GIMLI: a cross-platform permutation

Daniel J. Bernstein[1], Stefan Kölbl[2], Stefan Lucks[3], Pedro Maat Costa Massolino[4], Florian Mendel[5], Kashif Nawaz[6], Tobias Schneider[7], Peter Schwabe[4], François-Xavier Standaert[6], Yosuke Todo[8], and Benoît Viguier[4] ⋆ Date: June 27, 2017

[1] University of Illinois at Chicago
djb@cr.yp.to
[2] Technical University of Denmark
stek@dtu.dk
[3] Bauhaus-Universität Weimar
Stefan.Lucks@uni-weimar.de
[4] Radboud University
P.Massolino@cs.ru.nl,peter@cryptojedi.org,benoit@viguier.nl
[5] Graz University of Technology
florian.mendel@gmail.com
[6] Université Catholique de Louvain
kashif.nawaz@uclouvain.be,fstandae@uclouvain.be
[7] Ruhr-University Bochum
tobias.schneider-a7a@rub.de
[8] NTT Secure Platform Laboratories
todo.yosuke@lab.ntt.co.jp

Abstract. This paper presents GIMLI, a 384-bit permutation designed to achieve high security with high performance across a broad range of platforms, including 64-bit Intel/AMD server CPUs, 64-bit and 32-bit ARM smartphone CPUs, 32-bit ARM microcontrollers, 8-bit AVR microcontrollers, FPGAs, ASICs without side-channel protection, and ASICs with side-channel protection.

Keywords: Intel, AMD, ARM Cortex-A, ARM Cortex-M, AVR, FPGA, ASIC, side channels, the eyes of a hawk and the ears of a fox

⋆ Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work resulted from the Lorentz Center Workshop "HighLight: High-security lightweight cryptography". This work was supported in part by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO) and project number 645421 (ECRYPT-CSA); the Austrian Science Fund (FWF) under grant P26494-N15; the ARC project NANOSEC; the Belgian Fund for Scientific Research (FNRS-F.R.S.); the Technology Foundation STW (project 13499 TYPHOON), from the Dutch government; the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005; and the U.S. National Science Foundation under grant 1314919. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation." Permanent ID of this document: 93eb34af666d7fa7264d94c21c18034a.

1 Introduction

Keccak [11], the 1600-bit permutation inside SHA-3, is well known to be extremely energy-efficient: specifically, it achieves very high throughput in moderate-area hardware. Keccak is also well known to be easy to protect against side-channel attacks: each of its 24 rounds has algebraic degree only 2, allowing low-cost masking. The reason that Keccak is well known for these features is that *most symmetric primitives are much worse in these metrics*.

Chaskey [21], a 128-bit-permutation-based message-authentication code with a 128-bit key, is well known to be very fast on 32-bit embedded microcontrollers: for example, it runs at just 7.0 cycles/byte on an ARM Cortex-M3 microcontroller. The reason that Chaskey is well known for this microcontroller performance is that *most symmetric primitives are much worse in this metric*.

Salsa20 [7], a 512-bit-permutation-based stream cipher, is well known to be very fast on CPUs with vector units. For example, [9] shows that Salsa20 runs at 5.47 cycles/byte using the 128-bit NEON vector unit on a classic ARM Cortex-A8 (iPad 1, iPhone 4) CPU core. The reason that Salsa20 and its variant ChaCha20 [6] are well known for this performance is again that *most symmetric primitives are much worse in this metric*. This is also why ChaCha20 is now used by smartphones for HTTPS connections to Google [13] and Cloudflare [27].

Cryptography appears in a wide range of application environments, and each new environment seems to provide more reasons to be dissatisfied with most symmetric primitives. For example, Keccak, Salsa20, and ChaCha20 slow down dramatically when messages are short. As another example, Chaskey has a limited security level, and slows down dramatically when the same permutation is used inside a mode aiming for a higher security level.

Contributions of this paper. We introduce Gimli, a 384-bit permutation. Like other permutations with sufficiently large state sizes, Gimli can easily be used to build high-security block ciphers, tweakable block ciphers, stream ciphers, message-authentication codes, authenticated ciphers, hash functions, etc.

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.

We present a complete specification of Gimli (Section 2), a detailed design rationale (Section 3), an in-depth security analysis (Section 4), and performance results for a wide range of platforms (Section 5).

Availability of implementations. We place all software and hardware implementations described in this paper into the public domain to maximize reusability of our results. They are available at `https://gimli.cr.yp.to`.

2 Gimli specification

This section defines Gimli. See Section 3 for motivation.

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.

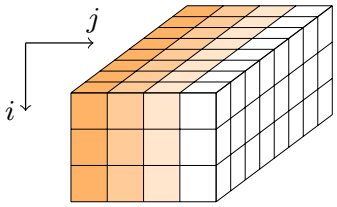


Fig. 1: State Representation

The state. Gimli 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.

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.

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.

