Gimli

20190927

This submission is from the following team, listed in alphabetical order:

- Daniel J. Bernstein (corresponding submitter)
- Stefan Kölbl
- Stefan Lucks
- Pedro Maat Costa Massolino
- Florian Mendel
- Kashif Nawaz
- Tobias Schneider
- Peter Schwabe
- François-Xavier Standaert
- Yosuke Todo
- Benoît Viguier

E-mail address (preferred): `authorcontact-gimli@box.cr.yp.to`

Telephone (if absolutely necessary): +1-312-996-3422

Postal address (if absolutely necessary): Daniel J. Bernstein, Department of Computer Science, University of Illinois at Chicago, 851 S. Morgan (M/C 152), Room 1120 SEO, Chicago, IL 60607–7053.

Document generated with the help of `lwskeleton` version 20190104.

Contents

1 Introduction

Why isn't AES an adequate standard for lightweight cryptography? There are several common answers: for example, people point to the energy consumption of AES, the difficulty of protecting AES against side-channel attacks and fault attacks, and the inability of AES to support compact implementations.

Does this mean that we should design a new standard with the smallest possible hardware area, another new standard with the lowest possible energy consumption, etc.? The problem with this approach is that users often need cryptosystems to communicate between two different environments. If these two different environments are handled by two different standards then the users will not have *any* satisfactory solution.

The point of Gimli is to have *one* standard that performs very well in a broad range of environments. This includes many different lightweight environments. This also includes non-lightweight environments such as software for common server CPUs, so that there will not be a problem if (e.g.) many lightweight ASICs are all communicating with a central server.

2 Algorithm specification

2.1 Overview

This submission includes a family "Gimli-Cipher" of authenticated ciphers. This family includes the following recommended members, defined in detail below:

- `aead/gimli24v1` (primary): Gimli-24-Cipher with 256-bit key, 128-bit nonce, 128-bit tag.

This submission also includes a family "Gimli-Hash" of hash functions. This family includes the following recommended members, defined in detail below:

- `hash/gimli24v1` (primary): Gimli-24-Hash with 256-bit output.

These families are built on top of a family of 384-bit permutations called Gimli. This family includes the following recommended member: Gimli-24.

2.2 Formal and informal specifications

Bhargavan, Kiefer, and Strub recently introduced a language hacspec [12] for specifying cryptographic primitives. They wrote that "Specifications (specs) written in hacspec are

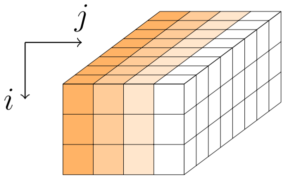


Figure 1: State Representation

succinct, easy to read and implement, and lend themselves to formal verification using a variety of existing tools."

In the rest of this section we follow the tradition of providing informal specifications. However, we also provide formal specifications of Gimli, Gimli-Hash, and Gimli-Cipher in hacspec. These specifications are the accompanying files `gimli.py`, `gimli_hash.py`, and `gimli_cipher.py`. Beware that the tag comparison in `gimli_cipher.py` is not performed in constant time; it is instead written as an early-abort loop for clarity.

2.3 Parameters

The authenticated cipher Gimli-Cipher and the hash function Gimli-Hash share one parameter: the number of rounds R in the Gimli permutation. We recommend $R = 24$, but we give definitions applicable to smaller and larger values of R. Time scales linearly in R with very little overhead.

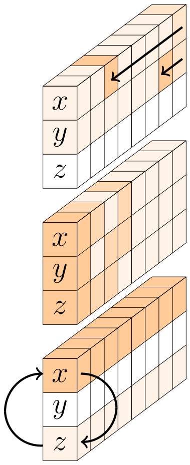
The name Gimli-24 refers to the Gimli permutation with 24 rounds. Similar comments apply to Gimli-24-Cipher and Gimli-24-Hash.

2.4 Notation

We denote by $\mathcal{W} = \{0,1\}^{32}$ the set of bitstrings of length 32. We will refer to the elements of this set as "words". We use

- $a \oplus b$ to denote a bitwise exclusive or (XOR) of the values a and b,
- $a \wedge b$ for a bitwise logical and of the values a and b,
- $a \vee b$ for a bitwise logical or of the values a and b,
- $a \lll k$ for a cyclic left shift of the value a by a shift distance of k, and
- $a \ll k$ for a non-cyclic shift (i.e, a shift that is filling up with zero bits) of the value a by a shift distance of k.

We index all vectors and matrices starting at zero. We encode words as bytes in little-endian form.

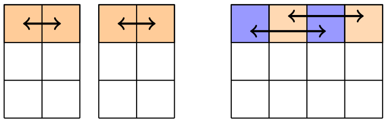


In parallel: $x \leftarrow x \lll 24$
$y \leftarrow y \lll 9$

In parallel:
$$\begin{aligned} x &\leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2) \\ y &\leftarrow y \oplus \quad x \quad \oplus ((x \vee z) \ll 1) \\ z &\leftarrow z \oplus \quad y \quad \oplus ((x \wedge y) \ll 3) \end{aligned}$$

In parallel: $x \leftarrow z$
$z \leftarrow x$

Figure 2: The SP-box applied to a column



Small Swap *Big Swap*

Figure 3: The linear layer

2.5 The Gimli state

The Gimli permutation applies a sequence of rounds to a 384-bit state. The state is represented as a parallelepiped with dimensions $3 \times 4 \times 32$ (see Fig. 1) or, equivalently, as a 3×4 matrix of 32-bit words.

We name the following sets of bits:

- a column j is a sequence of 96 bits such that $\mathbf{s}_j = \{s_{0,j}; s_{1,j}; s_{2,j}\} \in \mathcal{W}^3$
- a row i is a sequence of 128 bits such that $\mathbf{s}_i = \{s_{i,0}; s_{i,1}; s_{i,2}; s_{i,3}\} \in \mathcal{W}^4$

Each round is a sequence of three operations: (1) a non-linear layer, specifically a 96-bit SP-box applied to each column; (2) in every second round, a linear mixing layer; (3) in every fourth round, a constant addition.

2.6 The non-linear layer

The SP-box consists of three sub-operations: rotations of the first and second words; a 3-input nonlinear T-function; and a swap of the first and third words. See Figure 2 for details.

Algorithm 1 The Gimli permutation

```
Input: $\mathbf{s} = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
Output: Gimli($\mathbf{s}$) $= (s_{i,j}) \in \mathcal{W}^{3\times 4}$
  for $r$ from 24 downto 1 inclusive do
    for $j$ from 0 to 3 inclusive do
      $x \leftarrow s_{0,j} \lll 24$                                        ▷ SP-box
      $y \leftarrow s_{1,j} \lll 9$
      $z \leftarrow s_{2,j}$
      $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$
      $s_{1,j} \leftarrow y \oplus \quad x \quad \oplus ((x \vee z) \ll 1)$
      $s_{0,j} \leftarrow z \oplus \quad y \quad \oplus ((x \wedge y) \ll 3)$
    end for
                                                                            ▷ linear layer
    if $r \bmod 4 = 0$ then
      $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}$          ▷ Small-Swap
    else if $r \bmod 4 = 2$ then
      $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}$          ▷ Big-Swap
    end if

    if $r \bmod 4 = 0$ then
      $s_{0,0} = s_{0,0} \oplus \texttt{0x9e377900} \oplus r$                   ▷ Add constant
    end if
  end for
  return $(s_{i,j})$
```

2.7 The linear layer

The linear layer consists of two swap operations, namely *Small-Swap* and *Big-Swap*. *Small-Swap* occurs every 4 rounds starting from the 1st round. *Big-Swap* occurs every 4 rounds starting from the 3rd round. See Figure 3 for details of these swaps.

2.8 The round constants

There are R rounds in Gimli, numbered $R, R-1, \ldots, 1$. When the round number r is a multiple of 4 (e.g., $24, 20, 16, 12, 8, 4$ for $R = 24$), we XOR the round constant $\texttt{0x9e377900} \oplus r$ to the first state word $s_{0,0}$.

2.9 Putting it together: the Gimli permutation

Algorithm 1 is pseudocode for the full Gimli permutation.

Algorithm 2 The **absorb** function.

```
Input: $\mathbf{s} = (s_{i,j}) \in \mathcal{W}^{3\times 4}, m \in \mathbb{F}_{256}^{16}$
Output: $\mathbf{absorb}(\mathbf{s}, m) = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
  for $i$ from 0 to 3 do
    $s_{0,i} \leftarrow s_{0,i} \oplus \mathbf{touint32}(m_{4i}, \ldots, m_{4(i+1)})$
  end for
  $\mathbf{s} \leftarrow \textsc{Gimli}(\mathbf{s})$
  return $\mathbf{s}$
```

Algorithm 3 The **squeeze** function.

```
Input: $\mathbf{s} = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
Output: $\mathbf{squeeze}(\mathbf{s}) = h \in \mathbb{F}_{256}^{16}$
  $h \leftarrow \mathbf{tobytes}(s_{0,0})||\mathbf{tobytes}(s_{0,1})||\mathbf{tobytes}(s_{0,2})||\mathbf{tobytes}(s_{0,3})$
  return $h$
```

2.10 Hashing

Gimli-Hash initializes a 48-byte Gimli state to all-zero. It then reads sequentially through a variable-length input as a series of 16-byte input blocks.

Each full 16-byte input block is handled as follows:

- XOR the block into the first 16 bytes of the state (i.e., the top row of 4 words).
- Apply the Gimli permutation.

The input ends with exactly one final non-full (empty or partial) block, having b bytes where $0 \leq b \leq 15$. This final block is handled as follows:

- XOR the block into the first b bytes of the state.
- XOR 1 into the next byte of the state, position b.
- XOR 1 into the last byte of the state, position 47.
- Apply the Gimli permutation.

After the input is fully processed, a 32-byte hash output is obtained as follows:

- Output the first 16 bytes of the state.
- Apply the Gimli permutation.
- Output the first 16 bytes of the state.

Algorithm 4 The Gimli hash function.

```
Input: $M \in \{0,1\}^*$
Output: Gimli-Hash$(m) = h \in \{0,1\}^{256}$
  $\mathbf{s} \leftarrow \mathbf{0}$
  $m_1, \ldots, m_t \leftarrow \mathbf{pad}(M)$
  for $i$ from 1 to $t$ do
      if $i = t$ then
          $s_{2,3} \leftarrow s_{2,3} \oplus \texttt{0x01000000}$
      end if
      $\mathbf{s} \leftarrow \mathbf{absorb}(\mathbf{s}, m_i)$
  end for
  $h \leftarrow \mathbf{squeeze}(\mathbf{s})$
  $\mathbf{s} = \text{Gimli}(\mathbf{s})$
  $h \leftarrow h || \mathbf{squeeze}(\mathbf{s})$
  return $h$
```

A complete description is given in Alg. 4. Note that we make use of two functions **touint32**() and **tobytes**(). The former converts 4 bytes to a 32-bit unsigned integer in little-endian, while **tobytes**() converts a 32-bit unsigned integer to 4 bytes. Further, we use **pad**(M) to denote the padding of the message to full blocks.

2.11 Authenticated encryption

Gimli-Cipher initializes a 48-byte Gimli state to a 16-byte nonce followed by a 32-byte key. It then applies the Gimli permutation.

Gimli-Cipher then handles each block of associated data, including exactly one final non-full block, in the same way as Gimli-Hash.

Gimli-Cipher then handles each block of plaintext, including exactly one final non-full block, in the same way as Gimli-Hash. Whenever a plaintext byte is XORed into a state byte, the new state byte is output as ciphertext.

After the final non-full block of plaintext, the first 16 bytes of the state are output as an authentication tag.

