF } x_{hi} \\
\text{rrcF } x_{mh} \\
\text{rrcF } x_{ml} \\
\text{rrcF } x_{lo} \\
\text{movFF } x_{lo}, \text{tmp} \\
\text{movFF } x_{hi}, x_{lo} \\
\text{movFF } x_{mh}, x_{hi} \\
\text{movFF } x_{ml}, x_{mh} \\
\text{movFF } \text{tmp}, x_{ml}
\]

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

- \( \gg 16 \) needs 6 cycles
- \( \gg 12 \) needs 22 cycles
- \( \gg 8 \) needs 5 cycles
- \( \gg 7 \) needs 12 cycles
Below we detail the maximum cost of each line of the $G_i$ function:

- (76 cycles) $a \leftarrow a + b + (m_{\sigma r(2i)} \oplus c_{\sigma r(2i+1)})$
- (24 cycles) $d \leftarrow (d \oplus a) \gg 16$
- (24 cycles) $c \leftarrow c + d$
- (34 cycles) $b \leftarrow (b \oplus c) \gg 12$
- (67 cycles) $a \leftarrow a + b + (m_{\sigma r(2i+1)} \oplus c_{\sigma r(2i)})$
- (22 cycles) $d \leftarrow (d \oplus a) \gg 8$
- (24 cycles) $c \leftarrow c + d$
- (29 cycles) $b \leftarrow (b \oplus c) \gg 7$

The cycle count is different for $(b \oplus c) \gg 12$ and $(b \oplus c) \gg 7$ 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 $\nu$ (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 \texttt{gcc 4.1.0} with options \texttt{-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 \texttt{Hash} specified in NIST’s API, which includes

\(^6\text{See http://cr.yp.to/chacha.html}\)
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.

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

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 an 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

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

and the finalization of the compression function becomes

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

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

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

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 “PRF(hash, nonce)” approach, where the value “hash” is a universal hash of the message authenticated. UMAC authors propose to instantiate 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 HMAC(hash) with \(s = \text{nonce}\):

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

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

\[
\text{NMAC}_{k_1 \oplus k_2}(m) = H_{k_1}(H_{k_2}(m))
\]

where \(H_k\) is a hash function with initial value \(k\).

For combining BLAKE with NMAC, we recommend not to set directly \(IV \leftarrow k_i, i = 1, 2\), but instead \(IV \leftarrow \text{compress}(IV, i, 0, 0)\), 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 \( \text{Sign}(H(m)) \), the signer picks a random \( r \) and sends \( (\text{Sign}(H_r(m)), r) \) 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

\[
H(r \parallel (m^1 \oplus r) \parallel \ldots \parallel (m^{N-1} \oplus r)).
\]

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 $\sigma, \sigma'$ 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 $\sigma$ 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 $\sigma$ 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.
Table 5.1: Input of message words.

<table>
<thead>
<tr>
<th>Round</th>
<th>G₀</th>
<th>G₁</th>
<th>G₂</th>
<th>G₃</th>
<th>G₄</th>
<th>G₅</th>
<th>G₆</th>
<th>G₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>10</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>15</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>12</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>11</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>11</td>
<td>7</td>
<td>14</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>15</td>
<td>14</td>
<td>9</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

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) \(m_j, j \in \{0, \ldots, 15\}\); a function \(G_i\) computes

\[
\begin{align*}
a &\leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
d &\leftarrow (d \oplus a) \gg 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \gg 12 \\
a &\leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
d &\leftarrow (d \oplus a) \ll 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ll 7
\end{align*}
\]

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

\[
\begin{align*}
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ll 16 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ll 12 \\
a &\leftarrow a + b \\
d &\leftarrow (d \oplus a) \ll 8 \\
c &\leftarrow c + d \\
b &\leftarrow (b \oplus c) \ll 7
\end{align*}
\]

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 successfully 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:

\[
\begin{align*}
b & \leftarrow c \oplus (b \ll 7) \\
c & \leftarrow c - d \\
d & \leftarrow a \oplus (d \ll 8) \\
a & \leftarrow a - b - (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
b & \leftarrow c \oplus (b \ll 12) \\
c & \leftarrow c - d \\
d & \leftarrow a \oplus (d \ll 16) \\
a & \leftarrow a - b - (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>in\out</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4.6</td>
<td>11.7</td>
<td>10.0</td>
<td>6.5</td>
</tr>
<tr>
<td>b</td>
<td>6.6</td>
<td>14.0</td>
<td>11.5</td>
<td>8.4</td>
</tr>
<tr>
<td>c</td>
<td>2.4</td>
<td>6.6</td>
<td>4.8</td>
<td>2.4</td>
</tr>
<tr>
<td>d</td>
<td>2.4</td>
<td>8.4</td>
<td>6.7</td>
<td>3.4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>in\out</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4.4</td>
<td>9.9</td>
<td>8.2</td>
<td>6.3</td>
</tr>
<tr>
<td>b</td>
<td>6.3</td>
<td>12.4</td>
<td>9.8</td>
<td>8.1</td>
</tr>
<tr>
<td>c</td>
<td>1.9</td>
<td>3.9</td>
<td>2.9</td>
<td>1.9</td>
</tr>
<tr>
<td>d</td>
<td>1.9</td>
<td>4.9</td>
<td>3.9</td>
<td>2.9</td>
</tr>
</tbody>
</table>

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

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

**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 $v_0$:

<table>
<thead>
<tr>
<th>Column Step</th>
<th>Diagonal Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00000037, 00000000, 00000000, 00000000)</td>
<td>(00000027F, 10039015, 5002B070, C418A7D4)</td>
</tr>
<tr>
<td>(E06E0216, 00000000, 00000000, 00000000)</td>
<td>(66918C7, 1CBEED25, F1A8536F, C111AD29)</td>
</tr>
<tr>
<td>(37010B00, 00000000, 00000000, 00000000)</td>
<td>(F8D104F0, 6F08C6F9, 5F77131E, E4291FE7)</td>
</tr>
<tr>
<td>(37000700, 00000000, 00000000, 00000000)</td>
<td>(151703A7, 705002B0, F2C22207, 7F000170)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Column Step</th>
<th>Diagonal Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00000011, 00000000, 00000000, 00000000)</td>
<td>(00000101, 10001001, 10011010, 02202000)</td>
</tr>
<tr>
<td>(20220202, 00000000, 00000000, 00000000)</td>
<td>(40040040, 22022220, 00202202, 00222020)</td>
</tr>
<tr>
<td>(11010100, 00000000, 00000000, 00000000)</td>
<td>(01110010, 20020222, 01111101, 00111101)</td>
</tr>
<tr>
<td>(11000100, 00000000, 00000000, 00000000)</td>
<td>(01110001, 10100110, 22002200, 01001101)</td>
</tr>
</tbody>
</table>