The round constants. There are 24 rounds in Gimli, numbered $24, 23, \ldots, 1$. When the round number r is $24, 20, 16, 12, 8, 4$ we XOR the round constant $\texttt{0x9e377900} \oplus r$ to the first state word $s_{0,0}$.

Putting it together. Algorithm 1 is pseudocode for the full Gimli permutation. Appendix A is a C reference implementation.

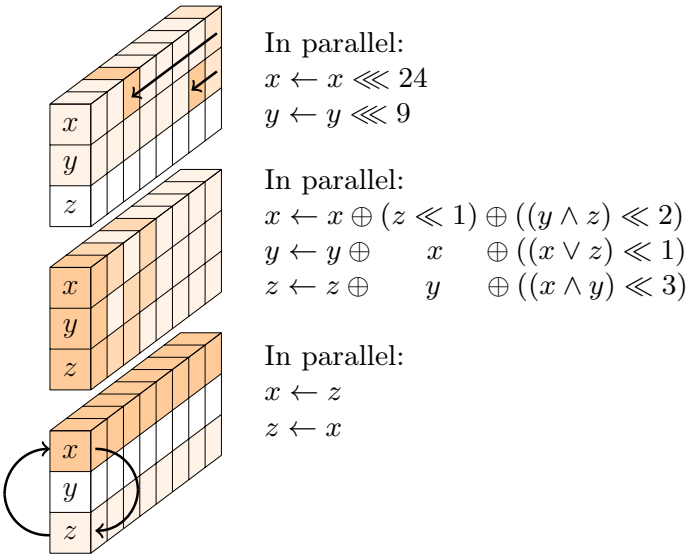


Fig. 2: The SP-box applied to a column

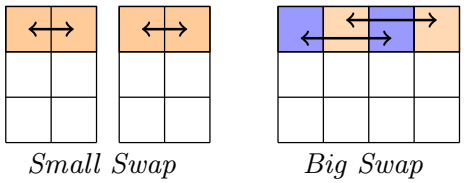


Fig. 3: The linear layer

3 Understanding the Gimli design

This section explains how we arrived at the Gimli design presented in Section 2.

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 [15], 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.

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

Algorithm 1 The GIMLI permutation

Require: $\mathbf{s} = (s_{i,j}) \in \mathcal{W}^{3\times 4}$
Ensure: 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 $r \mod 4 = 0$ **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 $r \mod 4 = 2$ **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 $r \mod 4 = 0$ **then**
      $s_{0,0} = s_{0,0} \oplus \texttt{0x9e377900} \oplus r$ ▷ Add constant
    end if
  end for
  return $(s_{i,j})$

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.

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 [2] 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.

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.

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 ac-

ceptable 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 [16]) 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 [3], 384-bit SPONGENT-256/256/128 [12], 320-bit Ascon, and 288-bit Photon-256/32/32 [17].

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.

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 [4] 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.

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 $\texttt{0x93e77900} \oplus r$. The implementor can also use $\texttt{0x93e77900} + r$ 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 [14]), 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.

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 Security analysis

4.1 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 Table 5 in Appendix F).

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

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

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

4.2 Differential Cryptanalysis

To study Gimli's resistance against differential cryptanalysis we use the same method as has been used for NORX [1] and Simon [20] 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 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*

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

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

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

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 Appendix G. 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 [1,20].

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 6 in Appendix G.

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 7 in Appendix G). If we look at the corresponding differential, this means we do not care about any intermediate differences; many trails might contribute 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 $\approx 2^{-158.63}$.

4.3 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 [28,29] to evaluate the algebraic degree, and the propagation search is assisted by mixed integer linear programming (MILP) [32]. See Appendix H.

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.

# 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.

5 Implementations

This section reports the performance of Gimli for several target platforms. See Tables 3 and 4 for cross-platform overviews of hardware and software performance.

5.1 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.

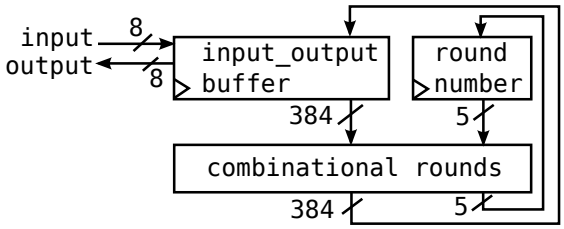


Fig. 4: Round-based architecture

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 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 3. 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