See Algorithm 5 for an algorithm for authenticated encryption, and Algorithm 6 for an algorithm for verified decryption.

2.12 Combining authenticated encryption with hashing

Pairs of (authenticated cipher, hash function):

- `aead/gimli24v1` with `hash/gimli24v1`.

Algorithm 5 The Gimli AEAD encryption process.

```
Input: $M \in \{0,1\}^*, A \in \{0,1\}^*, N \in \mathbb{F}_{256}^{16}, K \in \mathbb{F}_{256}^{32}$
Output: Gimli-Cipher-encrypt$(M, A, N, K) = C \in \{0,1\}^*, T \in \mathbb{F}_{256}^{32}$
  Initialization
  $\mathbf{s} \leftarrow \mathbf{0}$
  for $i$ from 0 to 3 do
      $s_{0,i} \leftarrow \mathbf{touint32}(N_{4i}||\dots||N_{4i+3})$
      $s_{1,i} \leftarrow \mathbf{touint32}(K_{4i}||\dots||K_{4i+3})$
      $s_{2,i} \leftarrow \mathbf{touint32}(K_{16+4i}||\dots||K_{16+4i+3})$
  end for
  $\mathbf{s} \leftarrow$ Gimli$(\mathbf{s})$
  Processing AD
  $a_1, \dots, a_s \leftarrow \mathbf{pad}(A)$
  for $i$ from 1 to $s$ do
      if $i == s$ then
          $s_{2,3} \leftarrow s_{2,3} \oplus \texttt{0x01000000}$
      end if
      $\mathbf{s} \leftarrow \mathbf{absorb}(\mathbf{s}, a_i)$
  end for
  Processing Plaintext
  $m_1, \dots, m_t \leftarrow \mathbf{pad}(M)$
  for $i$ from 1 to $t$ do
      $k_i \leftarrow \mathbf{squeeze}(\mathbf{s})$
      $c_i \leftarrow k_i \oplus m_i$
      if $i == t$ then
          $s_{2,3} \leftarrow s_{2,3} \oplus \texttt{0x01000000}$
      end if
      $\mathbf{s} \leftarrow \mathbf{absorb}(\mathbf{s}, m_i)$
  end for
  $C \leftarrow c_1||\dots||c_t$
  $T \leftarrow \mathbf{squeeze}(\mathbf{s})$
  return $C, T$
```

Algorithm 6 The GIMLI AEAD decryption process.

```
Input: $C \in \{0,1\}^*, T \in \mathbb{F}_{256}^{16}, A \in \{0,1\}^*, N \in \mathbb{F}_{256}^{16}, K \in \mathbb{F}_{256}^{32}$
Output: $\textsc{Gimli-Cipher-decrypt}(C, T, A, N, K) = M \in \{0,1\}^*$
  Initialization
  $\mathbf{s} \leftarrow \mathbf{0}$
  for $i$ from 0 to 3 do
      $s_{0,i} \leftarrow \mathbf{touint32}(N_{4i}||\dots||N_{4i+3})$
      $s_{1,i} \leftarrow \mathbf{touint32}(K_{4i}||\dots||K_{4i+3})$
      $s_{2,i} \leftarrow \mathbf{touint32}(K_{16+4i}||\dots||K_{16+4i+3})$
  end for
  $\mathbf{s} \leftarrow \textsc{Gimli}(\mathbf{s})$
  Process AD
  $a_1, \dots, a_s \leftarrow \mathbf{pad}(A)$
  for $i$ from 1 to $s$ do
      if $i == s$ then
          $s_{2,3} \leftarrow s_{2,3} \oplus \texttt{0x01000000}$
      end if
      $\mathbf{s} \leftarrow \mathbf{absorb}(\mathbf{s}, a_i)$
  end for
  Process Ciphertext
  $c_1, \dots, c_t \leftarrow \mathbf{pad}(C)$
  for $i$ from 1 to $t$ do
      $k_i \leftarrow \mathbf{squeeze}(\mathbf{s})$
      $m_i \leftarrow k_i \oplus c_i$
      if $i == t$ then
          $s_{2,3} \leftarrow s_{2,3} \oplus \texttt{0x01000000}$
      end if
      $\mathbf{s} \leftarrow \mathbf{absorb}(\mathbf{s}, m_i)$
  end for
  $M \leftarrow m_1||\dots||m_t$
  $T' \leftarrow \mathbf{squeeze}(\mathbf{s})$
  if $T == T'$ then
      return $M$
  else
      return $\perp$
  end if
```

Common design components across these pairs: The authenticated cipher and the hash function use exactly the same Gimli permutation. In unified hardware providing authenticated encryption, verified decryption, and hashing, the hardware area for this permutation (including computations and storage) is entirely shared. There is very little work outside the permutation. Similar comments apply to software.

3 List of parameter sets

3.1 Parameter set `hash/gimli24v1`

Gimli-Hash with 24 rounds.

3.2 Parameter set `aead/gimli24v1`

Gimli-Cipher with 24 rounds.

4 Design rationale

This section explains how we arrived at the Gimli design.

We started from the well-known goal of designing one unified cryptographic primitive suitable for many different applications: collision-resistant hashing, preimage-resistant hashing, message authentication, message encryption, etc. We found no reason to question the "new conventional wisdom" that a permutation is a better unified primitive than a block cipher. Like Keccak, Ascon [17], etc., we evaluate performance only in the forward direction, and we consider only forward modes; modes that also use the inverse permutation require extra hardware area and do not seem to offer any noticeable advantages.

Where Gimli departs from previous designs is in its objective of being a single primitive that performs well on every common platform. We do not insist on beating all previous primitives on all platforms simultaneously, but we do insist on coming reasonably close. Each platform has its own hazards that create poor performance for many primitives; what Gimli shows is that all of these hazards can be avoided simultaneously.

4.1 Vectorization

On common Intel server CPUs, vector instructions are by far the most efficient arithmetic/-logic instructions. As a concrete example, the 12-round ChaCha12 stream cipher has run at practically the same speed as 12-round AES-192 on several generations of Intel CPUs (e.g., 1.7 cycles/byte on Westmere; 1.5 cycles/byte on Ivy Bridge; 0.8 cycles/byte on Skylake), despite AES hardware support, because ChaCha12 takes advantage of the vector hardware on the same CPUs. Vectorization is attractive for CPU designers because the overhead of fetching and decoding an instruction is amortized across several data items.

Any permutation built from (e.g.) common 32-bit operations can take advantage of a $32b$-bit vector unit if the permutation is applied to b blocks in parallel. Many modes of use of a permutation support this type of vectorization. But this type of vectorization creates two performance problems. First, if b parallel blocks do not fit into vector registers, then there is significant overhead for loads and stores; vectorized Keccak implementations suffer exactly this problem. Second, a large b is wasted in applications where messages are short.

Gimli, like Salsa and ChaCha, views its state as consisting of 128-bit rows that naturally fit into 128-bit vector registers. Each row consists of a vector of $128/w$ entries, each entry being a w-bit word, where w is optimized below. Most of the Gimli operations are applied to every column in parallel, so the operations naturally vectorize. Taking advantage of 256-bit or 512-bit vector registers requires handling only 2 or 4 blocks in parallel.

4.2 Logic operations and shifts

Gimli's design uses only bitwise operations on w-bit words: specifically, and, or, xor, constant-distance left shifts, and constant-distance rotations.

There are tremendous hardware-latency advantages to being able to carry out w bit operations in parallel. Even when latency is not a concern, bitwise operations are much more energy-efficient than integer addition, which (when carried out serially) uses almost $5w$ bit operations for w-bit words. Avoiding additions also allows "interleaved" implementations as in Keccak, Ascon, etc., saving time on software platforms with word sizes below w.

On platforms with w-bit words there is a software cost in avoiding additions. One way to quantify this cost is as follows. A typical ARX design is roughly balanced between addition, rotation, and xor. NORX [3] replaces each addition $a + b$ with a similar bitwise operation $a \oplus b \oplus ((a \wedge b) \ll 1)$, so 3 instructions (add, rotate, xor) are replaced with 6 instructions; on platforms with free shifts and rotations (such as the ARM Cortex-M4), 2 instructions are replaced with 4 instructions; on platforms where rotations need to be simulated by shifts (as in typical vector units), 5 instructions are replaced with 8 instructions. On top of this near-doubling in cost, the diffusion in the NORX operation is slightly slower than the diffusion in addition, increasing the number of rounds required for security.

The pattern of Gimli operations improves upon NORX in three ways. First, Gimli uses a third input c for $a \oplus b \oplus ((c \wedge b) \ll 1)$, removing the need for a separate xor operation. Second, Gimli uses only two rotations for three of these operations; overall Gimli uses 19 instructions on typical vector units, not far behind the 15 instructions used by three ARX operations. Third, Gimli varies the 1-bit shift distance, improving diffusion compared to NORX and possibly even compared to ARX.

We searched through many combinations of possible shift distances (and rotation distances) in Gimli, applying a simple security model to each combination. Large shift distances throw away many nonlinear bits and, unsurprisingly, turned out to be suboptimal. The final Gimli shift distances $(2, 1, 3$ on three 32-bit words) keep 93.75% of the nonlinear bits.

4.2.1 Bijectivity of Gimli

The bijectivity of the SP-box is not easy to see. If we exclude the swapping and the rotations (which are trivially bijective), we can unroll SP over the first bits:

$$f_0 = \begin{cases} x'_0 \leftarrow x_0 \\ y'_0 \leftarrow y_0 \oplus x_0 \\ z'_0 \leftarrow z_0 \oplus y_0 \end{cases}$$

$$f_1 = \begin{cases} x'_1 \leftarrow x_1 \oplus z_0 \\ y'_1 \leftarrow y_1 \oplus x_1 \oplus (x_0 \vee z_0) \\ z'_1 \leftarrow z_1 \oplus y_1 \end{cases}$$

$$f_2 = \begin{cases} x'_2 \leftarrow x_2 \oplus z_1 \oplus (y_0 \wedge z_0) \\ y'_2 \leftarrow y_2 \oplus x_2 \oplus (x_1 \vee z_1) \\ z'_2 \leftarrow z_2 \oplus y_2 \end{cases}$$

and

$$f_n = \begin{cases} x'_n \leftarrow x_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y'_n \leftarrow y_n \oplus x_n \oplus (x_{n-1} \vee z_{n-1}) \\ z'_n \leftarrow z_n \oplus y_n \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

Thus:

$$f_0^{-1} = \begin{cases} x_0 \leftarrow x'_0 & = x'_0 \\ y_0 \leftarrow y'_0 \oplus x_0 & = y'_0 \oplus x'_0 \\ z_0 \leftarrow z'_0 \oplus y_0 & = z'_0 \oplus y'_0 \oplus x'_0 \end{cases}$$

$$f_1^{-1} = \begin{cases} x_1 \leftarrow x'_1 \oplus z_0 & = x'_1 \oplus z_0 \\ y_1 \leftarrow y'_1 \oplus x_1 \oplus (x_0 \vee z_0) & = y'_1 \oplus x'_1 \oplus z_0 \oplus (x_0 \vee z_0) \\ z_1 \leftarrow z'_1 \oplus y_1 & = z'_1 \oplus y'_1 \oplus x'_1 \oplus z_0 \oplus (x_0 \vee z_0) \end{cases}$$

$$f_2^{-1} = \begin{cases} x_2 \leftarrow x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) & = x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \\ y_2 \leftarrow y'_2 \oplus x_2 \oplus (x_1 \vee z_1) & = y'_2 \oplus x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \\ z_2 \leftarrow z'_2 \oplus y_2 & = z'_2 \oplus y'_2 \oplus x'_2 \oplus z_1 \oplus (y_0 \wedge z_0) \oplus (x_1 \vee z_1) \end{cases}$$

and

$$f_n^{-1} = \begin{cases} x_n \leftarrow x'_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \\ y_n \leftarrow y'_n \oplus x'_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \oplus (x_{n-1} \vee z_{n-1}) \\ z_n \leftarrow z'_n \oplus y'_n \oplus x'_n \oplus z_{n-1} \oplus (y_{n-2} \wedge z_{n-2}) \oplus (x_{n-1} \vee z_{n-1}) \oplus (x_{n-3} \wedge z_{n-3}) \end{cases}$$

SP^{-1} is fully defined by recurrence. SP is therefore bijective.