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 $(80000000, 00000000, 80000000, 80008000) \rightarrow (80000000, 0, 0, 0)$.
the diffusion is delayed one step, as illustrated below:

\[
\begin{bmatrix}
80000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
\end{bmatrix}
\text{(weight 1)}
\]

\[
\begin{bmatrix}
800003E8 & 00000000 & 00000000 & 00000000 \\
00000000 & 0B573F03 & 00000000 & 00000000 \\
00000000 & 00000000 & AB9F819D & 00000000 \\
00000000 & 00000000 & 00000000 & E8800083 \\
\end{bmatrix}
\text{(weight 49)}
\]

\[
\begin{bmatrix}
8007E4A0 & 2075B261 & 18E78828 & 9800099E \\
5944FE53 & F178A22F & 86B0A66B & 936C73CB \\
A27F0D24 & 98D6929A & 4088A5FB & 2E39EDA3 \\
A08FF64 & 2AD374B7 & 281E7E88 & 1E9883E1 \\
\end{bmatrix}
\text{(weight 236)}
\]

\[
\begin{bmatrix}
80000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
\end{bmatrix}
\text{(weight 1)}
\]

\[
\begin{bmatrix}
80000018 & 00000000 & 00000000 & 00000000 \\
00000000 & 10310101 & 00000000 & 00000000 \\
00000000 & 00000000 & 18808080 & 00000000 \\
00000000 & 00000000 & 00000000 & 18800080 \\
\end{bmatrix}
\text{(weight 18)}
\]

\[
\begin{bmatrix}
80000690 & E1101206 & 0801B818 & B8000803 \\
1D217176 & 600FC064 & 60111212 & 221E6712 \\
90B8B886 & 16E12133 & 00888138 & 83389890 \\
90803886 & 17E01122 & 180801B8 & 83B88010 \\
\end{bmatrix}
\text{(weight 155)}
\]

\[
\begin{bmatrix}
44E4E456 & 133468BD & DBBDA164 & 0F649833 \\
4E20F629 & 563A9099 & A62F39E9 & 7773C0BE \\
FEB6F508 & AABDDBF9 & 3262E291 & 87A10D6A \\
3C2B867B & B603B05C & DA695123 & F88E8007 \\
\end{bmatrix}
\text{(weight 251)}
\]

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

\[
\begin{bmatrix}
80000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
00000000 & 00000000 & 00000000 & 00000000 \\
\end{bmatrix}
\text{(weight 1)}
\]

\[
\begin{bmatrix}
80000018 & 00000000 & 00000000 & 00000000 \\
00000000 & 10310101 & 00000000 & 00000000 \\
00000000 & 00000000 & 18808080 & 00000000 \\
00000000 & 00000000 & 00000000 & 18800080 \\
\end{bmatrix}
\text{(weight 18)}
\]

\[
\begin{bmatrix}
80000690 & E1101206 & 0801B818 & B8000803 \\
1D217176 & 600FC064 & 60111212 & 221E6712 \\
90B8B886 & 16E12133 & 00888138 & 83389890 \\
90803886 & 17E01122 & 180801B8 & 83B88010 \\
\end{bmatrix}
\text{(weight 155)}
\]

\[
\begin{bmatrix}
44E4E456 & 133468BD & DBBDA164 & 0F649833 \\
4E20F629 & 563A9099 & A62F39E9 & 7773C0BE \\
FEB6F508 & AABDDBF9 & 3262E291 & 87A10D6A \\
3C2B867B & B603B05C & DA695123 & F88E8007 \\
\end{bmatrix}
\text{(weight 251)}
\]

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 $v_0, \ldots, v_{15}$ 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

$$E_{m||s}(h) \oplus h \oplus (s||s)$$
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

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

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

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

That is, we need $v_0 = v_8, v_1 = v_9, \ldots, v_7 = v_{15}$, 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 \(c_0, \ldots, c_3\), 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 

\[
\text{compress}(m, h, s, t) = \text{compress}(m', h, s', t') = h,
\]

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 \(IV \leftarrow \text{compress}(IV, k, 0, 0)\). 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 \(\text{compress}(\cdot, \cdot, \cdot, t)\) defines a family of pseudorandom functions (PRF’s); to make clear the abstraction we’ll write \((F_t)\) 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):
   \[
   I = \{ F_t, \exists k \in \mathbb{N}^*, t = 512 \cdot k \leq 2^{64} - 512 \}
   \]

2. the final compressions, called to compress message blocks containing padding data:
   \[
   F = \{ F_0 \} \cup \{ F_t, \exists k \in \mathbb{N}^*, p \in \{1, \ldots , 511\}, t = 512 \cdot k + p < 2^{64} \}
   \]

A function in \( I \) is never the last in a chain of iterations. A function in \( F \) 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 \( \langle 10 \text{ bits} \rangle \cdot \ldots \cdot 01 \langle 10 \text{ bits} \rangle \)). Clearly, \(| I | = 2^{55} - 1 \) and \(| F | = 511 \cdot | I | \). Functions in \( F \) can be seen as playing a role of output filter, in the same spirit as the HMAC 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 output; 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 \( h_3 = \text{compress}(h_2, m_2, s, 0) \), 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 \( \text{compress}(h^2, m_2, s, 1024) \), and thus be different from \( \text{compress}(h^2, m_2, s, 0) \). 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
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.

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.
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

\[
v_l \oplus v_{l+8} = v'_l \oplus v'_{l+8}, \quad i = 0, \ldots, 7.
\]
Put differently, it suffices to find a message difference that leads after the rounds sequence to a difference of the form
\[
\begin{pmatrix}
v_0 \oplus v'_0 \\
v_4 \oplus v'_4 \\
v_8 \oplus v'_8 \\
v_{12} \oplus v'_{12} \\
v_1 \oplus v'_1 \\
v_5 \oplus v'_5 \\
v_9 \oplus v'_9 \\
v_{13} \oplus v'_{13} \\
v_2 \oplus v'_2 \\
v_6 \oplus v'_6 \\
v_{10} \oplus v'_{10} \\
v_{14} \oplus v'_{14} \\
v_3 \oplus v'_3 \\
v_7 \oplus v'_7 \\
v_{11} \oplus v'_{11} \\
v_{15} \oplus v'_{15}
\end{pmatrix}
= \begin{pmatrix}
\Delta_0 \\ \Delta_4 \\ \Delta_8 \\ \Delta_{12} \\ \Delta_1 \\ \Delta_5 \\ \Delta_9 \\ \Delta_{13} \\ \Delta_2 \\ \Delta_6 \\ \Delta_{10} \\ \Delta_{14} \\ \Delta_3 \\ \Delta_7 \\ \Delta_{11} \\ \Delta_{15}
\end{pmatrix}.
\]

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 \( h_i \oplus s_i \mod 4 = h'_i \oplus s'_i \mod 4 \), for \( i = 0, \ldots, 7 \), 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 \( \sigma_i \) 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 \((m_{\sigma_i(2i)} \oplus c_{\sigma_i(2i+1)})\) 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 \( h_i \oplus s_i \text{mod} 4 = h'_i \oplus s'_i \text{mod} 4 \), 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 Flo-\nrian Mendel and Martin Schläffer for their analysis of BLAKE’s predecessor LAKE, and Orr\nDunkelman for communicating us new results on HAIFA. Finally, we thank Peter Steigmeier for\nimplementing BLAKE on the 8-bit PIC processor.