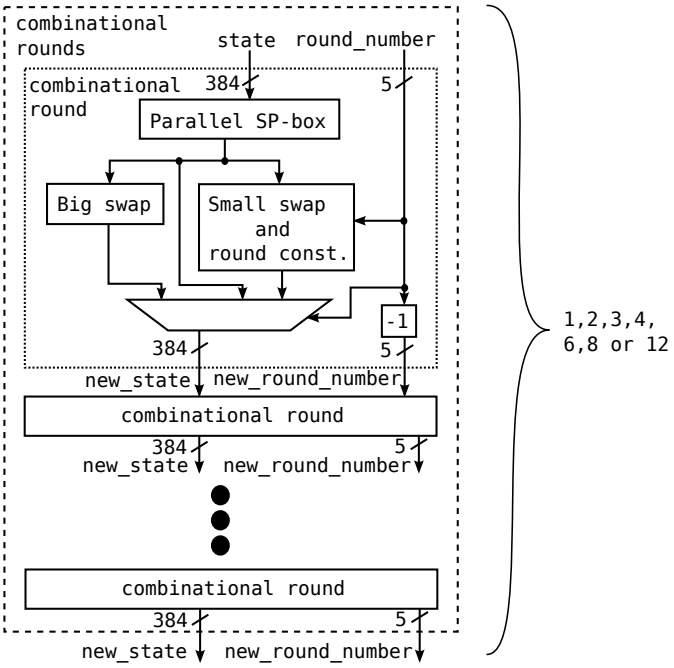


Fig. 5: Combinational round in round-based architecture

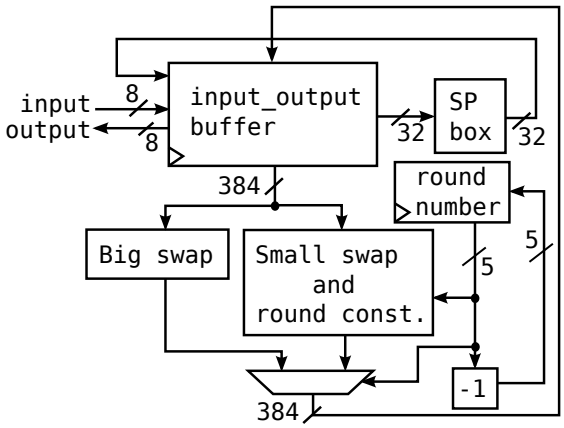


Fig. 6: Serial-based architecture

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.

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 permutation 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.

5.2 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 show in Listing 5.2 how to apply the Gimli SP-box to three 32-bit registers `x`, `y`, `z` using just two temporary registers `u`, `v`.

```
# Rotate
x ← x ⋘ 24
y ← y ⋘ 9
u ← x

# Compute x
v ← z ≪ 1
x ← y ∧ z
x ← x ≪ 2
x ← x ⊕ v
x ← x ⊕ u

# Compute y
v ← y
y ← u ∨ z
y ← y ≪ 1
y ← y ⊕ u
y ← y ⊕ v

# Compute z
u ← u ∧ v
u ← u ≪ 3
v ← v ⊕ u
z ← v ⊕ z
```

Listing 5.2: SP-box assembly instructions

5.3 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 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 Appendix C 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

Table 3: 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**

of operations as defined in Listing 5.2 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.

5.4 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.

5.5 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 [26, 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 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.

5.6 32-bit smartphone CPU: ARM Cortex-A8 with NEON

We focus on a Cortex-A8 for comparability with the highly optimized Salsa20 results of [9]. 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 [9], 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, [9] 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. [9] does better than this only by handling multiple blocks in parallel: 880 cycles for 192 bytes, plus the same overhead.

5.7 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 co-processors 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. 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. 10, 11
2. 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. 6
3. 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. 7
4. Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, and Luca Henzen. *The Hash Function BLAKE*. Information Security and Cryptography. Springer, 2014. 8
5. 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 implementation and performance evaluation of hash functions in ATtiny devices. Cryptology ePrint Archive: Report 2012/507, 2012. https://eprint.iacr.org/2012/507/. 20
6. Daniel J. Bernstein. ChaCha, a variant of Salsa20. SASC 2008: The State of the Art of Stream Ciphers, 2008. https://cr.yp.to/chacha/chacha-20080128.pdf. 2
7. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew J. B. Robshaw and Olivier Billet, editors, *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008. https://cr.yp.to/snuffle/salsafamily-20071225.pdf. 2
8. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to (accessed 2017-06-25). 20
9. 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. 2, 17, 18
10. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, January 2011. http://sponge.noekeon.org/CSF-0.1.pdf. 20
11. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, 2013. http://keccak.noekeon.org/Keccak-slides-at-Eurocrypt-May2013.pdf. 2

Table 4: Cross-platform software performance comparison of various permutations. "Hashing 500 bytes": AVR cycles for comparability with [5]. "Permutation": Cycles/byte for permutation on all platforms. AEAD timings from [8] are scaled to estimate permutaton timings.