4.3 32-bit words

Taking $w = 32$ is an obvious choice for 32-bit CPUs. It also works well on common 64-bit CPUs, since those CPUs have fast instructions for, e.g., vectorized 32-bit shifts. The 32-bit words can also be split into 16-bit words (with top and bottom bits, or more efficiently with odd and even bits as in "interleaved" Keccak software), and further into 8-bit words.

Taking $w = 16$ or $w = 8$ would lose speed on 32-bit CPUs that do not have vectorized 16-bit or 8-bit shifts. Taking $w = 64$ would interfere with Gimli's ability to work within a quarter-state for some time (see below), and we do not see a compensating advantage.

4.4 State size

On common 32-bit ARM microcontrollers, there are 14 easily usable integer registers, for a total of 448 bits. The 512-bit states in Salsa20, ChaCha, NORX, etc. produce significant load-store overhead, which Gimli avoids by (1) limiting its state to 384 bits (three 128-bit vectors), i.e., 12 registers, and (2) fitting temporary variables into just 2 registers.

Limiting the state to 256 bits would provide some benefit in hardware area, but would produce considerable slowdowns across platforms to maintain an acceptable level of security. For example, 256-bit sponge-based hashing at a 2^{100} security level would be able to absorb only 56 message bits (22% of the state) per permutation call, while 384-bit sponge-based hashing at the same security level is able to absorb 184 message bits (48% of the state) per permutation call, presumably gaining more than a factor of 2 in speed, even without accounting for the diffusion benefits of a larger state. It is also not clear whether a 256-bit state size leaves an adequate long-term security margin against multi-user attacks (see [18]) and quantum attacks; more complicated modes can achieve high security levels using small states, but this damages efficiency.

One of the SHA-3 requirements was 2^{512} preimage security. For sponge-based hashing this requires at least a 1024-bit permutation, or an even larger permutation for efficiency, such as Keccak's 1600-bit permutation. This requirement was based entirely on matching SHA-512, not on any credible assertion that 2^{512} preimage security will ever have any real-world value. Gimli is designed for useful security levels, so it is much more comparable to, e.g., 512-bit Salsa20, 400-bit Keccak-$f[400]$ (which reduces Keccak's 64-bit lanes to 16-bit lanes), 384-bit C-Quark [4], 384-bit SPONGENT-256/256/128 [13], 320-bit Ascon, and 288-bit Photon-256/32/32 [19].

4.5 Working locally

On the popular low-end ARM Cortex-M0 microcontroller, many instructions can access only 8 of the 14 32-bit registers. Working with more than 256 bits at a time incurs overhead to move data around. Similar comments apply to the 8-bit AVR microcontroller.

Gimli performs many operations on the left half of its state, and separately performs many operations on the right half of its state. Each half fits into 6 32-bit registers, plus 2 temporary registers.

It is of course necessary for these 192-bit halves to communicate, but this communication does not need to be frequent. The only communication is *Big-Swap*, which happens only once every 4 rounds, so we can work on the same half-state for several rounds.

At a smaller scale, Gimli performs a considerable number of operations within each column (i.e., each 96-bit quarter-state) before the columns communicate. Communication among columns happens only once every 2 rounds. This locality is intended to reduce wire lengths in unrolled hardware, allowing faster clocks.

4.6 Parallelization

Like Keccak and Ascon, Gimli has degree just 2 in each round. This means that, during an update of the entire state, all nonlinear operations are carried out in parallel: a nonlinear operation never feeds into another nonlinear operation.

This feature is often advertised as simplifying and accelerating masked implementations. The parallelism also has important performance benefits even if side channels are not a concern.

Consider, for example, software using 128-bit vector instructions to apply Salsa20 to a single 512-bit block. Salsa20 chains its 128-bit vector operations: an addition feeds into a rotation, which feeds into an xor, which feeds into the next addition, etc. The only parallelism possible here is between the two shifts inside a shift-shift-or implementation of the rotation. A typical vector unit allows more instructions to be carried out in parallel, but Salsa20 is unable to take advantage of this. Similar comments apply to BLAKE [5] and ChaCha20.

The basic NORX operation $a \oplus b \oplus ((a \wedge b) \ll 1)$ is only slightly better, depth 3 for 4 instructions. Gimli has much more internal parallelism: on average approximately 4 instructions are ready at each moment.

Parallel operations provide slightly slower forward diffusion than serial operations, but experience shows that this costs only a small number of rounds. Gimli has very fast backward diffusion.

4.7 Compactness

Gimli is intentionally very simple, repeating a small number of operations again and again. This gives implementors the flexibility to create very small "rolled" designs, using very little area in hardware and very little code in software; or to unroll for higher throughput.

This simplicity creates three directions of symmetries that need to be broken. Gimli is like Keccak in that it breaks all symmetries within the permutation, rather than (as in Salsa, ChaCha, etc.) relying on attention from the mode designer to break symmetries. Gimli puts more effort than Keccak into reducing the total cost of asymmetric operations.

The first symmetry is that rotating each input word by any constant number of bits produces a near-rotation of each output word by the same number of bits; "near" accounts for a few bits lost from shifts. *Occasionally* (after rounds 24, 20, 16, etc.) GIMLI adds an asymmetric constant to entry 0 of the first row. This constant has many bits set (it is essentially the golden ratio `0x9e3779b9`, as used in TEA), and is not close to any of its nontrivial rotations (never fewer than 12 bits different), so a trail applying this symmetry would have to cancel many bits.

The second symmetry is that each round is identical, potentially allowing slide attacks. This is much more of an issue for small blocks (as in, e.g., 128-bit block ciphers) than for large blocks (such as GIMLI's 384-bit block), but GIMLI nevertheless incorporates the round number r into the constant mentioned above. Specifically, the constant is `0x93e77900` $\oplus r$. The implementor can also use `0x93e77900` $+ r$ since r fits into a byte, or can have r count from `0x93e77918` down to `0x93e77900`.

The third symmetry is that permuting the four input columns means permuting the four output columns; this is a direct effect of vectorization. Occasionally (after rounds 24, 20, 16, etc.) GIMLI swaps entries $0, 1$ in the first row, and swaps entries $2, 3$ in the first row, reducing the symmetry group to 8 permutations (exchanging or preserving $0, 1$, exchanging or preserving $2, 3$, and exchanging or preserving the halves). Occasionally (after rounds 22, 18, 14, etc.) GIMLI swaps the two halves of the first row, reducing the symmetry group to 4 permutations (0123, 1032, 2301, 3210). The same constant distinguishes these 4 permutations.

We also explored linear layers slightly more expensive than these swaps. We carried out fairly detailed security evaluations of GIMLI-MDS (replacing a, b, c, d with $s \oplus a, s \oplus b, s \oplus c, s \oplus d$ where $s = a \oplus b \oplus c \oplus d$), GIMLI-SPARX (as in [16]), and GIMLI-Shuffle (with the swaps as above). We found some advantages in GIMLI-MDS and GIMLI-SPARX in proving security against various types of attacks, but it is not clear that these advantages outweigh the costs, so we opted for GIMLI-Shuffle as the final GIMLI.

4.8 Inside the SP-box: choice of words and rotation distances

The bottom bit of the T-function adds y to z and then adds x to y. We could instead add x to y and then add the new y to z, but this would be contrary to our goal of parallelism; see above.

After the T-function we exchange the roles of x and z, so that the next SP-box provides diffusion in the opposite direction. The shifted parts of the T-function already provide diffusion in both directions, but this diffusion is not quite as fast, since the shifts throw away some bits.

We originally described rotations as taking place after the T-function, but this is equivalent to rotation taking place before the T-function (except for a rotation of the input and output of the entire permutation). Starting with rotation saves some instructions outside the main loop on platforms with rotated-input instructions; also, some applications reuse portions of inputs across multiple permutation calls, and can cache rotations of those portions. These are minor advantages but there do not seem to be any disadvantages.

Rotating all three of x, y, z adds noticeable software cost and is almost equivalent to rotating only two: it merely affects which bits are discarded by shifts. So, as mentioned above, we rotate only two. In a preliminary Gimli design we rotated y and z, but we found that rotating x and y improves security by 1 round against our best integral attacks; see below.

This leaves two choices: the rotation distance for x and the rotation distance for y. We found very little security difference between, e.g., $(24, 9)$ and $(26, 9)$, while there is a noticeable speed difference on various software platforms. We decided against "aligned" options such as $(24, 8)$ and $(16, 8)$, although it seems possible that any security difference would be outweighed by further speedups.

4.9 Application to hashing

Gimli-Hash uses the well-known sponge mode [10, 9], the simplest way to hash using a permutation. A generic sponge with a 16-byte rate and a 32-byte capacity has 2^{128} security against a broad class of attacks.

By default, Gimli-Hash provides a fixed-length output of 32 bytes (the concatenation of two 16-byte blocks). However, Gimli-Hash can be used as an "extendable one-way function" (XOF). To generate n bytes of output,

- concatenate $\lceil n/16 \rceil$ blocks of 16 bytes, where each block is obtained by extracting the first 16 bytes of the state and then applying the Gimli permutation, and then
- truncate the resulting $16\lceil n/16 \rceil$ bytes to n bytes.

Note that Gimli-Hash applies the Gimli permutation with one empty input block, if the input length is a multiple of 16. The seemingly obvious alternative would be to have a 3-way fork after each block, e.g.:

- xor 0 into the capacity part after each non-final block,
- xor 1 into the capacity part after a full final block (i.e., if the size of the final block is exactly 16 bytes), and
- xor 2 into the capacity part after each partial final block (with an extra 1 at the end of the block).

This three-way fork saves one call of the permutation if the message length is a multiple of 16. However, the 2-way fork that we use has a performance benefit for lightweight applications.

Imagine a tiny device reading one byte (or even one bit) at a time, and at some point having the read instead say "end of data". With the 2-way fork, the device can handle each byte as follows:

- xor the byte into the state at the current position,
- increase the current position, and
- if the current position would exceed the end of the block, apply the permutation and set the current position back to the first byte.

Whenever the device receives an "end of data", it can immediately xor 1 into the state at the current position and apply the permutation.

With a 3-way fork, the device instead must delay calling the permutation at the end of each block until it knows whether the data is finished or not. If another byte arrives, the device must buffer that byte, perform the permutation, and then xor that byte into the block. This complicates the handling of every block.

We conclude that the two-way fork in Gimli-Hash is better suited for lightweight cryptosystems than the three-way fork, even though the three-way fork does save one application of the Gimli permutation for 1/16 of all message lengths.

4.10 Application to authenticated encryption

Gimli-Cipher uses the well-known duplex mode [11], the simplest way to encrypt using a permutation. Duplexing reads an input the same way as sponge hashing: each 16-byte message block m is added into the first block x of the state, changing this block of the state to $m + x$. Duplexing also outputs $m + x$ as a block of ciphertext.

We opted for a 256-bit key. This does not mean that we endorse the pursuit of (e.g.) a 2^{224} security level; it means that we want to reduce concerns about multi-target attacks, quantum attacks, etc.