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


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


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

\[
\begin{align*}
v_0 & \leftarrow v_0 + v_4 + (m_0 \oplus 85A308D3) \\
v_{12} & \leftarrow (v_{12} \oplus v_0) \gg 16 \\
v_8 & \leftarrow v_8 + v_{12} \\
v_4 & \leftarrow (v_4 \oplus v_8) \gg 12 \\
v_0 & \leftarrow v_0 + v_4 + (m_1 \oplus 243F6A88) \\
v_{12} & \leftarrow (v_{12} \oplus v_0) \gg 8 \\
v_8 & \leftarrow v_8 + v_{12} \\
v_4 & \leftarrow (v_4 \oplus v_8) \gg 7
\end{align*}
\]

where $85A308D3 = c_{o_0(2 \times 0 + 1)} = c_1$ and $243F6A88 = c_{o_0(2 \times 0)} = c_0$. Then $G_1(v_1, v_5, v_9, v_{13})$ computes

\[
\begin{align*}
v_1 & \leftarrow v_1 + v_5 + (m_2 \oplus 03707344) \\
v_{13} & \leftarrow (v_{13} \oplus v_1) \gg 16 \\
v_9 & \leftarrow v_9 + v_{13} \\
v_5 & \leftarrow (v_5 \oplus v_9) \gg 12 \\
v_1 & \leftarrow v_1 + v_5 + (m_3 \oplus 13198A2E) \\
v_{13} & \leftarrow (v_{13} \oplus v_1) \gg 8 \\
v_9 & \leftarrow v_9 + v_{13} \\
v_5 & \leftarrow (v_5 \oplus v_9) \gg 7
\end{align*}
\]

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

\[
\begin{align*}
v_3 & \leftarrow v_3 + v_4 + (m_{14} \oplus B5470917) \\
v_{14} & \leftarrow (v_{14} \oplus v_3) \gg 16 \\
v_9 & \leftarrow v_9 + v_{14} \\
v_4 & \leftarrow (v_4 \oplus v_9) \gg 12 \\
v_3 & \leftarrow v_3 + v_4 + (m_{15} \oplus 3F84D5B5) \\
v_{14} & \leftarrow (v_{14} \oplus v_3) \gg 8 \\
v_9 & \leftarrow v_9 + v_{14} \\
v_4 & \leftarrow (v_4 \oplus v_9) \gg 7
\end{align*}
\]

After $G_7(v_3, v_4, v_9, v_{14})$, the second round starts. Because of the round-dependent permuta-
tions, $G_0(v_0, v_4, v_8, v_{12})$ now uses the permutation $\sigma_1$ instead of $\sigma_0$, and thus computes

$$
\begin{align*}
    v_0 &\leftarrow v_0 + v_4 + (m_{14} \oplus \text{BE5466CF}) \\
    v_{12} &\leftarrow (v_{12} \oplus v_0) \gg 16 \\
    v_8 &\leftarrow v_8 + v_{12} \\
    v_4 &\leftarrow (v_4 \oplus v_8) \gg 12 \\
    v_0 &\leftarrow v_0 + v_4 + (m_{10} \oplus \text{3F84D5B5}) \\
    v_{12} &\leftarrow (v_{12} \oplus v_0) \gg 8 \\
    v_8 &\leftarrow v_8 + v_{12} \\
    v_4 &\leftarrow (v_4 \oplus v_8) \gg 7
\end{align*}
$$

Above, $14 = \sigma_1(2 \times 0) = \sigma_1(0)$, $10 = \sigma_1(2 \times 0 + 1) = \sigma_1(1)$, BE5466CF = $c_{10}$, and 3F84D5B5 = $c_{14}$. Applying similar rules, column steps and diagonal steps continue until the tenth round, which uses the permutation $\sigma_9$. 
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

```vhdl
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

51
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
  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"3DE01377"),
                          (x"BE5466CF"), (x"94E90C6C"),
                          (x"C0AC29B7"), (x"C97C50DD"),
                          (x"3F84D5B5"), (x"B5470917"));
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_uniform: for i in 15 downto 0 generate
    MxD(15-i) <= MxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
  end generate p_uniform;
  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);

begin -- MEMORY INPUTS

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  
);
KxDI => G5kxD, AxDO => G5A0xD, BxDI => G5B0xD, CxDI => G5C0xD, DxDO => G5D0xD);

u_gcomp6: gcomp
port map (
  AxDI => G2A0xD, BxDI => G3B0xD, CxDI => G0C0xD, DxDI => G1D0xD, MxDI => G6MxO,
  KxDI => G6kxD, AxDO => G6A0xD, BxDO => G6B0xD, CxDI => G6C0xD, DxDO => G6D0xD);

u_gcomp7: gcomp
port map (
  AxDI => G3A0xD, BxDI => G0B0xD, CxDI => G1C0xD, DxDI => G0D0xD, MxDI => G7MxO,
  KxDI => G7kxD, AxDO => G7A0xD, BxDO => G7B0xD, CxDI => G7C0xD, DxDO => G7D0xD);

-------------------------------------------------------------------------------
-- V MEMORY
-------------------------------------------------------------------------------

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

architecture hash of gcomp is

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

begin -- hash

  TK1 <= MxDI(WIDTH*2-1 downto WIDTH) xor KxDI(WIDTH*2-1 downto WIDTH);
  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_uniform4: for i in 0 to 3 generate
    SINxD(i) <= SxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
  end generate p_uniform4;

  p_uniform8: 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_uniform8;

  p_uniform16: for i in 0 to 15 generate
    VxD(i) <= VxDI(WWIDTH*(i+1)-1 downto WWIDTH*i);
  end generate p_uniform16;

end hash;
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

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;
      end when;
  end case;
end process fsm;
end hash;
```vhdl
when fin =>
  VALIDOUTxSO <= '1';
  STATExDN <= idle;
```