Hashing 500 bytes	**Cycles**	**ROM Bytes**	**RAM Bytes**
AVR ATmega			
Spongent [5]	25 464 000	364	101
Keccak-f[400] [5]	1 313 000	608	96
Gimli-Hash[h] **(this paper)** small	805 110	778	44
Gimli-Hash[h] **(this paper)** fast	362 712	19 218	45

Permutation	**Cycles/B**	**ROM Bytes**	**RAM Bytes**
AVR ATmega			
Gimli **(this paper)** small	413	778	44
ChaCha20 [31]	238	–[b]	132
Salsa20 [19]	216	1 750	266
Gimli **(this paper)** fast	213	19 218	45
AES-128 [22] small	171	1 570	–[b]
AES-128 [22] fast	155	3 098	–[b]
ARM Cortex-M0			
Gimli **(this paper)**	49	4 730	64
ChaCha20 [23]	40	–[b]	–[b]
Chaskey [21]	17	414	–[b]
ARM Cortex-M3/M4			
Spongent [12,24] (c-ref, our measurement)	129 486	1 180	–[b]
Ascon [15] (opt32, our measurement)	196	–[b]	–[b]
Keccak-f[400] [30]	106	540	–[b]
AES-128 [25]	34	3 216	72
Gimli **(this paper)**	21	3 972	44
ChaCha20 [18]	13	2 868	8
Chaskey [21]	7	908	–[b]
ARM Cortex-A8			
Keccak-f[400] (KetjeSR) [8]	37.52	–[b]	–[b]
Ascon [8]	25.54	–[b]	–[b]
AES-128 [8] many blocks	19.25	–[b]	–[b]
Gimli **(this paper)** single block	8.73	480	–[b]
ChaCha20 [8] multiple blocks	6.25	–[b]	–[b]
Salsa20 [8] multiple blocks	5.48	–[b]	–[b]
Intel Haswell			
Gimli **(this paper)** single block	4.46	252	–[b]
NORX-32-4-1 [8] single block	2.84	–[b]	–[b]
Gimli **(this paper)** two blocks	2.33	724	–[b]
Gimli **(this paper)** four blocks	1.77	1227	–[b]
Salsa20 [8] eight blocks	1.38	–[b]	–[b]
ChaCha20 [8] eight blocks	1.20	–[b]	–[b]
AES-128 [8] 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.

12. 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. 7, 20
13. Elie Bursztein. Speeding up and strengthening HTTPS connections for Chrome on Android, 2014. https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html. 2
14. 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. 8
15. 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. 4, 20
16. 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. 7
17. 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. 7
18. 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. 20
19. 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. 20
20. 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. 10, 11
21. 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. 2, 20
22. B. Poettering. AVRAES: The AES block cipher on AVR controllers, 2003. http://point-at-infinity.org/avraes/. 20
23. Niels Samwel and Moritz Neikes. arm-chacha20, 2016. https://gitlab.science.ru.nl/mneikes/arm-chacha20/tree/master. 20
24. Erik Schneider and Wouter de Groot. spongent-avr, 2015. https://github.com/weedegee/spongent-avr. 20
25. 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. 20

26. 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. 17
27. Nick Sullivan. Do the ChaCha: better mobile performance with cryptography, 2015. https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/. 2
28. 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. 12
29. 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. 12
30. Gilles Van Assche and Ronny Van Keer. Structuring and optimizing Keccak software. 2016. http://ccccspeed.win.tue.nl/papers/KeccakSoftware.pdf. 20
31. Rhys Weatherley. Arduinolibs, 2016. https://rweather.github.io/arduinolibs/crypto.html. 20
32. 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. 12

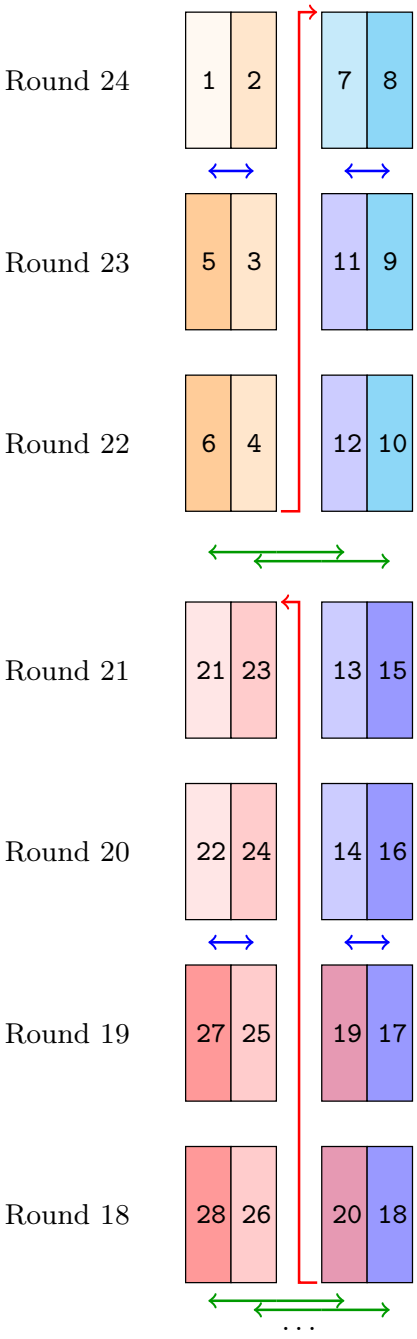
A The GIMLI permutation in C

```c
#include <stdint.h>

uint32_t rotate(uint32_t x, int bits)
{
  if (bits == 0) return x;
  return (x << bits) | (x >> (32 - bits));
}

extern void gimli(uint32_t *state)
{
  int round;
  int column;
  uint32_t x;
  uint32_t y;
  uint32_t z;

  for (round = 24; round > 0; --round)
  {
    for (column = 0; column < 4; ++column)
    {
      x = rotate(state[    column], 24);
      y = rotate(state[4 + column],  9);
      z =        state[8 + column];

      state[8 + column] = x ^ (z << 1) ^ ((y&z) << 2);
      state[4 + column] = y ^ x        ^ ((x|z) << 1);
      state[column]     = z ^ y        ^ ((x&y) << 3);
    }

    if ((round & 3) == 0) { // small swap: pattern s...s...s... etc.
      x = state[0];
      state[0] = state[1];
      state[1] = x;
      x = state[2];
      state[2] = state[3];
      state[3] = x;
    }
    if ((round & 3) == 2) { // big swap: pattern ..S...S...S. etc.
      x = state[0];
      state[0] = state[2];
      state[2] = x;
      x = state[1];
      state[1] = state[3];
      state[3] = x;
    }

    if ((round & 3) == 0) { // add constant: pattern c...c...c... etc.
      state[0] ^= (0x9e377900 | round);
    }
  }
}
```