NIST has recommended that 256-bit keys be accompanied by a 2^{224} single-key pre-quantum security level. We have considered various ways to accommodate this recommendation. For example, one can add 16 of the key bytes (which can be shared with the existing key bytes, as in single-key Even–Mansour) into each 16-byte ciphertext block. However, this requires the state storing the key to be accessed for each block, rather than just at the beginning of processing a message. In the absence of any explanation of any real-world relevance of security levels above 2^{128}, we have opted to avoid this complication.

We have also considered a mode called "Beetle", which is argued in [14] to achieve quantitatively higher security than duplexing. Beetle uses the key only at the beginning, and it involves only slightly more computation than duplexing:

- View the plaintext block as two halves: (m_0, m_1).
- View the first state block as (x_0, x_1).
- Output the ciphertext block $(m_0 + x_0, m_1 + x_1)$, as in duplexing.
- Replace the first state block with $(m_0 + x_1, m_1 + x_1 + x_0)$.

However, in environments that communicate (say) 1 bit at a time, it is not clear how to fit the Beetle mode into as little space as the duplex mode. The duplex mode allows each plaintext bit to be read, added into the state, output as a ciphertext bit with no further latency, and then forgotten, with a small (7-bit) counter keeping track of the position inside the block.

4.11 Other applications

A strong and small permutation, such as Gimli, has plenty of applications beyond the proposed Gimli-Hash and Gimli-Cipher. As an illustration, we explain how one can build independent hash functions for b-bit outputs, similarly to SHA-512/b, instead of one single extendable output function (XOF) for all output sizes.

A generic disadvantage of the XOF concept is that if one computes two hashes of different lengths from the same input, the shorter hash is just a prefix of the longer one. This may not matter for typical applications of an XOF, but in some hash applications, this can be really bad. E.g., if one uses the hash function to derive secret keys $K_1 = \mathrm{Hash}_k(X)$ and $K_2 = \mathrm{Hash}_\ell(X)$ of lengths k and $\ell > k$ from the same input X, one expects k-bit security, while the security level can acually drop down to $\ell - k < k$, for $\ell < 2k$.

Thus, for applications where this XOF-property is undesirable or dangerous, we propose a parameterized

$$\textsc{Gimli-Hash}_\ell \text{ (for } \ell \in \{0, 2^{32} - 1\}).$$

The approach is the same as for the SHA-512/(8ℓ) family of hash functions: Use the hash length parameter ℓ to define the initialization value or the initial state, then run the hash algorithm, and finally truncate the output to the given number of bytes:

1. Write ℓ as a 32-bit (i.e., 4-byte) number. As elsewhere in the Gimli specs, use little-endian form. Define the 48-byte initial state by the four bytes of ℓ, followed by 44 all-zero bytes.
2. Perform the Gimli-Hash as specified above.

3.a) If $\ell > 0$, generate ℓ output bytes.

3.b) The case $\ell = 0$ indicates the XOF mode. In that case, generate as many output bytes as required.

Note that the Gimli-Hash, as defined in Section 2.10, is identical to the special case of using Gimli-Hash$_0$ to generate 32 output bytes.

5 Expected strength in general

5.1 Hashing

5.1.1 Usage requirements

Each hash call is assumed to process at most 2^{64} bytes.

5.1.2 Security goals

The hash function is designed for collision resistance, preimage resistance, second-preimage resistance, and resistance against length-extension attacks.

See Section 6 for quantitative goals for specific parameter sets.

5.1.3 Rationale

See Section 7 for an analysis of known attacks.

5.2 Authenticated encryption

5.2.1 Usage requirements

Each cipher call is assumed to process at most 2^{64} bytes of plaintext and at most 2^{64} bytes of associated data. Each key is assumed to process a total of at most 2^{80} bytes of data. The total number of legitimate user keys is assumed to be at most 2^{80}. Foreseeable applications fit comfortably below these limits.

Keys are assumed to be chosen independently and uniformly at random.

By definition, a nonce is used at most once for each key: i.e., it is used with this key for only one (plaintext, associated data) pair. This rule is important because all ciphers lose security when public message numbers are not nonces, i.e., when public message numbers repeat. (Often the public message numbers are incorrectly called “nonces” even in this case, and the repetition of numbers is then called “nonce misuse”.) Some ciphers try to compete regarding the amount of damage done in this situation:

- The minimum possible damage is revealing patterns of repeated plaintexts with the same key and the same associated data.
- The maximum possible damage is a complete loss of confidentiality and integrity.

For this submission, the damage is intermediate. On the positive side, variations in the associated data have the same effect as variations in the public message numbers. On the negative side, when public message numbers and associated data repeat, the ciphertexts reveal the longest common prefix of plaintexts, and the xor of the block containing the first difference.

5.2.2 Security goals

The authenticated cipher is designed to protect confidentiality of plaintexts (under adaptive chosen-plaintext attacks) and integrity of ciphertexts (under adaptive forgery attempts).

See Section 6 for quantitative goals for specific parameter sets.

5.2.3 Rationale

See Section 7 for an analysis of known attacks.

6 Expected strength for each parameter set

6.1 Parameter set `hash/gimli24v1`

Security 2^{128} against all attacks.

6.2 Parameter set `aead/gimli24v1`

Security 2^{128} against all attacks.

7 Known attacks

7.1 Diffusion

As a first step in understanding the security of reduced-round Gimli, we consider the following two minimum security requirements:

- the number of rounds required to show the avalanche effect for each bit of the state.
- the number of rounds required to reach a *state full of* `1` starting from a state where only one bit is set. In this experiment we replace bitwise exclusive *or* (XOR) and bitwise logical *and* by a bitwise logical *or*.

Given the input size of the SP-box, we verify the first criterion with the Monte-Carlo method. We generate random states and flip each bit once. We can then count the number of bits flipped after a defined number of rounds. Experiments show that 10 rounds are required for each bit to change on the average half of the state (see tables in the original Gimli paper).

As for the second criterion, we replace the T-function in the SP-box by the following operations:

$$\begin{aligned} x' &\leftarrow x \vee (z \lll 1) \vee ((y \vee z) \lll 2) \\ y' &\leftarrow y \vee x \vee ((x \vee z) \lll 1) \\ z' &\leftarrow z \vee y \vee ((x \vee y) \lll 3) \end{aligned}$$

By testing the 384 bit positions, we prove that a maximum of 8 rounds are required to fill up the state.

7.2 Differential cryptanalysis

To study Gimli's resistance against differential cryptanalysis we use the same method as has been used for NORX [2] and Simon [23] by using a tool-assisted approach to find the optimal differential trails for a reduced number of rounds. In order to enable this approach we first need to define the valid transitions of differences through the Gimli round function.

The non-linear part of the round function shares similarities with the NORX round function, but we need to take into account the dependencies between the three lanes to get a correct description of the differential behavior of Gimli. In order to simplify the description we will look at the following function which only covers the non-linear part of Gimli:

$$f(x, y, z): \quad \begin{aligned} x' &\leftarrow y \wedge z \\ y' &\leftarrow x \vee z \\ z' &\leftarrow x \wedge y \end{aligned} \tag{1}$$

where $x, y, z \in \mathcal{W}$. For the Gimli SP-box we only have to apply some additional linear functions which behave deterministically with respect to the propagation of differences. In

Table 1: The optimal differential trails for a reduced number of rounds of Gimli.

Rounds	1	2	3	4	5	6	7	8
Weight	0	0	2	6	12	22	36	52

the following we denote $(\Delta_x, \Delta_y, \Delta_z)$ as the input difference and $(\Delta_{x'}, \Delta_{y'}, \Delta_{z'})$ as the output difference. The *differential probability* of a differential trail T is denoted as $\mathrm{DP}(T)$ and we define the weight of a trail as $w = -\log_2(\mathrm{DP}(T))$.

Lemma 1 (Differential Probability) *For each possible differential through f it holds that*

$$\begin{aligned}
\Delta_{x'} \wedge \neg(\Delta_y \vee \Delta z) &= 0 \\
\Delta_{y'} \wedge \neg(\Delta_x \vee \Delta z) &= 0 \\
\Delta_{z'} \wedge \neg(\Delta_x \vee \Delta y) &= 0 \\
(\Delta_x \wedge \Delta_y \wedge \neg\Delta_z) \wedge \neg(\Delta_{x'} \oplus \Delta_{y'}) &= 0 \\
(\Delta_x \wedge \neg\Delta_y \wedge \Delta_z) \wedge (\Delta_{x'} \oplus \Delta_{z'}) &= 0 \\
(\neg\Delta_x \wedge \Delta_y \wedge \Delta_z) \wedge \neg(\Delta_{y'} \oplus \Delta_{z'}) &= 0 \\
(\Delta_x \wedge \Delta_y \wedge \Delta_z) \wedge \neg(\Delta_{x'} \oplus \Delta_{y'} \oplus \Delta_{z'}) &= 0.
\end{aligned} \tag{2}$$

The differential probability *of $(\Delta_x, \Delta_y, \Delta_z) \xrightarrow{f} (\Delta_{x'}, \Delta_{y'}, \Delta_{z'})$ is given by*

$$\mathrm{DP}((\Delta_x, \Delta_y, \Delta_z) \xrightarrow{f} (\Delta_{x'}, \Delta_{y'}, \Delta_{z'})) = 2^{-2 \cdot \boldsymbol{hw}(\Delta_x \vee \Delta_y \vee \Delta_z)}. \tag{3}$$

A proof for this lemma is given in Proof 1. We can then use these conditions together with the linear transformations to describe how differences propagate through the Gimli round functions. For computing the differential probability over multiple rounds we assume that the rounds are independent. Using this model we then search for the optimal differential trails with the SAT/SMT-based approach [2, 23].

We are able to find the optimal differential trails up to 8 rounds of Gimli (see Table 1). After more rounds this approach failed to find any solution in a reasonable amount of time. The 8-round differential trail is given in Table 3 in Proof 1.

In order to cover more rounds of Gimli we restrict our search to a *good* starting difference and expand it in both directions. As the probability of a differential trail quickly decreases with the Hamming weight of the state it is likely that any high probability trail will contain some rounds with very low Hamming weight. In Table 2, we show the results when starting from a single bit difference in any of the words. Interestingly, the best trails match the optimal differential trails up to 8 rounds given in Table 1.

Using the optimal differential for 7 rounds we can construct a 12-round differential trail with probability 2^{-188} (see Table 4 in Proof 1). If we look at the corresponding differential, this means we do not care about any intermediate differences; many trails might contribute

Table 2: The optimal differential trails when expanding from a single bit difference in any of the words.

Rounds	1	2	3	4	5	6	7	8	9
$r = 0$	0	2	6	14	28	58	102		
$r = 1$	0	0	2	6	12	26	48	88	
$r = 2$	-	0	2	6	12	22	36	66	110
$r = 3$	-	-	8	10	14	32	36	52	74
$r = 4$	-	-	-	26	28	32	38	52	74

to the probability. In the case of our 12-round trail we find 15800 trails with probability 2^{-188} and 20933 trails with probability 2^{-190} contributing to the differential. Therefore, we estimate the probability of the differential to be $\approx 2^{-158.63}$.