```vhdl
when others =>
  STATExDN <= idle;
end case;
end process fsm;
```
incF pointer2mc ; pointer now (2i+1)
movF pointer2mc ; load pointer into w
movlw high permut_table_c ; find c signum_r (2i+1) lowbyte address
movwf TBLPTRH
rlncf pointer2mc, w
movwf TBLPTRL
tblrd* ; table read here into TABLAT
movff TABLAT, FSR1L ; move address 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 ...
ingF tmpXOR_ml ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_mh ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry
addWFC a_lo, f ; ADD a, place result in a
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_ml ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_mh ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry
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 ...
ingF tmpXOR_mh ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry
addWFC a_ml, f ; ADD a, place result in a
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_mh ; then ... add carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry
movF PREINC1 ; content of c signum_r (2i+1) midhigh byte loaded in w
xorWF tmpXOR_mh, w ; midhigh byte [m signum_r (2i) XOR c signum_r (2i+1)]
addWFC b_mh, w ; ADD b with carry
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry
addWFC a_mh, f ; ADD a, place result in a
btfsc STATUS, C ; IF carrybit =1 ...
ingF tmpXOR_hi ; then ... add carry

60
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)]
addWF b_hi,w         ; ADD b with carry, but carry disappears in black hole
addWF a_hi,f         ; ADD a, place result in a

term_d1
    call compute_dxora
movFF d_hi,tmpXOR_hi
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

... next is d = d xor a \ll 16

term_c1
    call compute_c

term_b1
    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
        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_ml
        rlcF b_lo
        rlcF b_ml
        rlcF b_lo
        rlcF b_ml
        rlcF b_lo
        rlcF b_ml
        rlcF b_lo
        rlcF b_ml

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
movf TBLPTRH
rlncf pointer2mc, w
movf TBLPTRL
tblrd*            ; table read here into TABLAT
movff TABLAT, FSR0L ; move adress to pointer
movFF INDFO,tmpXOR_lo
movFF PREINC0,tmpXOR_ml
movFF PREINC0,tmpXOR mh
movFF PREINC0,tmpXOR_hi

61
decF pointer2mc ; pointer now (2i)
movF pointer2mc ; load pointer into w
movlw high permut_table_c ; find c signum_r (2i) low byte 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) 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

movF PREINCl ; content of c signum_r (2i) mid low byte loaded in w
xorWF tmpXOR_ml, w ; mid low 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

movF PREINCl ; content of c signum_r (2i) mid high byte loaded in w
xorWF tmpXOR_mh, w ; mid high 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

    call compute_dxora ;... next is d = d xor a <<< 8
    movFF d_hi,tmpXOR_hi
    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

call compute_bxorc ;... next is b = b xor c <<< 7

        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 ; temporarily 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

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
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[ 9] = state->salt32[1];
v[10] = state->salt32[2];
v[12] = state->salt32[4];
v[13] = state->salt32[5];
v[14] = state->salt32[6];
v[15] = state->salt32[7];
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[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[3] = state->salt32[3];
state->h32[4] = state->salt32[0];
state->h32[5] = state->salt32[1];
state->h32[7] = state->salt32[3];
return SUCCESS;
}
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 $000\ldots000$ with BLAKE-32 and BLAKE-28, and we hash the 1152-bit message $000\ldots000$ 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:**

<table>
<thead>
<tr>
<th>6A09E667</th>
<th>BB67AE85</th>
<th>3C6EF372</th>
<th>A54FF53A</th>
<th>510E527F</th>
<th>9805688C</th>
<th>1F83D9AB</th>
<th>5BEOCD19</th>
</tr>
</thead>
</table>

**Message block after padding:**

| 00800000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000001 | 00000008 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

**Salt and counter**

| 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|

**Initial state of $v$:**

<table>
<thead>
<tr>
<th>6A09E667</th>
<th>BB67AE85</th>
<th>3C6EF372</th>
<th>A54FF53A</th>
<th>510E527F</th>
<th>9805688C</th>
<th>1F83D9AB</th>
<th>5BEOCD19</th>
</tr>
</thead>
</table>

**State $v$ after 1 round:**

<table>
<thead>
<tr>
<th>E78B8DFE</th>
<th>15050AE7</th>
<th>CABCS992</th>
<th>D15E8984</th>
<th>066DFDF2</th>
<th>084E6663</th>
<th>A516C4B3</th>
<th>3390DE5B</th>
</tr>
</thead>
</table>

**State $v$ after 2 rounds:**

<table>
<thead>
<tr>
<th>26051FB7</th>
<th>09D188B7</th>
<th>3AEEF8A8</th>
<th>49850659</th>
<th>13E513EE</th>
<th>B37ED53E</th>
<th>16AC7B8</th>
<th>75AF6DF6</th>
</tr>
</thead>
</table>

**State $v$ after 5 rounds:**

<table>
<thead>
<tr>
<th>5AF61049</th>
<th>FD4A2ADC</th>
<th>5C1DBBDB</th>
<th>5BA19232</th>
<th>9A685791</th>
<th>2B3DD795</th>
<th>A84DF6D6</th>
<th>A1D508A3</th>
</tr>
</thead>
</table>

**State $v$ after 10 rounds:**

<table>
<thead>
<tr>
<th>9ED875FD</th>
<th>8286272E</th>
<th>AD20174</th>
<th>F1B0F1B7</th>
<th>37A1A6D3</th>
<th>CF90583A</th>
<th>B67E00D2</th>
<th>943A1F4F</th>
</tr>
</thead>
</table>

**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
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Salt and counter

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

Initial state of $v$:

6A09E667 BB67AE85 3C6EF372 A54FF53A 510E527F 9B05688C 1F83D9AB 5BE0CD19
243F6A85 85A308D3 13198A2E 03707344 A4093A22 299F33D0 082EFA98 EC4E6C89

State $v$ after 1 round:

CC8704B8 1AF5E97 448BD7A4 7D5ED80F 8D088192 8DF5C28F B11E631F 0AC6CEAB
01A455BA 43BAE3C3 C07C7DEC 4C912C63 6F8C5DFC 87FD02D0 D69B7B1 874125B6

State $v$ after 2 rounds:

D7ED8FC3 CC0A55F2 24014945 38A9D033 8DA19E93 9B91D76A 18E0448C C10A0DF6
FB350B3C D894B64E F1835175 D0DF8F37 54E0DFF8 B3131C53 64BC97A4 819DFE8A

State $v$ after 5 rounds:

6BB8EAA1 FB2D35B9 F1C87115 8CCED083 C3CCF47F EC295B60 18CF9A21 DC2AC833
1F87F8A1 759A65F0 EE2F791D 11410F9F 46C442D0 EC5BE440 DC9ED226 97E68BBC

State $v$ after 10 rounds:

58B76F7A 24300259 EA5BAE66 7ABECB5C BEEA0C3C 38251B69 527E30C0C 4EBFC5FA
BF73D485 8B538346 03C56421 D1B9147E 63662F6C 70E9EEB2

Intermediate hash value

60C0B511 D1E86926 69468911 54A2B2D2 EC613A62 72996744 8C36C068 D4917832

**Second compression**  Message block after padding:

00000000 00000000 80000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Salt and counter

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

Initial state of $v$:

60C0B511 D1E86926 69468911 54A2B2D2 EC613A62 72996744 8C36C068 D4917832
243F6A85 85A308D3 13198A2E 03707344 A4093A22 299F33D0 082EFA98 EC4E6C89

State $v$ after 1 round:

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

Intermediate hash value

60C0B511 D1E86926 69468911 54A2B2D2 EC613A62 72996744 8C36C068 D4917832
State $v$ after 2 rounds:

```
F4C6263D 7327094B D139C80D 18A95331 6211D241 1BA339FA 4F059AB4 AA1580E9
211995BC CE94B414 5391B476 6D480D9D 70988FB3 114F5AF1 8648B874 4F87AF38
```

State $v$ after 5 rounds:

```
ECFEE77A 1F878081 339A7A59 D4CED068 73649B08 A3ACE1DA A0B085A5 22CCBB53
27BD497 30FB68D3 0ACF6405 524F093A 14E976D7 DCC7C7B0 98EA099A A41ECBAA
```

State $v$ after 10 rounds:

```
74CBFCFA BC46AECD 8835BA12 FA9767EE E1AAF6A5 2394033A D433008D 897E05BB
9E68CD63 AE60243C C3592B10 B979EC7A B6AD289C 58A2B983 272EEF06 4BF407E4
```

Hash value output:

```
8A638488 C318C5A8 222A1813 174C36B4 BB6E45B8 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 00000000
```

Salt and counter:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Initial state of $v$:

```
C1059ED8 367CD507 3070DD17 F70E5939 FFC00B31 68581511 64F98FA7 BEFA4FA4
243F6A88 85A308D3 131982A2 03707344 A409382A 299F31D8 082EFA98 EC46E639
```

State $v$ after 1 round:

```
04027914 24CFDD6B 7D33F394 12CBCC67 2DE38C62 6664F3D3 1D8D68FC D6CD0B08
481423A7 2F45B4F9 21C35492 50FB35FE 1255AE24 DFF2A626 9240D453 E8530B9D
```

State $v$ after 2 rounds:

```
9FB36742 31BC5AC2 064D4095 4A2260B2 C12165D2 00DD0EE58 AD1D8245 4F7B0F17
36EF0086 38DF9AE5 A67CC4B5 20963EEB F2B21838 D01907D2 7D15E12D 9B9EF864
```

State $v$ after 5 rounds:

```
AA629F77 16DE6E4A 5E78A622 257E83EC 8669EA65 99D687FD A632EA5E 511B1C46
93068AB9 67EA727C 5EC49C9A 7212CD8A 7F90526F E89825D4 70E30791 16C1EBDD
```

State $v$ after 10 rounds:

```
C9E1652F B9E85BDE 660E702E 67FC6579 BE6BC47F F5F0749A 1DFE158F 3B49131F
62A1B43D E2D6F00A 67AAA716 E006A66D 95565F38 8145A426 1ECD4DE7 ECF75FF74
```

Hash value output:

```
6A454FCA 6E347ED3 31D40A2F 70F49A2D D4FE2B86 1CEDC5AD 67C34456
```
Two-block message

IV:

\[
\begin{align*}
\text{C1059ED8} & \quad 367CD507 & \quad 3070DD17 & \quad F70E5939 & \quad FFC00B31 & \quad 68581511 & \quad 64F98FA7 & \quad \text{BEFA4FA4} \\
\end{align*}
\]

First compression  Message block after padding:

\[
\begin{align*}
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
\end{align*}
\]

Salt and counter

\[
\begin{align*}
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
\end{align*}
\]

Initial state of \( v \):

\[
\begin{align*}
\text{C1059ED8} & \quad 367CD507 & \quad 3070DD17 & \quad F70E5939 & \quad FFC00B31 & \quad 68581511 & \quad 64F98FA7 & \quad \text{BEFA4FA4} \\
243F6A88 & \quad 85A308D3 & \quad 13198A2E & \quad 03707344 & \quad A4093A22 & \quad 299F33D0 & \quad \text{082EFA98} & \quad \text{EC46CB89} \\
\end{align*}
\]

State \( v \) after 1 round:

\[
\begin{align*}
\text{E5B52991} & \quad 1FB7E4CB & \quad \text{F7350E64} & \quad 08D11C6 & \quad 148B1E94 & \quad \text{7C688FED} & \quad \text{C8EEE1B} & \quad \text{4046AC6E} \\
\text{8B84F36C} & \quad \text{C1C7F8EC} & \quad 1FA6AEC5 & \quad \text{EE4DC034} & \quad \text{87863887} & \quad \text{2D70805B} & \quad \text{4FA9A232} & \quad \text{D9860F12} \\
\end{align*}
\]

State \( v \) after 2 rounds:

\[
\begin{align*}
\text{2F3A90E3} & \quad \text{EBBC331} & \quad 5737A2D1 & \quad 6480F282 & \quad \text{DB711B83} & \quad 43014ABD & \quad \text{88924F03} & \quad \text{5160CB72} \\
\text{6E8F7EEB} & \quad \text{115D1FD6} & \quad \text{4387C5F} & \quad \text{FBB59797} & \quad \text{F8663D1A} & \quad \text{D5FA0EC9} & \quad \text{0C0ED9E5} & \quad \text{8579D4A6} \\
\end{align*}
\]

State \( v \) after 5 rounds:

\[
\begin{align*}
\text{F729608D} & \quad 8119B461 & \quad \text{E62F4D54} & \quad \text{7889D045} & \quad \text{838FB07D} & \quad \text{1A1E5E18} & \quad \text{872C02B} & \quad \text{E973E337} \\
\text{06F32665} & \quad \text{23B502C7} & \quad \text{FEDC26FC} & \quad \text{CEFD14A6} & \quad \text{DAD6B58F} & \quad \text{4DCA0D19} & \quad \text{31D904CB} & \quad \text{3C7E2160} \\
\end{align*}
\]

State \( v \) after 10 rounds:

\[
\begin{align*}
\text{D3465C90} & \quad \text{9AF5DB6} & \quad \text{77044D06} & \quad \text{8782E7B8} & \quad \text{F53CF50A} & \quad \text{78A3A751} & \quad \text{D7923EF6} & \quad \text{647B0D32} \\
\text{7B8026EF} & \quad \text{21577A7A} & \quad \text{CE253568} & \quad \text{1B6A082B} & \quad \text{D5E512E2} & \quad \text{E213D8E0} & \quad \text{F9651A7} & \quad \text{F9FDE6BE} \\
\end{align*}
\]

Intermediate hash value

\[
\begin{align*}
\text{69C34027} & \quad \text{8DDE22CB} & \quad \text{8951A579} & \quad \text{6BE6B6AA} & \quad \text{DFE6ED99} & \quad \text{F2E86AA0} & \quad \text{40FDE0F6} & \quad \text{237C6CF8} \\
\end{align*}
\]

Second compression  Message block after padding:

\[
\begin{align*}
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
\end{align*}
\]

Salt and counter

\[
\begin{align*}
00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 & \quad 00000000 \\
\end{align*}
\]

Initial state of \( v \):

\[
\begin{align*}
\text{69C34027} & \quad \text{8DDE22CB} & \quad \text{8951A579} & \quad \text{6BE6B6AA} & \quad \text{DFE6ED99} & \quad \text{F2E86AA0} & \quad \text{40FDE0F6} & \quad \text{237C6CF8} \\
\text{243F6A88} & \quad \text{85A308D3} & \quad \text{13198A2E} & \quad \text{03707344} & \quad \text{A4093A22} & \quad \text{299F33D0} & \quad \text{082EFA98} & \quad \text{EC46CB89} \\
\end{align*}
\]

State \( v \) after 1 round:

\[
\begin{align*}
\text{215AE8B6} & \quad \text{8A40E284} & \quad \text{8889C5CF} & \quad \text{3A7A9F3F} & \quad \text{3ECC4417} & \quad \text{4EB11689} & \quad \text{3B06106F} & \quad \text{0982D184} \\
\text{7F047CFA} & \quad \text{BCBFA0C8} & \quad \text{8E907E6C} & \quad \text{582C5CC4} & \quad \text{C7C016E8} & \quad \text{696F917E} & \quad \text{0AF46854} & \quad \text{929FD9AB} \\
\end{align*}
\]
State \( v \) after 2 rounds:

\[
\begin{align*}
998F9380 & \quad 6DC16FD & \quad 79CE8034 & \quad 6SB3E4A4 & \quad 459C22CC & \quad 388EA998 & \quad 35638B8B & \quad D9F54BB2 \\
A3C7177D & \quad A3E59D0B & \quad A059BBAF & \quad C62D9E5A & \quad B1A280BE & \quad 9032CCCB & \quad B36DB002 & \quad ECDC6D0D
\end{align*}
\]

State \( v \) after 5 rounds:

\[
\begin{align*}
2E967A8A & \quad 65885CE5 & \quad 8218A5B6 & \quad CFBA4356 & \quad 32627515 & \quad 913CB1C0 & \quad F80A1AE & \quad B524AE3A \\
643AE882 & \quad 4195A0AA & \quad 74CDF767 & \quad CFC048DF & \quad 2FDDA24A & \quad 42651292 & \quad 2B4A4CE2 & \quad 87B83536
\end{align*}
\]

State \( v \) after 10 rounds:

\[
\begin{align*}
2C975117 & \quad 5D90E6A5 & \quad 78A0F5C4 & \quad FB0EDE6F & \quad E88CE2F8 & \quad 03206935 & \quad CD05A414 & \quad 05F47C03 \\
2B9CC580 & \quad 2E07DFA & \quad A110229E & \quad DCE37F48 & \quad 4E31D239 & \quad 23EC233D & \quad D697DF58 & \quad 86F74FCC
\end{align*}
\]

Hash value output:

\[
\begin{align*}
6EC8D4B0 & \quad FEAE494 & \quad 50E17223 & \quad 4C0B178E & \quad 795BDC18 & \quad D22420A8 & \quad 5B6F9BB9
\end{align*}
\]

C.3 BLAKE-64

One-block message

Message block after padding:

\[
\begin{align*}
0080000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000
\end{align*}
\]

IV:

\[
\begin{align*}
6A09E667F3EC908 & \quad BB67AEB8S4CAA73B & \quad 3C6EF372FE94F82B & \quad A54FF53A5F1D36F1 \\
510527FADE682D1 & \quad 9B05688C2E36E61F & \quad 1F83D9ABFB411DB6 & \quad 5BEC0D19137E2179
\end{align*}
\]

Salt and counter

\[
\begin{align*}
0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000 & \quad 0000000000000000
\end{align*}
\]

Initial state of \( v \):

\[
\begin{align*}
6A09E667F3EC908 & \quad BB67AEB8S4CAA73B & \quad 3C6EF372FE94F82B & \quad A54FF53A5F1D36F1 \\
510527FADE682D1 & \quad 9B05688C2E36E61F & \quad 1F83D9ABFB411DB6 & \quad 5BEC0D19137E2179 \\
243F6A8865A308D3 & \quad 03F92332A668036B & \quad 02A5DDF1AFF95A3A
\end{align*}
\]

State \( v \) after 1 round:

\[
\begin{align*}
98957836D61905B3 & \quad 206435713954E43 & \quad 391FB64BD757FB63 & \quad A77C0E008BE362B5 \\
86DAB6C41F60C7E1 & \quad 823F50063BEE147C & \quad 6E6FC038D3B870 & \quad D93165F347733DF \\
D9D948AS1DE686F & \quad 3B73BB8B50C22B1 & \quad 03F92332A668036B & \quad E2F0869EE636BB9 \\
A4010390A3FD2AE & \quad 016613AD1A47C604 & \quad BFBC229C6E28B76 & \quad 02A5DDF1AFF95A3A
\end{align*}
\]

State \( v \) after 2 rounds:

\[
\begin{align*}
84DAC4B310F876B & \quad 01CE15A3A8D8BB2 & \quad F12C708C9D10A8B0 & \quad 77C288779642198 \\
13D4F878F3C03F5E & \quad 5B049744B1932015 & \quad 0FCFC0DEEE2CF40A & \quad 0BB67926A85E5AD8 \\
80D3E3FB6C957E2E8 & \quad A186830B9171C7 & \quad 06D755881837E80F & \quad B8792CFED5112CA0 \\
9226A72D98AD1F76 & \quad 8265C6B6C126BC1 & \quad COBFC6C6EECCFF19B & \quad E48AB82EAE436A
\end{align*}
\]

State \( v \) after 5 rounds:

\[
\begin{align*}
EF689A66BDC0A95 & \quad 2253DDE0C505B8FFC & \quad 886B8A405AE244FA & \quad CA317DF422522691 \\
F5123641D9F58E7 & \quad 17EEF7C5FD09F586 & \quad 8E20FEBBD491D29 & \quad E34A0ACDF256D303 \\
6D4719E51FA0833 & \quad 2721A86BD7D4BC0 & \quad 922783EA1497AD64 & \quad 72B2C92255DB2F9 \\
855C5D1C4D5D57A4 & \quad FC1340AE5577E39 & \quad 03B57F927BE21FC8 & \quad B43F42F4AA368791
\end{align*}
\]
State $v$ after 14 rounds:

1C803AADBC03622B 055EB72E5A0615B3 4624E5B1391E8A33 7B2A7AA93E27710A
F7EA864EFD591DF7 34E2FF788DBD71A7 01D13A3673488668 390D346D5CB82ECF
00D6AC4E1B3D8DE0 58CD6E304B8AD357 33E864217D9C1147 C9C6864A3790D49F
8C76318C3B9E3C07 20952009E26AE7A1 E63865AEC6B7E10C 2FAFFDCB74ADE2DE

Hash value output:

765F7084548226C3 E6F4779B954661DF 49A272E2BA16635F 1A3093756AA9364
2A92E58DDB21A321 87F2B7FD44E9FA19 F86A86334EBEAD0F 4D204B4F3BE6ED68

Two-block message

IV:

6A09E667F3BCC908 BB67AE8584CAA73B 3C6EF372FE94F82B A54FF53A5F1D36F1
510E527FADE682D1 9B05688C2B3E6C1F 1F83D9ABFB41BD6B 580ECD1913E2179

First compression  Message block after padding:

Salt and counter

Initial state of $v$:

State $v$ after 1 round:

State $v$ after 2 rounds:

State $v$ after 5 rounds:

State $v$ after 14 rounds:
Intermediate hash value:

2BEBCF2EC9AB9D4 AEAEF6EB309EE695A 85741F419946F883 C336E965CAD4AAD0
AD6CD62D18F4223 95BA7D2F338DF7FD 36603D22892EA9B 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 \(v\):

2BEBCF2EC9AB9D4 AEAEF6EB309EE695A 85741F419946F883 C336E965CAD4AAD0
AD6CD62D18F4223 95BA7D2F338DF7FD 36603D22892EA9B 3F6C6677F416F450
2436E88858308D3 1319B8E0E3707344 A409822299F31D0 082EFA98EC46C89
452821E638D017F7 BE5666CF34E908EC C0AC2987C975C0DD 3F84D5BSB5479017

State \(v\) after 1 round:

97B7744F66047D30 EFA6C7255A85A64 1826EF181027DF0 5845FCE8352477AA
3945C405020094E4 BF2E239191F3FB2A F52AFF3F0E1C94 O3D3936F6D90428
C965F095FF595911 BAC2B96500E645A 043F7421E6185D06 F65041D944A3383
4573F2E42626EDC79 9C1FDEEE8C387F1F6 E362DA8A81C065 972A9B182816E3C

State \(v\) after 2 rounds:

77DDF1D318481AF5 0E5B6B7B3A077AB3 52AC32E7820E8C4 9F2D723DFB2590B0E
AA80CFDA3D1AC0EC 85AC17EB7D90CB3C 457BA2C250B182B F70ADDEDE7FEB95B
4B81458C0391D37 8C8035CE83322CCE 5160F2E0762575C9 5F1A457FC0A34518
B033BABB0B0B947 4690E832AC7037B1 A8641F193796A0AF E0C40F4CDA8553A

State \(v\) after 5 rounds:

4E1F57385697E55 851DDE13C3C990D8 AA98BC1621BFDB4C 24308C6892728BE
C72F23D922B7805 5CAB7B8581FC011F ADA92906C20920C02 32CB00006666FBE
2A6849090FC886D FFA2A13A3CC310DF ECA63E2297722CD8 95D5C2CCF01C274
AF20A3721A949B 63C461C3134774B6 48C942D2E800355 D5BF2537AA44AE1

State \(v\) after 14 rounds:

60E7F9796F603C5 EB78F18831CFFE8 1207B65336348F8E D380A238CA002C04
87CE47BA3E608B8 568B3E38BD009077 5D4147CB6987380 504CD06EA9E16AF
A1B38091204CB914 3424EFBE7293F03 29CF1C0F0A356568 A7A86D76E2B3CF1
19F87A0EB186D235 715873557832859 C99F5DEF5E6170F 0A074F68BF273C1

Hash value output:

EAB7302804282105 71FS8FDE678A9B1 BBF58DF55471265 B71E262BB8FEFFA25
3C15317C3E99BF97 26969D4146AED0F3 A29827060055CA14 652753EFE20A913E
C.4 BLAKE-48

One-block message

Message block after padding:

\[
\begin{align*}
0080000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000000 \\
0000000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000000 \\
0000000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000000 \\
0000000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000008 \\
\end{align*}
\]

IV:

\[
\begin{align*}
\text{CBBBD} & \text{D1059ED8} & \text{629A292A367CD507} & \text{9159015A3070DD17} & \text{152FECD8F70E5939} \\
\text{67332667FCC00831} & \text{8EB44A8768581511} & \text{DB0C2ED64F98FAD7} & \text{47B5481DBEFA4FA4} \\
\end{align*}
\]

Salt and counter

\[
\begin{align*}
0000000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000000 \\
0000000000000000 & 0000000000000000 & 0000000000000000 & 0000000000000000 \\
\end{align*}
\]

Initial state of \(v\):

\[
\begin{align*}
\text{CBBBD} & \text{D1059ED8} & \text{629A292A367CD507} & \text{9159015A3070DD17} & \text{152FECD8F70E5939} \\
\text{67332667FCC00831} & \text{8EB44A8768581511} & \text{DB0C2ED64F98FAD7} & \text{47B5481DBEFA4FA4} \\
\text{243F6A885A30BD3} & \text{131982E03707344} & \text{A409382299F31DD0} & \text{082EFA98EC48C89} \\
\text{452821E638D0137F} & \text{BE5466CF34E90C64} & \text{COAC2987C97C50DD} & \text{3F84D6B5B470917} \\
\end{align*}
\]

State \(v\) after 1 round:

\[
\begin{align*}
\text{5B063A0F51A479BB} & \text{82C1717B7A4F69F4} & \text{4F58DFBA593FBBF} & \text{F826C578573BCE7E} \\
\text{C0836949C07A50A9} & \text{A4D8C74D27A8A0F5} & \text{7524F421149EFEF2} & \text{A94A548795A319EC} \\
\text{5EB1A73B8F91EE} & \text{3DF23E461EC546F9} & \text{F2C230E14FF4299} & \text{9191632BEE7E45E} \\
\text{C83CF461EDC796D} & \text{8F3F88591A781656} & \text{9BE2FO2DE19B8A} & \text{5844930E1FE8370D} \\
\end{align*}
\]

State \(v\) after 2 rounds:

\[
\begin{align*}
\text{5B2B7C158E6FEEA6} & \text{7413D0FE48C32BE2} & \text{535CA6F699C38D0} & \text{BBED0C0B5D30269} \\
\text{9E3CD99FC1168D0} & \text{A4D8C74D27A8A0F5} & \text{7524F421149EFEF2} & \text{A94A548795A319EC} \\
\text{B9F68999C61A94A6} & \text{EB0C499C541F99AA} & \text{260D24A2D818CBA3} & \text{BA391617A2D98EC} \\
\text{F78A66DC1AEB289C} & \text{9C362BCE597B9D9} & \text{74B326260C514D32} & \text{D53EB1184A89C053} \\
\end{align*}
\]

State \(v\) after 5 rounds:

\[
\begin{align*}
\text{4292D0F764CA9A5} & \text{17DF7CF80E7A6542} & \text{24CA7FE6607B393} & \text{C91DDCA2AFED146} \\
\text{7ECAF286B02C0DF7} & \text{00D475104786E1B9} & \text{FA12F69587EAF07B0} & \text{B2D845DA7D26918} \\
\text{A0E94F5B18546B4A} & \text{FBC986F9C13F1717} & \text{B4F94584075D75C4} & \text{BF9DC0E7F53657FF} \\
\text{CB09E853B9A1C13D} & \text{F4D6E7FE45A85E3} & \text{CE61CB91F8AAEF9} & \text{E29E502759826A4} \\
\end{align*}
\]

State \(v\) after 14 rounds:

\[
\begin{align*}
\text{1DD69F386C16DB30} & \text{EB4B1AD311C7C265} & \text{4204AA20151C2A0} & \text{1BD8CB637D2B52D} \\
\text{94ABF0918D497949} & \text{64591B873A8150B} & \text{56EE21C11395B066} & \text{008B540A4C94C03B} \\
\text{2E5D56650765B51} & \text{B85F78188E22A8D} & \text{514DF33128FAAC1} & \text{8E52CD242ADB8E8A} \\
\text{88EA3091A1873AA} & \text{DABF865D055D4AF} & \text{51168CA096930C62} & \text{E42652FBB6D559CF} \\
\end{align*}
\]

Hash value output:

\[
\begin{align*}
\text{F8A8D703FD654DB9} & \text{319AC478AF593DEF} & \text{821494CBE23AEB576} & \text{80A5EA1AE0A65CC} \\
\text{7B72E69F6893FED2} & \text{3E5233511EA5D425} & \text{74} \\
\end{align*}
\]

Two-block message

IV:

\[
\begin{align*}
\text{CBBBD} & \text{D1059ED8} & \text{629A292A367CD507} & \text{9159015A3070DD17} & \text{152FECD8F70E5939} \\
\text{67332667FCC00831} & \text{8EB44A8768581511} & \text{DB0C2ED64F98FAD7} & \text{47B5481DBEFA4FA4} \\
\end{align*}
\]
**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
```

**Initial state of v:**

```
CBBBD5C105ED8 629A292A367CD507 9159016A3070DD17 152F6C8D7F0E5939
6733267FC00B31 8E44A876858111 DB0CE0D64F9F8AF7 47B5481DBEFA4FAA
243F6885A308D3 13198AE03707344 A4093822299F31D0 082EFA98E4EC69
4E52821E638D0177 8E546CF34E9806C COAC29B7C97C50DD 3F84D6B5B470917
```

**State v after 1 round:**

```
3BBF567D6DE7C9A 826A179F64B2F2A D3598AB4A73A76F9 7FFB66FFAA07B8B4
1F78BFE2284B7162E D1F997BF243CD2A 70B6A23B3B32F2D 85416F6ECED2031
ADA8F2F0D0769947 C23086272083261 F6A871C70393F9FA 8D515B125606EDDA
C802F0CF294F6269 C5F36399DF7E1E35 8F20EDDF0A7D74A DE4472F1D1506E6F
```

**State v after 2 rounds:**

```
EA88A242A7F6CFC8 890A5C234B7CA8BF 5C8893D38EF6B3FB 46B0878A28D56B6E5
500B54433F1929C 8134381EE29381F 36505EC762DAB50C D71519E8814D4E39
4A22357959110F0F 58AD370D224CF9B0 47D1E79A61966B91 0563F8E3BA610BDE
4B6DE244317C9D00 C079DE27CBA8F3C8 DD1345A6384EAC 7E27A4AC04CF472D
```

**State v after 5 rounds:**

```
802C1FE2198A80 E6B5B8B836A1D70 8157BE2A7F87781D 9295E0C42C728FC
D88DF0EB4F00ADAB 781B1B5B4555CAB F89864B70E11F5F F015F543CB24E5F
014C1C7F0186ED 5A2B26F742DA21D0 3C0C3F7DFB0166DC 1142F58CF88765
0D2FB5CD1ADE0AE 7C972BBFE957FB5 7D874F260DD2E3FB 8CF8958C6233803
```

**State v after 14 rounds:**

```
48D32ABEC0D71CSC 435ACF7FB753BBF1 8AD951B5121E1F2 67D0D249D9A715A
AF9FDE1EE3CAD4D0 C66F145A899504DC 8439AEE5D81698D5 C74BCC12B51E1A
12D0217DE4E5B1 C77B5E254C52817 8636BF1D986E636B E5FDF66195146B0
16DA4C5878471174 CD4E5B050C98E92A 121004668DBAB665 AEFF5816CEA29F2
```

**Intermediate hash value:**

```
91B917CE0DA667AC E8B033B35EAA4E51 9DB8EFF2B900A8BE 9DA2CA7F73DC56E33
DE763C21644DCE48 857BE5D6ED55F6E7 4D26B4E3155A217 2E0DD6E9C4784C
```

---

**Second 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
```

**Second compression**  
**Message block after padding:**

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

---

75
Initial state of $v$:

$\begin{array}{cccccccc}
91B917CE0DA667AC & EDBB33B3D5EA45E1 & 9DB6EFF2B900AB8E & 9DA2CAF73DC56E83 \\
DE763C21644DCE48 & 85B7E5DBE556F6E7 & 4D26B48E3155A217 & 2E0DD68EC91784C \\
243F6A885A308D3 & 1319BAE203070344 & A4093822299F31D0 & 082EF984CE6C89 \\
452B81E63D017F7 & B8E566C3F4E908EC & CC9928C7C9750D0 & 3F84D65B5B470917 \\
\end{array}$

State $v$ after 1 round:

$\begin{array}{cccccccc}
EB5305AF9C67531E & B04F4367EF5BC01 & C5ACFFA4A502B3AC & 7B1494BE21EA8AE \\
EFC2114AF5B99E14 & 85C5D51A005E2343 & FB38714AE93CD0C4 & 730A928E549F309C \\
EB562A3636B5994 & 380D6D5F56E6E51 & 0C3A9D08930CE741 & C8996FA0C4FE476 \\
5406B1E65E8E0B04 & DE7BCC2A14B5687D & 1B9291C9E89CD45F & CC0AEDF77238F5A \\
\end{array}$

State $v$ after 2 rounds:

$\begin{array}{cccccccc}
EB5305AF9C67531E & 4CFDCDFA77330A4 & A3BEC556427F66DD0 & E61E7A015FB4065 \\
697C76A05841756C & C4238D4E0EF480C & 19244AE4F334FCE1 & 410666C930607C \\
4CD8F10D348326C4 & 8AC79266B6607D4 & FCB62721166BF27C & BF00A632885CC7ED \\
1B470C101AA73F07 & F21D3F36E0C97536 & C6A24BD865648A5 & OC9F27FDC4A89C1 \\
\end{array}$

State $v$ after 5 rounds:

$\begin{array}{cccccccc}
0B54F96A35B74457 & A4315CE1BO9ADE8D & E3078EA3D51F8EA4 & 453748C8FDED0071 \\
EADF5CCB9D038D & D9763C08677A4587 & BOE1A2243DEE4E974 & AFEA28B0BAE56C \\
BB57DF78BB4D038 & 7B0C7F7DE1E1F01 & 5EFE8B5E5E63A1D & 943FC16047631D4 \\
437715901F3DBBAF & 3AC592C75C0S56B & 0475414152111284 & 80397DEF4F32BEFB \\
\end{array}$

State $v$ after 14 rounds:

$\begin{array}{cccccccc}
767F77BF12F5C1B0 & 223D06220DB98FF2 & 332D0252DA9321D4 & CC96ECE63F1A08EA \\
3BCB8A5F7303E929 & 9AF6094763E64DE & 5790F88BF3C64982 & ADDA3E22C583C5 \\
2C45116E8AED40F & 5C35D58949ACF0D & 89406B7B3F0A86C0 & F814998AE3067F48 \\
073E11901769FF5 & 83F7A1741EE1F446 & 1A3F6053FDO8F90 & 082BE29A95035097 \\
\end{array}$

Hash value output:

$\begin{array}{cccc}
C802316791FD7C13 & 95D68C944CC9351E & 2BFA1785C990C9A & A920BF981611921 \\
E283A7660F07F894 & 9CFA4DEB2F8A667F & \end{array}$