B The GIMLI-Hash in C

```c
#include "gimli_hash.h"

#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define rateInBytes 16

void Gimli_hash(const uint8_t *input,
                uint64_t inputByteLen,
                uint8_t *output,
                uint64_t outputByteLen)
{
    uint32_t state[12];
    uint8_t* state_8 = (uint8_t*)state;
    uint64_t blockSize = 0;
    uint64_t i;

    // === Initialize the state ===
    memset(state, 0, sizeof(state));

    // === Absorb all the input blocks ===
    while(inputByteLen > 0) {
        blockSize = MIN(inputByteLen, rateInBytes);
        for(i=0; i<blockSize; i++)
            state_8[i] ^= input[i];
        input += blockSize;
        inputByteLen -= blockSize;

        if (blockSize == rateInBytes) {
            gimli(state);
            blockSize = 0;
        }
    }

    // === Do the padding and switch to the squeezing phase ===
    state_8[blockSize] ^= 0x1F;
    // Add the second bit of padding
    state_8[rateInBytes-1] ^= 0x80;
    // Switch to the squeezing phase
    gimli(state);

    // === Squeeze out all the output blocks ===
    while(outputByteLen > 0) {
        blockSize = MIN(outputByteLen, rateInBytes);
        memcpy(output, state, blockSize);
        output += blockSize;
        outputByteLen -= blockSize;

        if (outputByteLen > 0)
            gimli(state);
    }
}
```

C Computation order on AVR



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 $s_{0,2}$, $s_{0,3}$
...

Fig. 7: Computation order on AVR

D Test Vectors for GIMLI-Hash

input: "There's plenty for the both of us, may the best Dwarf win."
input (bytes):
`54686572 65277320 706c656e 74792066 6f722074 68652062 6f746820 6f662075`
`732c206d 61792074 68652062 65737420 44776172 66207769 6e2e`
output:
`baf8f9bfcb870a122e9c341f7dc5b27d57f3376ddc875f4aecbaacb89f6b90a4`

input: "If anyone was to ask for my opinion, which I note they're not, I'd say we were taking the long way around."
input (bytes):
`49662061 6e796f6e 65207761 7320746f 2061736b 20666f72 206d7920 6f70696e`
`696f6e2c 20776869 63682049 206e6f74 65207468 65792772 65206e6f 742c2049`
`27642073 61792077 65207765 72652074 616b696e 67207468 65206c6f 6e672077`
`61792061 726f756e 642e`
output:
`e52dffeb49300acdcf43f33d66ac0349bcc0c2ef54c29de5174143823d748f41`

input: "It's true you don't see many Dwarf-women. And in fact, they are so alike in voice and appearance, that they are often mistaken for Dwarf-men. And this in turn has given rise to the belief that there are no Dwarf-women, and that Dwarves just spring out of holes in the ground! Which is, of course, ridiculous."
input (bytes):
`49742773 20747275 6520796f 7520646f 6e277420 73656520 6d616e79 20447761`
`72662d77 6f6d656e 2e20416e 6420696e 20666163 742c2074 68657920 61726520`
`736f2061 6c696b65 20696e20 766f6963 6520616e 64206170 70656172 616e6365`
`2c207468 61742074 68657920 61726520 6f667465 6e206d69 7374616b 656e2066`
`6f722044 77617266 2d6d656e 2e202041 6e642074 68697320 696e2074 75726e20`
`68617320 67697665 6e207269 73652074 6f207468 65206265 6c696566 20746861`
`74207468 65726520 61726520 6e6f2044 77617266 2d776f6d 656e2c20 616e6420`
`74686174 20447761 72766573 206a7573 74207370 72696e67 206f7574 206f6620`
`686f6c65 7320696e 20746865 2067726f 756e6421 20576869 63682069 732c206f`
`6620636f 75727365 2c207269 64696375 6c6f7573 2e`
output:
`db7e9ebd4043bddbc922261615282793c5268af026cc1741f699010b44194f8e`