Proof 1 (Proof of Lemma 1) *We want to show how to compute the set of valid differentials for a given input difference*

$$\{(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) : f(x, y, z) \oplus f(x \oplus \Delta_x, y \oplus \Delta_y, z \oplus \Delta_z) = (\Delta_{x'}, \Delta_{y'}, \Delta_{z'})\}. \tag{4}$$

It is sufficient to look at the case where $\mathcal{W}$ is $\mathbb{F}_2$ as there is no interaction between different coordinates in f. The output differences for f are given by

$$\begin{aligned} \Delta_{x'} &= (y \wedge z) \oplus (y \oplus \Delta_y \wedge z \oplus \Delta_z) \\ \Delta_{y'} &= (x \vee z) \oplus (x \oplus \Delta_x \vee z \oplus \Delta_z) \\ \Delta_{z'} &= (x \wedge y) \oplus (x \oplus \Delta_x \wedge y \oplus \Delta_y). \end{aligned} \tag{5}$$

If the input difference $(\Delta_x, \Delta_y, \Delta_z) = (0, 0, 0)$, then the output difference is clearly $(0, 0, 0)$ as well. We can split the remaining cases in three groups

Case 1 $(\Delta_x, \Delta_y, \Delta_z) = (1, 0, 0)$. *This simplifies Equation 5 to*

$$\begin{aligned} \Delta_{x'} &= (y \wedge z) \oplus (y \wedge z) = 0 \\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee z) = -z \\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge y) = y. \end{aligned} \tag{6}$$

and gives us the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)\}. \tag{7}$$

In a similar way we can find the differentials for the other cases with a single bit difference which gives us the first three conditions in Lemma 1.

Case 2 $(\Delta_x, \Delta_y, \Delta_z) = (1, 1, 0)$. *This simplifies Equation 5 to*

$$\begin{aligned} \Delta_{x'} &= (y \wedge z) \oplus (\neg y \wedge z) = z \\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee z) = -z \\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge \neg y) = \neg(x \oplus y). \end{aligned} \tag{8}$$

giving the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0,1,0), (0,1,1), (1,0,0), (1,0,1)\}. \tag{9}$$

Again we can derive the other two cases in a similar way, giving us conditions 4-6 in Lemma 1.

Case 3 $(\Delta_x, \Delta_y, \Delta_z) = (1,1,1)$. *This simplifies Equation 5 to*

$$\begin{aligned} \Delta_{x'} &= (y \wedge z) \oplus (\neg y \wedge \neg z) = \neg(y \oplus z) \\ \Delta_{y'} &= (x \vee z) \oplus (\neg x \vee \neg z) = \neg(x \oplus y) \\ \Delta_{z'} &= (x \wedge y) \oplus (\neg x \wedge \neg y) = \neg(x \oplus y). \end{aligned} \tag{10}$$

giving the set of possible output differences

$$(\Delta_{x'}, \Delta_{y'}, \Delta_{z'}) \in \{(0,0,1), (0,1,0), (1,0,0), (1,1,1)\}. \tag{11}$$

This corresponds to the last condition in Lemma 1.

As in all but the $(0,0,0)$ *cases the size of the set of possible output differences is 4 the probability of any differential transition is* 2^{-2}. □

.

7.3 Algebraic degree and integral attacks

Since the algebraic degree of the round function of Gimli is only 2, it is important how the degree increases by iterating the round function. We use the (bit-based) division property [31, 32] to evaluate the algebraic degree, and the propagation search is assisted by mixed integer linear programming (MILP) [35]. See Section 7.4.

We first evaluated the upper bound of the algebraic degree on r-round Gimli, and the result is summarized as follows.

# rounds	1	2	3	4	5	6	7	8	9
	2	4	8	16	29	52	95	163	266

When we focus on only one bit in the output of r-round Gimli, the increase of the degree is slower than the general case. Especially, the algebraic degree of z_0 in each 96-bit value is lower than other bits because z_0 in rth round is the same as x_6 in $(r-1)$th round. All bits except for z_0 is mixed by at least two bits in $(r-1)$th round. Therefore, we next evaluate the upper bound of the algebraic degree on four z_0 in r-round Gimli, and the result is summarized as follows.

Table 3: Optimal differential trail for 8-round Gimli.

Round	$s_{*,0}$	$s_{*,1}$	$s_{*,2}$	$s_{*,3}$	Weight
0	0x80404180	0x00020100	-	-	18
	0x80002080	-	-	-	
	0x80002080	0x80010080	-	-	
1	0x80800100	-	-	-	8
	0x80400000	-	-	-	
	0x80400080	-	-	-	
2	0x80000000	-	-	-	0
	0x80000000	-	-	-	
	0x80000000	-	-	-	
3	-	-	-	-	0
	-	-	-	-	
	0x80000000	-	-	-	
4	0x00800000	-	-	-	2
	-	-	-	-	
	-	-	-	-	
5	-	-	-	-	4
	0x00000001	-	-	-	
	0x00800000	-	-	-	
6	0x01008000	-	-	-	6
	0x00000200	-	-	-	
	0x01000000	-	-	-	
7	-	-	-	-	14
	0x01040002	-	-	-	
	0x03008000	-	-	-	
8	0x02020480	-	-	-	-
	0x0a00040e	-	0x06000c00	-	
	0x06010000	-	0x00010002	-	

Table 4: A 12-round differential trail for Gimli with probability 2^{-188} expanding the optimal 7-round differential trail.

Round	$s_{*,0}$	$s_{*,1}$	$s_{*,2}$	$s_{*,3}$	Weight
0	0x04010100	0x80010380	0x06010100	0x80100C00	46
	-	0x40010180	0x02000000	0x40100400	
	0x02008080	0x40010180	0x03018080	0x40104400	
1	-	0x80020080	-	0x80210180	24
	-	0x00060080	-	0x40200080	
	-	0x00070480	-	0x00318400	
2	-	0x00003100	-	0x80401180	20
	-	0x00000100	-	0x80000180	
	-	0x80000980	-	0x80000980	
3	-	-	-	0x80800100	8
	-	-	-	0x80400000	
	-	-	-	0x80400080	
4	-	-	-	0x80000000	0
	-	-	-	0x80000000	
	-	-	-	0x80000000	
5	-	-	-	-	0
	-	-	-	-	
	-	-	-	0x80000000	
6	-	-	-	0x00800000	2
	-	-	-	-	
	-	-	-	-	
7	-	-	-	-	4
	-	-	-	0x00000001	
	-	-	-	0x00800000	
8	-	-	-	0x01008000	6
	-	-	-	0x00000200	
	-	-	-	0x01000000	
9	-	-	0x00010002	-	14
	-	-	-	0x01040002	
	-	-	-	0x03008000	
10	-	-	-	0x020A0480	24
	-	-	0x02000400	0x0A000402	
	-	-	0x00010002	0x0A010000	
11	0x02020104	0x02000100	-	-	40
	-	-	0x00080004	0x14010430	
	-	-	0x00020004	0x1E081480	
12	-	-	0x00000A00	0xB00A0910	-
	0x04020804	0x00020004	0x10001800	0x02186078	
	0x02020104	0x02000100	0x00040008	0x3C102900	

# rounds	1	2	3	4	5	6	7	8	9	10	11
	1	2	4	8	15	27	48	88	153	254	367

In integral attacks, a part of the input is chosen as active bits and the other part is chosen as constant bits. Then, we have to evaluate the algebraic degree involving active bits. From the structure of the round function of Gimli, the algebraic degree will be small when 96 entire bits in each column are active. We evaluated two cases: the algebraic degree involving $s_{i,0}$ is evaluated in the first case, and the algebraic degree involving $s_{i,0}$ and $s_{i,1}$ is evaluated in the second case. Moreover, all z_0 in 4 columns are evaluated, and the following table summarizes the upper bound of the algebraic degree in the weakest column in every round.

# rounds		3	4	5	6	7	8	9	10	11	12	13	14
active columns	0	0	0	4	8	15	28	58	89	95	96	96	96
	0 and 1	0	0	7	15	30	47	97	153	190	191	191	192

The above result implies that Gimli has 11-round integral distinguisher when 96 bits in $s_{i,0}$ are active and the others are constant. Moreover, when 192 bits in $s_{i,0}$ and $s_{i,1}$ are active and the others are constant, Gimli has 13-round integral distinguisher.

7.4 Degree evaluation by division property

The division property is normally used to search for integral distinguishers. Evaluation of the algebraic degree, which we use in this paper, is kind of a reverse use of the division property. Assume that the MILP model $\mathcal{M}$ in which the propagation rules of the division property for Gimli are described, and $\vec{x}$ and $\vec{y}$ denote MILP variables corresponding to input and output of Gimli, respectively. In the normal use of the division property, $\vec{x}$ has a specific value. To be precise, $x_i = 1$ when the ith bit of the input is active, and $x_i = 0$ otherwise. Then, we check the feasibility that $\vec{y} = \vec{e_j}$, where $\vec{e_j}$ is 384-dimensional unit vector whose jth element is 1. If it is impossible then the jth bit is balanced.

In the reverse use, we constrain $\vec{y}$ and maximize $\sum_{i=1}^{384} x_i$ by MILP. For example, we constrain $\sum_{i=1}^{384} y_i = 1$ and maximize $\sum_{i=1}^{384} x_i$ by using MILP. Suppose the maximized value is d in r-round Gimli. Then, in other words, if $\sum_{i=1}^{384} x_i = d + 1$, it is impossible that $\sum_{i=1}^{384} y_i = 1$. From this it follows that the algebraic degree of r-round Gimli is at most d. If we focus on a specific bit in the output, e.g., the jth bit, we constrain $\vec{y} = \vec{e_j}$ and maximize $\sum_{i=1}^{384} x_i$ by using MILP. Moreover, if the algebraic degree involving active bits chosen by attackers is evaluated, we maximize $\sum_{i \in S} x_i$, where S is chosen by attackers. This strategy allows us to efficiently evaluate the algebraic degree in several scenarios.

7.5 Attacks against "hermetic" properties

After the initial Gimli paper, Hamburg posted a note called "Cryptanalysis of 22 1/2 rounds of Gimli". We issued the following statement in response.

To achieve high security with the best possible efficiency on a broad range of platforms, Gimli works locally with portions of its internal state for a few rounds before the portions communicate. As an analogy, a Feistel cipher achieves high security with just 4 communication steps between halves of the state when the computations on half-states are strong enough.

In a paper titled "Cryptanalysis of 22 1/2 rounds of Gimli", Hamburg claims to "show" that this is "dangerous", that Gimli's "slow diffusion is a serious weakness", etc.

However, Hamburg's specific claim is that his "attack" takes "$2^{138.5}$ work" and "2^{129} bits of memory". This is more hardware and more time than a naive brute-force attack against Hamburg's 192-bit key.

Specifically, a cluster of 2^{80} brute-force key-search units would cost billions of times less than Hamburg's "attack" and would find the key millions of times faster. Replacing the cipher with another 192-bit cipher, such as AES-192, would not stop this brute-force attack.

Applying the "golden collision" techniques of 1996 van Oorschot–Wiener would allow memory to be reduced in Hamburg's "attack", but at a huge cost in time, again making the attack more expensive than a naive brute-force attack against the 192-bit key.

Furthermore, there are actually many users with many keys. Brute force gains far more from this than Hamburg's "attack" does, making the actual gap even larger than the above analysis indicates.

Furthermore, Hamburg's "attack" is against an artificial, ad-hoc mode that we would not recommend, that we did not recommend, and that, as far as we know, has never appeared before. Most importantly, the standard practice established by Salsa20 and ChaCha20 is for stream-cipher ("PRF") users to add key words to positions selected to maximize diffusion for the underlying permutation, whereas Hamburg adds key words to positions selected to *minimize* diffusion.

Finally, Hamburg's "attack" will not be feasible in the foreseeable future, even with quantum computers. Even if the "attack" were extended to the full 24 rounds, it would not contradict any security claims made in the Gimli paper.

7.6 The Gimli modes

Gimli-Hash uses the well-known sponge mode [10] and profits from the extensive literature on sponges, in particular the results on its indifferentiability in [9].