input: "" (empty string)
input (bytes): *(0 bytes)*
output:
`bd2a1b1cdab81f9fea9d5fd513372ab9d1481428385de2b2d3571d8504fdd703`

E 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.

F Avalanche Criterion

The following tables shows the average number of flipped bits after 10 rounds if the bit at the *index* position is flipped. Sampling has been done over 1024 independent random inputs.

Table 5: average number bit flipped and standard deviation
format: *bit index* $(\bar{x}, \sigma)$

$s_{0,0}$		$s_{1,0}$		$s_{2,0}$	
000	(192.3, 9.6)	032	(192.5, 9.5)	064	(191.8, 9.9)
001	(191.8, 9.8)	033	(192.2, 9.8)	065	(192.8, 9.9)
002	(191.8, 9.8)	034	(192.0, 10.2)	066	(191.7, 9.5)
003	(192.3, 9.6)	035	(191.7, 9.7)	067	(191.5, 9.6)
004	(192.1, 9.8)	036	(192.4, 9.6)	068	(192.0, 10.0)
005	(191.7, 9.9)	037	(191.3, 9.7)	069	(192.0, 10.1)
006	(192.1, 9.9)	038	(191.8, 9.9)	070	(192.0, 9.5)
007	(191.9, 9.8)	039	(192.2, 9.8)	071	(191.2, 9.8)
008	(191.7, 9.8)	040	(192.2, 9.9)	072	(192.2, 9.9)
009	(192.1, 9.8)	041	(192.6, 10.0)	073	(191.7, 9.6)
010	(191.8, 10.1)	042	(192.1, 9.9)	074	(192.2, 9.9)
011	(191.7, 9.9)	043	(192.7, 9.9)	075	(191.8, 9.7)
012	(191.9, 9.8)	044	(191.9, 9.8)	076	(191.9, 9.9)
013	(191.7, 9.3)	045	(192.1, 9.4)	077	(192.4, 9.5)
014	(192.2, 9.6)	046	(192.5, 9.7)	078	(191.9, 9.6)
015	(192.4, 9.5)	047	(192.5, 9.7)	079	(192.0, 10.2)
016	(191.9, 9.7)	048	(192.3, 9.9)	080	(191.8, 9.7)
017	(191.9, 9.7)	049	(192.0, 9.6)	081	(192.7, 9.6)
018	(191.5, 9.7)	050	(191.8, 9.9)	082	(192.2, 9.8)
019	(191.8, 9.6)	051	(191.5, 9.7)	083	(191.9, 9.9)
020	(191.9, 9.7)	052	(192.0, 10.1)	084	(192.5, 9.9)
021	(192.0, 9.8)	053	(192.0, 9.8)	085	(192.1, 9.9)
022	(192.1, 9.7)	054	(191.6, 9.8)	086	(192.2, 9.6)
023	(191.8, 10.2)	055	(192.3, 9.9)	087	(191.6, 9.9)
024	(191.9, 10.0)	056	(191.9, 9.6)	088	(191.6, 9.7)
025	(192.1, 9.9)	057	(192.1, 9.5)	089	(192.4, 9.5)
026	(191.8, 9.9)	058	(192.2, 10.3)	090	(192.5, 10.1)
027	(191.9, 10.1)	059	(192.1, 9.8)	091	(191.8, 9.8)
028	(192.0, 10.0)	060	(192.7, 10.1)	092	(192.2, 9.6)
029	(192.4, 9.9)	061	(192.0, 9.5)	093	(191.9, 10.1)
030	(192.0, 10.0)	062	(192.0, 9.8)	094	(192.3, 9.7)
031	(192.1, 10.0)	063	(191.6, 10.2)	095	(191.7, 9.7)

$s_{0,1}$		$s_{1,1}$		$s_{2,1}$	
096	(191.8, 9.7)	128	(192.5, 9.8)	160	(192.0, 9.8)
097	(191.3, 9.9)	129	(192.0, 10.1)	161	(192.0, 9.8)
098	(192.1, 10.1)	130	(191.9, 10.0)	162	(191.7, 9.7)
099	(191.7, 9.9)	131	(191.9, 10.0)	163	(191.6, 9.7)
100	(191.8, 9.8)	132	(192.1, 10.0)	164	(192.2, 9.9)
101	(191.7, 10.0)	133	(192.1, 9.7)	165	(192.1, 10.3)
102	(192.3, 10.0)	134	(192.0, 9.7)	166	(192.3, 10.1)
103	(191.8, 9.6)	135	(192.4, 9.4)	167	(192.0, 9.8)
104	(192.0, 9.4)	136	(192.4, 9.9)	168	(192.2, 9.8)
105	(191.8, 9.8)	137	(191.9, 9.8)	169	(192.1, 9.5)
106	(192.2, 10.0)	138	(191.9, 10.3)	170	(191.6, 9.6)
107	(192.3, 9.6)	139	(191.6, 9.7)	171	(192.2, 10.0)
108	(192.0, 9.7)	140	(191.8, 9.9)	172	(192.5, 10.1)
109	(191.9, 9.5)	141	(192.6, 9.8)	173	(192.2, 9.6)
110	(192.1, 9.6)	142	(191.6, 9.8)	174	(192.6, 9.9)
111	(192.5, 9.6)	143	(191.7, 9.8)	175	(192.3, 9.6)
112	(192.0, 9.6)	144	(192.0, 9.8)	176	(192.0, 9.8)
113	(191.9, 9.6)	145	(191.8, 9.6)	177	(192.4, 9.9)
114	(191.7, 9.5)	146	(191.7, 10.0)	178	(192.5, 9.6)
115	(192.4, 9.8)	147	(191.7, 9.9)	179	(191.5, 9.5)
116	(192.0, 9.7)	148	(191.7, 9.9)	180	(191.9, 9.7)
117	(191.8, 9.8)	149	(192.1, 9.7)	181	(192.4, 9.7)
118	(192.1, 9.6)	150	(191.7, 9.9)	182	(192.0, 9.9)
119	(192.4, 10.0)	151	(191.9, 10.0)	183	(191.5, 9.9)
120	(191.9, 10.0)	152	(191.9, 9.9)	184	(192.1, 9.8)
121	(191.6, 9.6)	153	(192.5, 10.1)	185	(191.8, 9.8)
122	(192.1, 9.6)	154	(192.2, 10.1)	186	(191.9, 9.7)
123	(191.6, 9.6)	155	(191.6, 9.9)	187	(192.1, 9.8)
124	(191.8, 9.6)	156	(191.9, 9.3)	188	(192.2, 9.9)
125	(191.6, 9.7)	157	(192.2, 9.8)	189	(192.2, 9.6)
126	(191.6, 9.8)	158	(192.1, 9.9)	190	(192.4, 9.8)
127	(192.2, 9.8)	159	(191.6, 9.5)	191	(192.8, 10.1)

$s_{0,2}$		$s_{1,2}$		$s_{2,2}$	
192	(192.0, 9.8)	224	(192.5, 9.8)	256	(192.2, 10.0)
193	(191.6, 9.9)	225	(191.5, 10.2)	257	(192.4, 9.7)
194	(191.9, 10.0)	226	(192.9, 9.8)	258	(191.9, 9.6)
195	(192.0, 9.6)	227	(191.5, 9.5)	259	(192.5, 9.7)
196	(191.5, 10.0)	228	(192.3, 9.8)	260	(191.9, 9.9)
197	(192.1, 9.9)	229	(192.2, 9.8)	261	(192.9, 9.5)
198	(191.9, 9.8)	230	(191.9, 9.7)	262	(192.4, 9.8)
199	(191.7, 9.4)	231	(191.9, 9.8)	263	(191.9, 10.0)
200	(192.0, 9.6)	232	(192.5, 10.2)	264	(191.9, 10.0)
201	(191.3, 9.8)	233	(192.0, 9.9)	265	(192.2, 9.6)
202	(191.5, 9.9)	234	(191.6, 10.0)	266	(191.9, 10.0)
203	(192.0, 9.9)	235	(192.1, 9.7)	267	(191.9, 10.0)
204	(191.8, 9.8)	236	(191.9, 9.4)	268	(191.9, 9.7)
205	(191.9, 9.9)	237	(192.1, 9.3)	269	(191.9, 9.6)
206	(192.2, 9.9)	238	(191.9, 9.8)	270	(192.2, 9.6)
207	(192.4, 9.8)	239	(192.2, 10.0)	271	(192.1, 9.7)
208	(191.7, 10.2)	240	(191.8, 9.7)	272	(191.7, 9.9)
209	(191.9, 9.7)	241	(191.6, 10.4)	273	(191.9, 9.8)
210	(192.0, 9.5)	242	(192.0, 10.0)	274	(192.4, 10.1)
211	(192.3, 10.0)	243	(192.0, 9.6)	275	(192.0, 9.7)
212	(192.3, 9.9)	244	(192.5, 9.5)	276	(192.3, 10.0)
213	(191.8, 9.4)	245	(192.3, 9.8)	277	(192.1, 9.9)
214	(192.3, 9.8)	246	(192.0, 9.7)	278	(192.3, 9.8)
215	(192.0, 10.2)	247	(192.3, 9.6)	279	(191.5, 10.0)
216	(191.8, 10.2)	248	(192.1, 10.2)	280	(192.0, 9.6)
217	(192.4, 9.8)	249	(192.0, 9.6)	281	(191.6, 9.8)
218	(192.3, 10.0)	250	(191.7, 9.7)	282	(192.2, 9.8)
219	(192.1, 9.7)	251	(192.3, 9.5)	283	(192.1, 9.9)
220	(192.1, 9.9)	252	(192.0, 9.7)	284	(191.5, 9.9)
221	(191.8, 10.0)	253	(192.4, 10.4)	285	(192.1, 9.7)
222	(192.6, 9.8)	254	(192.3, 9.6)	286	(191.9, 9.7)
223	(191.8, 10.0)	255	(192.3, 9.9)	287	(192.1, 9.9)