Gimli-Cipher uses the well-known duplex mode [11]. AEAD modes using the duplex construction have also received considerable attention from the research community, and several security proofs have been provided for different parameter choices [1, 11, 15, 22].

We encourage analysis of reduced-round variants of Gimli-Hash and Gimli-Cipher. Since the round counter decreases from 24 through 1, the natural R-round generalization has the round counter decreasing from R through 1. However, we also encourage analysis of the initial rounds. We suggest the terminology "last 3 rounds" for round counters $3, 2, 1$, and "first 3 of 24 rounds" for round counters $24, 23, 22$.

Liu, Isobe, and Meier [24] have presented attacks on reduced-round versions of GIMLI-HASH: the last 4 rounds, the first 5 of 24 rounds, and an example of a collision for the first 3 of 24 rounds.

7.7 Release of unverified plaintext

The damage caused by release of unverified plaintext depends on the application and can be very large. To avoid this damage, each decryption device must buffer the plaintext that it is producing, until it receives and verifies the authenticator.

Consequently, if the smallest device used in an application can buffer only B bytes, then the application must keep all plaintexts below B bytes. To transmit longer messages, the application must split the messages into separately authenticated plaintexts.

To protect an application that incorrectly ignores error messages from decryption, our software clears the plaintext buffer when the authenticator is invalid. The pattern of valid authenticators might be secret in some applications; to prevent timing attacks in this scenario, our software takes the time to rewrite the plaintext buffer whether or not the authenticator is valid.

8 Advantages and limitations

8.1 Overview

What distinguishes GIMLI from other permutations is its *cross-platform* performance. GIMLI is designed for energy-efficient hardware *and* for side-channel-protected hardware *and* for microcontrollers *and* for compactness *and* for vectorization *and* for short messages *and* for a high security level.

The following subsections report the performance of GIMLI for several target platforms. See Tables 5 and 6 for cross-platform overviews of hardware and software performance.

8.2 Speed of permutations vs. speed of modes

We emphasize that the performance analysis below focuses primarily on permutation speed, the cycles for the GIMLI permutation *per byte of the state.* Both GIMLI-HASH and GIMLI-CIPHER use three times as many cycles per byte, since they use simple sponge and duplex modes, handling only 128 bits of input per 384-bit permutation call.

For example, on a 3.5GHz Intel Haswell CPU core, measuring our `gimli24v1/sse` software with gcc 5.4.0 and do-part from SUPERCOP shows 31544 cycles to hash 2048 bytes, 32460 cycles to encrypt 2048 bytes, and 33544 cycles to decrypt 2048 bytes. Using Gimli to instead build a block cipher in Even–Mansour mode, and top of this a stream cipher in counter mode, would process three times as many bytes per Gimli call, and would also allow multiple blocks to be handled in parallel on this CPU. There are also many fast choices of authenticators.

We recommend simple sponge and duplex modes for lightweight applications. Applications using other modes also benefit from the cross-platform performance of the Gimli permutation. It is natural to factor performance analysis into (1) analysis of permutation speed and (2) analysis of mode speed.

8.3 FPGA & ASIC

We designed and evaluated three main architectures to address different hardware applications. These different architectures are a tradeoff between resources, maximum operational frequency and number of cycles necessary to perform the full permutation. Even with these differences, all 3 architectures share a common simple communication interface which can be expanded to offer different operation modes. All this was done in VHDL and tested in ModelSim for behavioral results, synthesized and tested for FPGAs with Xilinx ISE 14.7. In case of ASICs this was done through Synopsis Ultra and Simple Compiler with 180nm UMC L180, and Encounter RTL Compiler with ST 28nm FDSOI technology.

The first architecture, depicted in Figure 4, performs a certain number of rounds in one clock cycle and stores the output in the same buffer as the input. The number of rounds it can perform in one cycle is chosen before the synthesis process and can be 1, 2, 3, 4, 6, or 8. In case of 12 or 24 combinational rounds, optimized architectures for these cases were done, in order to have better results. The rounds themselves are computed as shown in Figure 5. In every round there is one SP-box application on the whole state, followed by the linear layer. In the linear layer, the operation can be a small swap with round constant addition, a big swap, or no operation, which are chosen according to the two least significant bits of the round number. The round number starts from 24 and is decremented by one in each combinational round block.

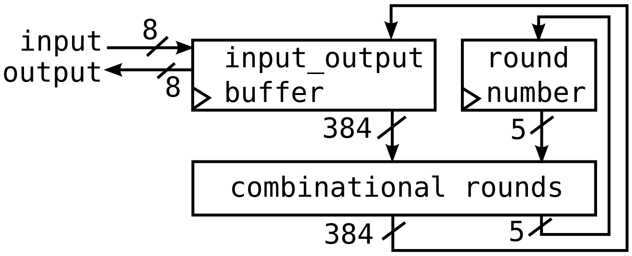


Figure 4: Round-based architecture

Besides the round and the optimized half and full combinational architectures, the other one is a serial-based architecture illustrated in Figure 6. The serial-based architecture performs one SP-box application per cycle, through a circular-shift-based architecture, therefore taking

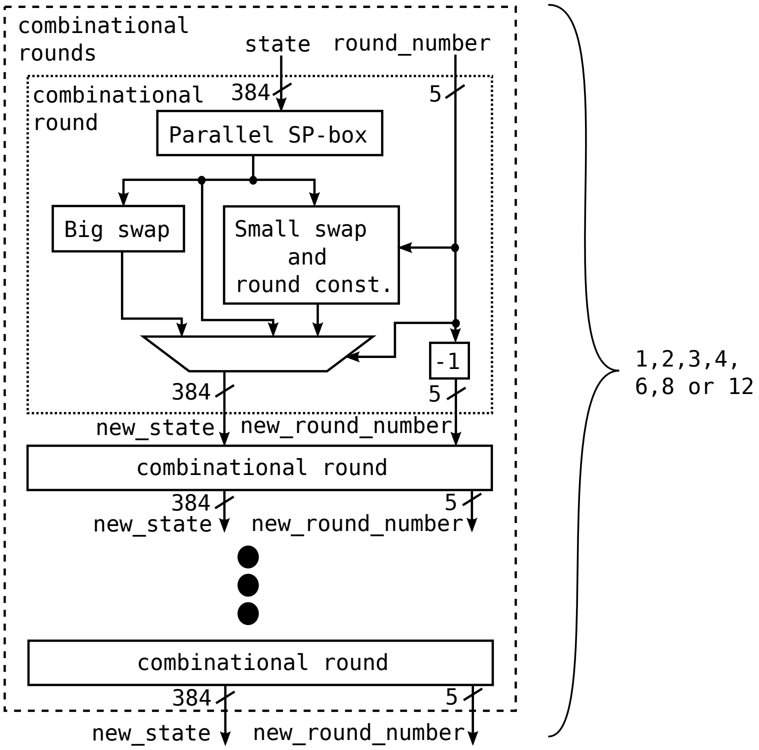


Figure 5: Combinational round in round-based architecture

in total 4 cycles. In case of the linear layer, it is still executed in one cycle in parallel. The reason of not being done in a serial based manner, is because the parallel version cost is very low.

All hardware results are shown in Table 5. In case of FPGAs the lowest latency is the one with 4 combinational rounds in one cycle, and the one with best Resources×Time/State is the one with 2 combinational rounds. For ASICs the results change as the lowest latency is the one with full combinational setting, and the one with best Resources×Time/State is the one with 8 combinational rounds for 180nm and 4 combinational rounds for 28nm. This difference illustrates that each technology can give different results, making it difficult to compare results on different technology.

Hardware variants that do 2 or 4 rounds in one cycle appear to be attractive choices, depending on the application scenario. The serial version needs 4.5 times more cycles than the 1-round version, while saving around 28% of the gate equivalents (GE) in the 28nm ASIC technology, and less in the other ASIC technology and FPGA. If resource constraints are extreme enough to justify the serial version then it would be useful to develop a new version optimized for the target technology, for better results.

To compare the Gimli permutation to other permutations in the literature, we synthesized all permutations with similar half-combinational architectures, taking exactly 2 cycles to perform a permutation. The permutations that were chosen for comparison were selected close to Gimli in terms of size, even though in the end the final metric was divided by the permutation size to try to "normalize" the results.

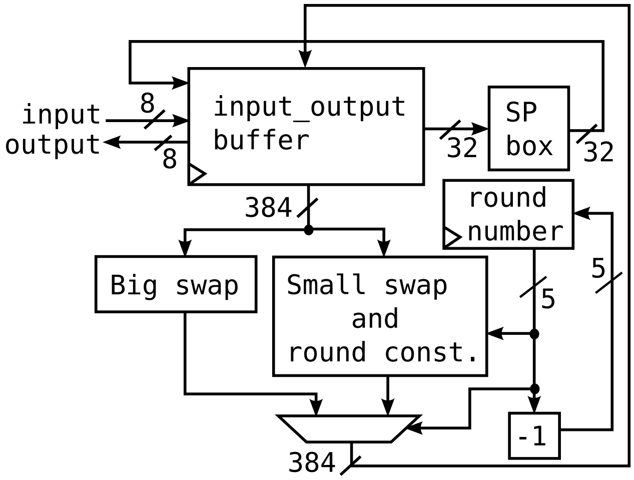


Figure 6: Serial-based architecture

The best results in Resources×Time/State are from 24-round GIMLI and 12-round Ascon-128, with Ascon slightly more efficient in the FPGA results and GIMLI more efficient in the ASIC results. Both permutations in all 3 technologies had very similar results, while Keccak-f[400] is worse in all 3 technologies. The permutations SPONGENT-256/256/128, Photon-256/32/32 and C-Quark have a much higher resource utilization in all technologies. This is because they were designed to work with little resources in exchange for a very high response time (e.g., SPONGENT is reported to use 2641 GE for 18720 cycles, or 5011 GE for 195 cycles), therefore changing the resource utilization from logic gates to time. GIMLI and Ascon are the most efficient in the sense of offering a similar security level to SPONGENT, Photon and C-Quark, with much lower product of time and logic resources.

8.4 SP-box in assembly

We now turn our attention to software. Subsequent subsections explain how to optimize GIMLI for various illustrative examples of CPUs. As a starting point, we note that one can efficiently apply the GIMLI SP-box to three 32-bit registers `x`, `y`, `z` using just two temporary registers `u`, `v`. The order of operations is shown in the original GIMLI paper and in various GIMLI implementations.

8.5 8-bit microcontroller: AVR ATmega

The AVR architecture provides 32 8-bit registers (256 bits). This does not allow the full 384-bit GIMLI state to stay in the registers: we are forced to use loads and stores in the main loop.

To minimize the overhead for loads and stores, we work on a half-state (two columns) for as long as possible. For example, we focus on the left half-state for rounds 21, 20, 19, 18, 17, 16, 15, 14. Before doing this, we focus on the right half-state through the end of round

Table 5: Hardware results for Gimli and competitors.
Gates Equivalent(GE). Slice(S). LUT(L). Flip-Flop(F).
* Could not finish the place and route.