$s_{0,3}$		$s_{1,3}$		$s_{2,3}$	
288	(191.7, 9.6)	320	(192.2, 9.6)	352	(191.6, 9.7)
289	(192.3, 10.0)	321	(192.1, 9.8)	353	(192.3, 9.9)
290	(192.0, 9.8)	322	(191.6, 9.7)	354	(192.2, 9.7)
291	(192.2, 10.2)	323	(192.2, 9.4)	355	(191.7, 9.9)
292	(192.3, 9.5)	324	(192.0, 9.6)	356	(191.5, 9.8)
293	(191.8, 10.0)	325	(191.5, 9.7)	357	(192.3, 9.7)
294	(192.0, 9.7)	326	(192.5, 10.2)	358	(192.2, 9.8)
295	(192.5, 9.7)	327	(192.6, 10.0)	359	(191.7, 9.9)
296	(192.1, 9.7)	328	(192.0, 9.6)	360	(192.0, 10.0)
297	(192.1, 9.4)	329	(192.2, 9.9)	361	(192.2, 9.7)
298	(192.1, 9.8)	330	(192.0, 9.8)	362	(191.9, 9.5)
299	(191.8, 9.7)	331	(191.9, 9.9)	363	(191.9, 9.7)
300	(192.2, 9.5)	332	(192.1, 9.7)	364	(191.9, 10.1)
301	(192.3, 10.2)	333	(192.5, 9.9)	365	(191.9, 9.9)
302	(192.1, 9.7)	334	(191.9, 9.8)	366	(192.0, 9.9)
303	(191.9, 10.0)	335	(191.9, 9.6)	367	(192.0, 9.8)
304	(192.0, 10.2)	336	(192.3, 9.7)	368	(191.9, 9.5)
305	(191.9, 9.8)	337	(191.7, 9.6)	369	(191.9, 9.9)
306	(192.5, 9.5)	338	(192.0, 9.7)	370	(192.1, 10.0)
307	(191.9, 9.5)	339	(192.1, 10.2)	371	(191.9, 10.2)
308	(191.8, 9.8)	340	(192.0, 9.8)	372	(191.8, 9.8)
309	(192.4, 9.6)	341	(192.3, 9.6)	373	(191.9, 9.8)
310	(192.0, 9.8)	342	(192.3, 9.8)	374	(192.1, 10.1)
311	(191.5, 9.7)	343	(191.7, 9.6)	375	(192.2, 9.7)
312	(192.3, 10.0)	344	(192.4, 10.3)	376	(192.3, 9.9)
313	(191.8, 9.7)	345	(192.2, 9.9)	377	(192.3, 9.7)
314	(192.2, 10.2)	346	(192.2, 10.0)	378	(192.0, 9.8)
315	(192.4, 9.8)	347	(192.3, 9.9)	379	(191.4, 10.0)
316	(192.2, 9.9)	348	(191.8, 9.9)	380	(191.9, 9.9)
317	(192.3, 9.7)	349	(192.3, 9.3)	381	(191.8, 9.8)
318	(191.8, 9.5)	350	(192.4, 9.6)	382	(191.9, 9.7)
319	(192.2, 9.6)	351	(192.1, 9.8)	383	(191.0, 9.6)

G Differential Cryptanalysis

Proof (Proof of Lemma 1).

We want to show how to compute the set of valid differentials for a given input difference

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

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

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

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

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

and gives us the set of possible output differences

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

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

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

giving the set of possible output differences

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

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

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

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}. □

.

Table 6: Optimal differential trail for 8-round Gimli.

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

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

H 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 $\boldsymbol{x}$ and $\boldsymbol{y}$ denote MILP variables corresponding to input and output of Gimli, respectively. In the normal use of the division property, $\boldsymbol{x}$ 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 $\boldsymbol{y} = \boldsymbol{e}_j$, where $\boldsymbol{e}_j$ 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 $\boldsymbol{y}$ 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 $\boldsymbol{y} = \boldsymbol{e}_j$ 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.