Perm.	State size	Version	Cycles	Resources	Period (ns)	Time (ns)	Res.×Time/ State
FPGA – Xilinx Spartan 6 LX75							
Ascon	320		2	732 S(2700 L+325 F)	34.570	70	158.2
Gimli	**384**	**12**	**2**	**1224 S(4398 L+389 F)**	**27.597**	**56**	**175.9**
Keccak	400		2	1520 S(5555 L+405 F)	77.281	155	587.3
C-quark*	384		2	2630 S(9718 L+389 F)	98.680	198	1351.7
Photon	288		2	2774 S(9430 L+293 F)	74.587	150	1436.8
Spongent*	384		2	7763 S(19419 L+389 F)	292.160	585	11812.7
Gimli	**384**	**24**	**1**	**2395 S(8769 L+385 F)**	**56.496**	**57**	**352.4**
Gimli	**384**	**8**	**3**	**831 S(2924 L+390 F)**	**24.531**	**74**	**159.3**
Gimli	**384**	**6**	**4**	**646 S(2398 L+390 F)**	**18.669**	**75**	**125.6**
Gimli	**384**	**4**	**6**	**415 S(1486 L+391 F)**	**8.565**	**52**	**55.5**
Gimli	**384**	**3**	**8**	**428 S(1587 L+393 F)**	**10.908**	**88**	**97.3**
Gimli	**384**	**2**	**12**	**221 S(815 L+392 F)**	**5.569**	**67**	**38.5**
Gimli	**384**	**1**	**24**	**178 S(587 L+394 F)**	**4.941**	**119**	**55.0**
Gimli	**384**	**Serial**	**108**	**139 S(492 L+397 F)**	**3.996**	**432**	**156.2**
28nm ASIC – ST 28nm FDSOI technology							
Gimli	**384**	**12**	**2**	**35452GE**	**2.2672**	**5**	**418.6**
Ascon	320		2	32476GE	2.8457	6	577.6
Keccak	400		2	55683GE	5.6117	12	1562.4
C-quark	384		2	111852GE	9.9962	20	5823.4
Photon	288		2	296420GE	10.0000	20	20584.7
Spongent	384		2	1432047GE	12.0684	25	90013.1
Gimli	**384**	**24**	**1**	**66205GE**	**4.2870**	**5**	**739.1**
Gimli	**384**	**8**	**3**	**25224GE**	**1.5921**	**5**	**313.7**
Gimli	**384**	**6**	**4**	**21675GE**	**2.1315**	**9**	**481.2**
Gimli	**384**	**4**	**6**	**14999GE**	**1.0549**	**7**	**247.2**
Gimli	**384**	**3**	**8**	**14808GE**	**2.0119**	**17**	**620.6**
Gimli	**384**	**2**	**12**	**10398GE**	**1.0598**	**13**	**344.4**
Gimli	**384**	**1**	**24**	**8097GE**	**1.0642**	**26**	**538.5**
Gimli	**384**	**Serial**	**108**	**5843GE**	**1.5352**	**166**	**2522.7**
180nm ASIC – UMC L180							
Gimli	**384**	**12**	**2**	**26685**	**9.9500**	**20**	**1382.9**
Ascon	320		2	23381	11.4400	23	1671.7
Keccak	400		2	37102	22.4300	45	4161.0
C-quark	384		2	62190	37.2400	75	12062.1
Photon	288		2	163656	99.5900	200	113183.8
Spongent	384		2	234556	99.9900	200	122151.9
Gimli	**384**	**24**	**1**	**53686**	**17.4500**	**18**	**2439.6**
Gimli	**384**	**8**	**3**	**19393**	**7.9100**	**24**	**1198.4**
Gimli	**384**	**6**	**4**	**15886**	**12.5100**	**51**	**2070.0**
Gimli	**384**	**4**	**6**	**11008**	**10.1700**	**62**	**1749.1**
Gimli	**384**	**3**	**8**	**10106**	**10.0500**	**81**	**2115.8**
Gimli	**384**	**2**	**12**	**7112**	**15.2000**	**183**	**3377.8**
Gimli	**384**	**1**	**24**	**5314**	**9.5200**	**229**	**3161.4**
Gimli	**384**	**Serial**	**108**	**3846**	**11.2300**	**1213**	**12146.0**

18, so that the *Big-Swap* at the end of round 18 can feed 2 words (64 bits) from the right half-state into the left half-state. See Figure 7 for the exact order of computation.

A half-state requires a total of 24 registers (6 words), leaving us with 8 registers (2 words) to use as temporaries. We can therefore use the same order of operations as in Section 8.4 for each SP-box. In a stretch of 8 rounds on a half-state (16 SP-boxes) there are just a few loads and stores.

We provide two implementations of this construction. One is fully unrolled and optimized for speed: it runs in just 10 264 cycles, using 19 218 bytes of ROM. The other is optimized for size: it uses just 778 bytes of ROM and runs in 23 670 cycles. Each implementation requires about the same amount of stack, namely 45 bytes.

8.6 32-bit low-end embedded microcontroller: ARM Cortex-M0

ARM Cortex-M0 comes with 14 32-bit registers. However `orr`, `eor`, `and`-like instructions can only be used on the lower registers (`r0` to `r7`). This forces us to use the same computation layout as in the AVR implementation. We split the state into two halves: one in the lower registers, one in the higher ones. Then we can operate on each during multiple rounds before exchanging them.

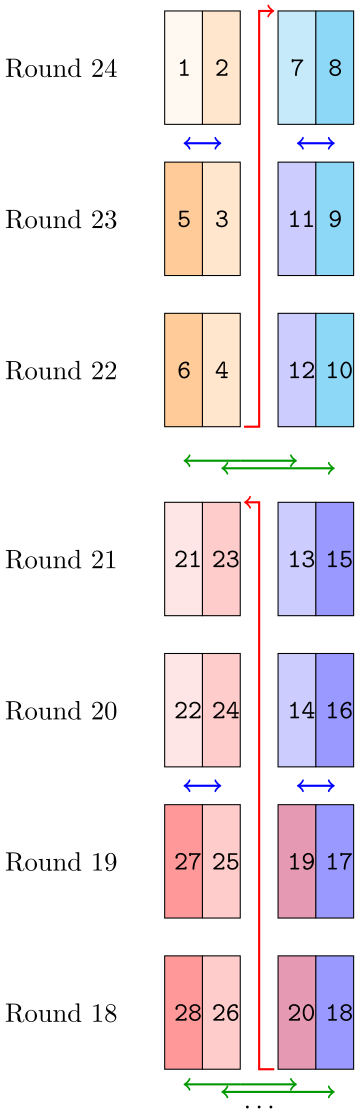
8.7 32-bit high-end embedded microcontroller: ARM Cortex-M3

We focus here on the ARM Cortex-M3 microprocessor, which implements the ARMv7-M architecture. There is a higher-end microcontroller, the Cortex-M4, implementing the ARMv7E-M architecture; but our Gimli software does not make use of any of the DSP, (optional) floating-point, or additional saturated instructions added in this architecture.

The Cortex-M3 features 16 32-bit registers `r0` to `r15`, with one register used as program counter and one as stack pointer, leaving 14 registers for free use. As the Gimli state fits into 12 registers and we need only 2 registers for temporary values, we compute the Gimli permutation without requiring any load or store instructions beyond the initial loads of the input and the final stores of the output.

One particularly interesting feature of various ARM instruction sets including the ARMv7-M instruction set are free shifts and rotates as part of arithmetic instructions. More specifically, all bit-logical operations allow one of the inputs to be shifted or rotated by an arbitrary fixed distance for free. This was used, e.g., in [30, Sec. 3.1] to eliminate all rotation instructions in an unrolled implementation of BLAKE. For Gimli this feature gives us the non-cyclic shifts by 1, 2, 3 and the rotation by 9 for free. We have not found a way to eliminate the rotation by 24. Each SP-box evaluation thus uses 10 instructions: namely, 9 bit-logical operations (6 xors, 2 ands, and 1 or) and one rotation.

From these considerations we can derive a lower bound on the amount of cycles required for the Gimli permutation: Each round performs 4 SP-box evaluations (one on each of the



```
1.  SP-box col.  0
2.  SP-box col.  1
swap word $s_{0,0}$ and $s_{0,1}$
3.  SP-box col.  1
4.  SP-box col.  1
5.  SP-box col.  0
6.  SP-box col.  0
store columns 0,1 ; load columns 2,3
7.  SP-box col.  2
8.  SP-box col.  3
swap word $s_{0,2}$ and $s_{0,3}$
9.  SP-box col.  3
10.  SP-box col.  3
11.  SP-box col.  2
12.  SP-box col.  2
push word $s_{0,2}$, $s_{0,3}$ ; load word $s_{0,0}$, $s_{0,1}$
13.  SP-box col.  2
14.  SP-box col.  2
15.  SP-box col.  3
16.  SP-box col.  3
swap word $s_{0,2}$ and $s_{0,3}$
17.  SP-box col.  3
18.  SP-box col.  3
19.  SP-box col.  2
20.  SP-box col.  2
store columns 2,3 ; load columns 0,1
pop word $s_{0,0}$, $s_{0,1}$
21.  SP-box col.  0
22.  SP-box col.  0
23.  SP-box col.  1
24.  SP-box col.  1
swap word $s_{0,0}$ and $s_{0,1}$
25.  SP-box col.  1
26.  SP-box col.  1
27.  SP-box col.  0
28.  SP-box col.  0
push word $s_{0,0}$, $s_{0,1}$ ; load word $s_{0,2}$, $s_{0,3}$
...
```

Figure 7: computation order on AVR

columns of the state), each using 10 instructions, for a total of 40 instructions. In 24 rounds we thus end up with $24 \cdot 40 = 960$ instructions from the SP-boxes, plus 6 xors for the addition of round constants. This gives us a lower bound of 966 cycles for the Gimli permutation, assuming an unrolled implementation in which all *Big-Swap* and *Small-Swap* operations are handled through (free) renaming of registers. Our implementation for the M3 uses such a fully unrolled approach and takes 1 047 cycles.

8.8 32-bit smartphone CPU: ARM Cortex-A8 with NEON

We focus on a Cortex-A8 for comparability with the highly optimized Salsa20 results of [8]. As a future optimization target we suggest a newer Cortex-A7 CPU core, which according to ARM has appeared in more than a billion chips. Since our Gimli software uses almost purely vector instructions (unlike [8], which mixes integer instructions with vector instructions), we expect it to perform similarly on the Cortex-A7 and the Cortex-A8.

The Gimli state fits naturally into three 128-bit NEON vector registers, one row per vector. The T-function inside the Gimli SP-box is an obvious match for the NEON vector instructions: two ANDs, one OR, four shifts, and six XORs. The rotation by 9 uses three vector instructions. The rotation by 24 uses two 64-bit vector instructions, namely permutations of byte positions (`vtbl`) using a precomputed 8-byte permutation. The four SP-boxes in a round use 18 vector instructions overall.

A straightforward 4-round-unrolled assembly implementation uses just 77 instructions for the main loop: 72 for the SP-boxes, 1 (`vrev64.i32`) for *Small-Swap*, 1 to load the round constant from a precomputed 96-byte table, 1 to xor the round constant, and 2 for loop control (which would be reduced by further unrolling). We handle *Big-Swap* implicitly through the choice of registers in two `vtbl` instructions, rather than using an extra `vswp` instruction. Outside the main loop we use just 9 instructions, plus 3 instructions to collect timing information and 20 bytes of alignment, for 480 bytes of code overall.

The lower bound for arithmetic is $65 \cdot 6 = 390$ cycles: 16 arithmetic cycles for each of the 24 rounds, and 6 extra for the round constants. The Cortex-A8 can overlap permutations with arithmetic. With moderate instruction-scheduling effort we achieved 419 cycles, just 8.73 cycles/byte. For comparison, [8] says that a “straightforward NEON implementation” of the inner loop of Salsa20 “cannot do better than 11.25 cycles/byte” (720 cycles for 64 bytes), plus approximately 1 cycle/byte overhead. [8] does better than this only by handling multiple blocks in parallel: 880 cycles for 192 bytes, plus the same overhead.

8.9 64-bit server CPU: Intel Haswell

Intel’s server/desktop/laptop CPUs have had 128-bit vectorized integer instructions (“SSE2”) starting with the Pentium 4 in 2001, and 256-bit vectorized integer instructions (“AVX2”) starting with the Haswell in 2013. In each case the vector registers appeared in CPUs a few years earlier supporting vectorized floating-point instructions (“SSE” and

"AVX"), including full-width bitwise logic operations, but not including shifts. The vectorized integer instructions include shifts but not rotations. Intel has experimented with 512-bit vector instructions in coprocessors such as Knights Corner and Knights Landing, and has announced a 512-bit instruction set that includes vectorized rotations and three-input logical operations, but we focus here on CPUs that are commonly available from Intel and AMD today.

Our implementation strategy for these CPUs is similar to our implementation strategy for NEON: again the state fits naturally into three 128-bit vector registers, with Gimli instructions easily translating into the CPU's vector instructions. The cycle counts on Haswell are better than the cycle counts for the Cortex-A8 since each Haswell core has multiple vector units. We save another factor of almost 2 for 2-way-parallel modes, since 2 parallel copies of the state fit naturally into three 256-bit vector registers. As with the Cortex-A8, we outperform Salsa20 and ChaCha20 for short messages.

References

[1] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 364–384. Springer, 2015.

[2] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Analysis of NORX: investigating differential and rotational properties. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 306–324. Springer, 2014. https://eprint.iacr.org/2014/317.pdf.

[3] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX: parallel and scalable AEAD. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2014.

[4] Jean-Philippe Aumasson, Simon Knellwolf, and Willi Meier. Heavy Quark for secure AEAD. In *DIAC 2012: Directions in Authenticated Ciphers*, 2012. https://131002.net/data/papers/AKM12.pdf.

[5] Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *The Hash Function BLAKE.* Information Security and Cryptography. Springer, 2014.

[6] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact

Table 6: Cross-platform software performance comparison of various permutations. “Hashing 500 bytes”: AVR cycles for comparability with [6]. “Permutation”: Cycles/byte for permutation on all platforms. AEAD timings from [7] are scaled to estimate permutaton timings.

Hashing 500 bytes	**Cycles**	**ROM Bytes**	**RAM Bytes**
AVR ATmega			
Spongent [6]	25 464 000	364	101
Keccak-f[400] [6]	1 313 000	608	96
Gimli-Hash [h] **(this submission)** small	805 110	778	44
Gimli-Hash [h] **(this submission)** fast	362 712	19 218	45

Permutation	**Cycles/B**	**ROM Bytes**	**RAM Bytes**
AVR ATmega			
Gimli **(this submission)** small	413	778	44
ChaCha20 [34]	238	– [b]	132
Salsa20 [21]	216	1 750	266
Gimli **(this submission)** fast	213	19 218	45
AES-128 [26] small	171	1 570	– [b]
AES-128 [26] fast	155	3 098	– [b]
ARM Cortex-M0			
Gimli **(this submission)**	49	4 730	64
ChaCha20 [27]	40	– [b]	– [b]
Chaskey [25]	17	414	– [b]
ARM Cortex-M3/M4			
Spongent [13, 28] (c-ref, our measurement)	129 486	1 180	– [b]
Ascon [17] (opt32, our measurement)	196	– [b]	– [b]
Keccak-f[400] [33]	106	540	– [b]
AES-128 [29]	34	3 216	72
Gimli **(this submission)**	21	3 972	44
ChaCha20 [20]	13	2 868	8
Chaskey [25]	7	908	– [b]
ARM Cortex-A8			
Keccak-f[400] (KetjeSR) [7]	37.52	– [b]	– [b]
Ascon [7]	25.54	– [b]	– [b]
AES-128 [7] many blocks	19.25	– [b]	– [b]
Gimli **(this submission)** single block	8.73	480	– [b]
ChaCha20 [7] multiple blocks	6.25	– [b]	– [b]
Salsa20 [7] multiple blocks	5.48	– [b]	– [b]
Intel Haswell			
Gimli **(this submission)** single block	4.46	252	– [b]
NORX-32-4-1 [7] single block	2.84	– [b]	– [b]
Gimli **(this submission)** two blocks	2.33	724	– [b]
Gimli **(this submission)** four blocks	1.77	1227	– [b]
Salsa20 [7] eight blocks	1.38	– [b]	– [b]
ChaCha20 [7] eight blocks	1.20	– [b]	– [b]
AES-128 [7] many blocks	0.85	– [b]	– [b]

[b] no data

[h] Sponge construction[10] with $c = 256$ bits, $r = 128$ bits and 256 bits of output.

implementation and performance evaluation of hash functions in ATtiny devices. Cryptology ePrint Archive: Report 2012/507, 2012. https://eprint.iacr.org/2012/507/.

[7] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to (accessed 2017-06-25).

[8] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012. https://cryptojedi.org/papers/#neoncrypto.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, January 2011. http://sponge.noekeon.org/CSF-0.1.pdf.

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[12] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. hacspec: Towards verifiable crypto standards. In Cas Cremers and Anja Lehmann, editors, *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*, volume 11322 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018. https://github.com/HACS-workshop/hacspec/blob/master/doc/hacspec-ssr18-paper.pdf.

[13] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing, 2011. https://eprint.iacr.org/2011/697.

[14] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):218–241, 2018.

[15] Joan Daemen, Bart Mennink, and Gilles Van Assche. Full-state keyed duplex with built-in multi-user support. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 606–637. Springer, 2017.

[16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for ARX with provable bounds: SPARX and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, volume 10031 of *LNCS*, pages 484–513. Springer, 2016. https://eprint.iacr.org/2016/984.pdf.

[17] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/asconv12.pdf, 2016.

[18] Pierre-Alain Fouque, Antoine Joux, and Chrysanthi Mavromati. Multi-user collisions: Applications to discrete logarithm, Even-Mansour and PRINCE. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 420–438. Springer, 2014. https://eprint.iacr.org/2013/761.pdf.

[19] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[20] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS – computing a 41KB signature in 16KB of RAM. In Giuseppe Persiano and Bo-Yin Yang, editors, *Public Key Cryptography – PKC 2016*, volume 9614 of *LNCS*, pages 446–470. Springer, 2016. Document ID: c7ea17f606835ab4368235a464e1f9f6, https://cryptojedi.org/papers/#armedsphincs.

[21] Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR microcontrollers. In Amr Youssef and Abderrahmane Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 156–172. Springer, 2013. https://cryptojedi.org/papers/#avrnacl.

[22] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond 2 c/2 security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 2014.

[23] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the SIMON block cipher family. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 161–185. Springer, 2015. https://eprint.iacr.org/2015/145.pdf.

[24] Fukang Liu, Takanori Isobe, and Willi Meier. Preimages and collisions for up to 5-round Gimli-Hash using divide-and-conquer methods, 2019. https://eprint.iacr.org/2019/1080.

[25] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. volume 8781 of *LNCS*, pages 306–323. Springer, 2014.

[26] B. Poettering. AVRAES: The AES block cipher on AVR controllers, 2003. http://point-at-infinity.org/avraes/.

[27] Niels Samwel and Moritz Neikes. arm-chacha20, 2016. https://gitlab.science.ru.nl/mneikes/arm-chacha20/tree/master.

[28] Erik Schneider and Wouter de Groot. spongent-avr, 2015. https://github.com/weedegee/spongent-avr.

[29] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptology – SAC 2016*, LNCS. Springer, to appear. Document ID: 9fc0b970660e40c264e50ca389dacd49, https://cryptojedi.org/papers/#aesarm.

[30] Peter Schwabe, Bo-Yin Yang, and Shang-Yi Yang. SHA-3 on ARM11 processors. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *Progress in Cryptology – AFRICACRYPT 2012*, volume 7374 of *LNCS*, pages 324–341. Springer, 2012. https://cryptojedi.org/papers/#sha3arm.

[31] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 287–314. Springer, 2015. https://eprint.iacr.org/2015/090.pdf.

[32] Yosuke Todo and Masakatu Morii. Bit-based division property and application to Simon family. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016*, volume 9783 of *LNCS*, pages 357–377. Springer, 2016. https://eprint.iacr.org/2016/285.pdf.

[33] Gilles Van Assche and Ronny Van Keer. Structuring and optimizing Keccak software. 2016. http://ccccspeed.win.tue.nl/papers/KeccakSoftware.pdf.

[34] Rhys Weatherley. Arduinolibs, 2016. https://rweather.github.io/arduinolibs/crypto.html.

[35] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, volume 10031 of *LNCS*, pages 648–678. Springer, 2016. https://eprint.iacr.org/2016/857.

A Statements

"Due to the specific requirements of the intellectual property statements as specified in Section 2.4, e-mail submissions shall not be accepted for these statements. The statements specified in Section 2.4 must be mailed to Dr. Kerry McKay, Information Technology Laboratory, Attention: Lightweight Cryptographic Algorithm Submissions, 100 Bureau Drive – Stop 8930, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930."

First blank in submitter statement: full name. Second blank: full postal address. Third, fourth, and fifth blanks: name of cryptosystem. Sixth and seventh blanks: describe and enumerate or state "none" if applicable.

First blank in patent statement: full name. Second blank: full postal address. Third blank: enumerate. Fourth blank: name of cryptosystem.

First blank in implementor statement: full name. Second blank: full postal address. Third blank: full name of the owner.

A.1 Statement by Each Submitter

I, ____________________________, of ____, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as ____________________________, is my own original work, or if submitted jointly with others, is the original work of the joint submitters.

I further declare that (check at least one of the following):

- □ I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as ____________________________;
- □ to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as ___________________, may be covered by the following U.S. and/or foreign patents: ____
- □ I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations: ____

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment. I do hereby agree to provide the statements required by Sections 2.4.2 and 2.4.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the lightweight crypto evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.4.1, 2.4.2 and 2.4.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

A.2 Statement by Patent (and Patent Application) Owner(s)

If there are any patents (or patent applications) identified by the submitter, including those held by the submitter, the following statement must be signed by each and every owner, or each owner's authorized representative, of each patent and patent application identified.

I, ____________________________*, of* ____*, am the owner or authorized representative of the owner (print full name, if different than the signer) of the following patent(s) and/or patent application(s):* __________________________________
____
and do hereby commit and agree to grant to any interested party on a worldwide basis, if the cryptosystem known as __________________ *is selected for standardization, in consideration of its evaluation and selection by NIST, a non-exclusive license for the purpose of implementing the standard (check one):*

- □ *without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination, OR*
- □ *under reasonable terms and conditions that are demonstrably free of any unfair discrimination.*

I further do hereby commit and agree to license such party on the same basis with respect to any other patent application or patent hereafter granted to me, or owned or controlled by me, that is or may be necessary for the purpose of implementing the standard.

I further do hereby commit and agree that I will include, in any documents transferring ownership of each patent and patent application, provisions to ensure that the commitments and assurances made by me are binding on the transferee and any future transferee.

I further do hereby commit and agree that these commitments and assurances are intended by me to be binding on successors-in-interest of each patent and patent application, regardless of whether such provisions are included in the relevant transfer documents.

I further do hereby grant to the U.S. Government, during the public review and the evaluation process, and during the lifetime of the standard, a nonexclusive, nontransferrable, irrevocable, paid-up worldwide license solely for the purpose of modifying my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability) for incorporation into the standard.

Signed:

Title:

Date:

Place:

A.3 Statement by Reference/Optimized/Additional Implementations' Owner(s)

The following must also be included:

I, ________________________, ____, am the owner or authorized representative of the owner ________________________ of the submitted reference implementation, optimized and additional implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the lightweight cryptography public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place: