Secure Boolean Masking of Gimli

Optimization and Evaluation on the Cortex-M4

Tzu-Hsien Chang[1,4], Yen-Ting Kuo[5], Jiun-Peng Chen[3,4](✉), and Bo-Yin Yang[2,4]

[1] Cybersecurity Center of Excellence Program, National Applied Research Laboratories, Taipei, Taiwan

[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan
`by@crypto.tw`

[3] Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan
`jpchen@ieee.org`

[4] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
`r10921a19@ntu.edu.tw`

[5] University of Tokyo, Tokyo, Japan

Abstract. Gimli is a highly secure permutation with high performance across a broad range of platforms. However, side-channel analysis poses a threat to the Gimli without any masking protection. To resist side-channel analysis, the current state of the art of Boolean masking in software proposes an efficient scheme of bitwise logic operations. In practice, a software implementation of masked Gimli may leak information due to pipeline registers and also due to other effects. To avoid unintentional leakage, costly overheads are required, such as more randomness and higher-order share implementation. For our implementation, we present two efficient optimal masked Gimli implementations for the ARM Cortex-M4 on the STM32F407 Discovery(a common Cortex-M4 board) and evaluate their security using TVLA. In 3-shared scenarios, our approach performs with high security with a t-statistic value bounded by a threshold of 4.5 standard deviations, which implies that leakage information cannot be detected. Furthermore, our results promise significant performance improvement for the implementation on Cortex-M processors, with a reduction of the amount of overhead for masking by 61% and 76% for 2 and 3 shared scenarios, respectively.

Keywords: Gimli · ARM Cortex-M4 · Threshold implementation · DPA

1 Introduction

In several emerging areas (e.g. sensor networks, healthcare, distributed control systems, or the Internet of Things), highly resource-constrained devices are interconnected, typically communicating wirelessly with one another, and working in concert to accomplish some task. Because the majority of current cryptographic algorithms were designed for desktop/server environments, many of these algorithms do not fit into these devices.

Gimli [6], a high secure permutation with high performance across a broad range of platforms, is suitable for use in constrained environments [20]. Ciphers using the three operations + (add), $\lll$ (rotation) and XOR are usually called ARX ciphers, and these include SPECK [5] and ChaCha20 [7]. All of Gimli, SPECK, and ChaCha20, can be attacked by side-channel. Side-channel analyses are physical attacks based on the exploitation of the information (typically time, power consumption, or electromagnetic radiation), which can be measured while the cryptographic algorithm is operating on the device. Differential power analysis (DPA) [21] and Correlation Power Analysis (CPA) [11] based on power consumption or electromagnetic radiation, have received significant attention since it is very powerful and does not usually require detailed knowledge of the target device to be successfully implemented.

In ARX ciphers, Addition without protection is vulnerable to DPA [21,31]. Furthermore, early work on masking for addition in software has a high-performance overhead [18]. Gimli, replacing addition $a+b$ with a similar bitwise operation $a \oplus b \oplus ((c \wedge b) \ll 1)$ is suitable for implementing a DPA resistant cryptographic algorithm. There are different papers discussing attack and resistance of Gimli, such as [15,16]. However, there have not been many papers in the past discussing Gimli's resistance to SCA in optimized software implementations.

Threshold Implementation (TI) is a masking scheme based on secret sharing and multi-party computation [23–25]. TI is fairly simple to apply to a wide range of ciphers [17,18], and its implementation is not very error-prone if a known set of requirements and best practices is followed. Though early works on masking suggested using two-share TI to reduce the size of the sequential logic in hardware implementations [12,28], however, it is vulnerable to implement two-share TI for most microprocessors in practice. For example, Cortex-M3 and Cortex-M4 pipeline register leak information about Hamming distance between the current operand value and the previous one [13]. Since these problems also appear in most of the software implementation of Boolean masking, implementing a secure masking algorithm is very challenging.

Therefore, how to make Gimli have good execution efficiency in Cortex-M3 and Cortex-M4, resist Side-Channel Attacks, and evaluate the protective effect, all need to be considered.

1.1 Our Contributions

In this paper, we present an efficient and high-security level method for masked Gimli without leakage due to pipeline registers in embedded software applications. We investigate how to implement the TI for the non-linear layer of Gimli, finding possibilities to optimize the instruction count for ARM implementations.

First of all, we implement the Gimli for ARM Cortex-M3 and Cortex-M4 processors and optimize it on the assembly level by using ARM features such as the flexible second operand and by minimizing the number of memory operations.

Second, the Threshold Implementation of GIMLI to resist Side-Channel Attacks, two-share and three-share masking are presented, respectively. For two-share masking, we construct masked non-linear layers of Gimli based on SecAND and SecOR [10], which do not consume entropy and could utilize the flexible second operand to reduce the cycle count. For three-share masking, we are inspired by *changing of the guard* [14], which is a generic method to generate the threshold implementation scheme.

Finally, we use the Test Vector Leakage Assessment (TVLA) method to evaluate the t-test score of the threshold implementation on Cortex-M4. These inspection methods follow the ISO/IEC 17825 [1]. To reduce leakage due to pipeline registers, we rearrange the parallel instruction of two-share TI. However, it is hard to limit the t-value to a threshold of 4.5 standard deviations. This problem could be more severe on other platforms. On the other hand, the statistical value of our three-share TI is inside the ± 4.5 [30] interval for every point in time.

As mentioned above, we qualify DPA-resistant software implementations and prove that our three-share TI is uniform without additional randomness by the reversible property of Gimli. The method of proof about uniformity can also generate a high secure three-share TI of NORX family ciphers [3] because of their similar structure. Furthermore, the three-share TI is a better choice in the strictly secured scenarios, since it can prevent information leakage due to the architecture of microprocessors. These different architectures are a trade-off between performance and security to perform the full permutation.

2 Preliminaries

2.1 GIMLI

GIMLI is a 384-bit permutation designed to achieve high security with high performance across a broad range of platforms. In this paper, we focus on the GIMLI-CIPHER, which performs Authenticated Encryption with Associated Data (AEAD).

The GIMLI Permutation. The GIMLI permutation applies a sequence of rounds to a 384-bit state. We denote by $W = \{0,1\}^{32}$ the set of bit-strings of length 32. We will refer to the elements of this set as *words*; The state is represented as a 3×4 matrix of words $W^{3\times 4}$; the rows are named x, y, z; the columns are enumerated by $0, 1, 2, 3$; the round number is denoted by r. For example, x_1^8 denotes the second 32-bit word before the execution of round 8. Finally, 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.

Algorithm 1 describes how this state is permuted in 24 rounds: a non-linear layer starts with round 24 and ends with round 1. During each round, the state is first substituted and permuted (SP-Box). Every second round, the state is mixed linearly (alternating between a "small" or "big" swap). Finally, in every fourth round, a constant is added.

Algorithm 1: The Gimli permutation

```
input  : $s = (s_{i,j}) \in W^{3\times4}$
output: Gimli$(s) = (s_{i,j}) \in W^{3\times4}$
for $r = 24$ down to $1$ do
    for $j = 0$ to $3$ do
        $x \leftarrow s_{0,j} \lll 24$                                ▷ SP-box
        $y \leftarrow s_{1,j} \lll 9$
        $z \leftarrow s_{2,j}$
        $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$
        $s_{1,j} \leftarrow y \oplus x \oplus ((x \vee z) \ll 1)$
        $s_{0,j} \leftarrow z \oplus y \oplus ((x \wedge y) \ll 3)$
    end
    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 $r$ mod $4 = 0$ then
        $s_{0,0} \leftarrow s_{0,0} \oplus$ 0x9e377900 $\oplus r$                  ▷ Add constant
    end
end
return $(s_{i,j})$
```

2.2 Threshold Implementation

Threshold Implementation [25] (TI) is a special case of Boolean masking. Even in the presence of glitches, it has been proven secure against first-order differential power analysis for digital circuits. The advantages are that it does not need fresh random values after every non-linear transformation, unlike traditional masking methods.

Definition. TIs use shares with the following properties: correctness, incompleteness, and uniformity:

Correctness states that applying the sub-functions to a valid shared input must always yield a valid sharing of the correct output.
$\mathbf{z} = \bigoplus_{i=1}^{d} z_i = \bigoplus_{i=1}^{d} f_i(x, y, ...) = f(\mathbf{x}, \mathbf{y}, ...)$

Non-Completeness requires sub-functions f_i of a shared function F to be independent of at least one input share for first-order SCA resistance. That is, a function $F(x, y, ...)$ shall be split into sub-functions $f_i(x_{j \neq i}, y_{j \neq i}, ...)$. This requirement was updated in [9] to require any d sub-functions to be independent of at least one input share to achieve d-th order SCA resistance. Non-completeness ensures that the final circuit is not affected by glitches. Since glitches can only occur in sub-functions f_i, and each sub-function has insufficient knowledge to reconstruct a secret state (since it has no knowledge of at least one share x_i), no leakage can be caused by glitches.

Uniformity requires all intermediate states (shares) to be uniformly distributed. Uniformity ensures that the mean leakages are state-independent, a key requirement to thwart first-order DPA. To ensure uniformity in a circuit, it suffices to ensure uniformity for the output share of each function, as well as for the inputs of the circuit. This property is often the most difficult to achieve and most costly in terms of hardware area.

2.3 The ARM Cortex-M Processors

The ARM Cortex-M4 is a 32-bit RISC processor based on the ARMv7E-M architecture, targeting low-cost and energy-efficient microcontrollers. It is equipped with 13 general-purpose registers (GPRs, `r0-r12`), plus the link register (`lr`, which holds the return address from a subroutine), the stack pointer (`sp`), and the program counter (`pc`). The `lr` register can also be used as a GPR after its content has been saved to the stack. Besides GPRs, most existing M4s also have 32 floating-point registers ("M4f"), as is the case for the cheap, widely available, and popular STM32F407 Discovery board. Aside from performing floating-point instructions, floating point registers can be used as a cache to store frequently used constants or loop counters by using the `vmov` instruction that moves 32 bits between general-purpose and floating-point registers in exactly 1 cycle.

A very helpful feature, called "flexible second operand", can also save lots of time. In most data-processing instructions, the second operand can be a register shifted or rotated in the same instruction without causing extra latency. A shift or rotation that operates on a flexible second operand can be the arithmetic right shift (`ASR`), logical right or left shift (`LSL` and `LSR`), or rotation (`ROR`), plus rotate right extended by one bit (`RRX`). For example, the instruction `ADD r0,r1,r2,LSL #3` can calculate $r0 = r1 + (r2 \ll 3)$ in one clock cycle. As all common instructions like `EOR` and `AND` support the flexible second operand, shifts and rotates on registers can be had "for free" in many cases.

Since the functionalities we use for the implementation are also present in the ARM Cortex-M3 processor (which is missing only what's usually called DSP instructions compared to the M4), our work can be extended there without too much trouble, needing only to replace all caching in the floating point registers with stack access operations (and possibly re-optimizing).

3 Side-Channel Countermeasures

In this section, we provide three ways to avoid information leakage in Gimli implementation. In Sect. 3.1 and Sect. 3.2, we discuss 2-share TI targeted for software implementations since they require a register stage after some operations to achieve non-completeness. This is easier to accomplish on software than highly parallel hardware implementations because each operation is stored to a register anyway. Therefore, if none of the individual terms recombines 2 shares of the same variable prior to the register write, and if each input share is independently uniform, non-completeness is always fulfilled. On the other hand, 3-share TI can be implemented on both platforms since it does not require overhead on the register areas and additional time cycles in hardware implementations.

For all the methods below, the first step is to apply random masking to all of the twelve 32-bit states. In an s-share scheme, we generate $12(s-1)$ random numbers and exclusive-or each state with $s-1$ random numbers. For example, in a 3-share scheme, state $s_{0,0}$ will xor 2 random numbers R_0 and R_1. The shared state will then be $\mathbf{s}_{0,0} = (s_{0,0} \oplus R_0 \oplus R_1, R_0, R_1)$. Though we perform these three ways on Gimli implementation, these methods are not unique to Gimli but are also implementable on other NORX family ciphers because of their similar structure.

3.1 2-Share with ChaCha-8 Randomness

In this method, we recall that the classical Boolean masking schemes in AND/OR gates simply put one random value in one share, and the other share is the value we want to protect xor'ed with that random value, so that xor'ing the two shares yields the sensitive intermediate and both shares are uniformly random. We use the same idea on the SP-box in Gimli with 3 random numbers R_0, R_1, R_2, and the result is given in Eq. 1, where $\{x_0, x_1\}$, $\{x'_0, x'_1\}$ is the input, output shares for $\mathbf{x}$ respectively. $\mathbf{y}$ and $\mathbf{z}$ are the same.

$$\begin{cases} x'_0 = R_0 \oplus x_0 \oplus (z_0 \ll 1) \oplus ((y_0 \wedge z_0) \ll 2) \oplus ((y_0 \wedge z_1) \ll 2) \oplus ((y_1 \wedge z_0) \ll 2) \\ x'_1 = R_0 \oplus x_1 \oplus (z_1 \ll 1) \oplus ((y_1 \wedge z_1) \ll 2) \\ y'_0 = R_1 \oplus y_0 \oplus \quad x_0 \quad \oplus ((x_0 \vee z_0) \ll 1) \oplus ((\neg x_0 \wedge z_1) \ll 1) \oplus ((x_1 \wedge \neg z_0) \ll 1) \\ y'_1 = R_1 \oplus y_1 \oplus \quad x_1 \quad \oplus ((x_1 \wedge z_1) \ll 1) \\ z'_0 = R_2 \oplus z_0 \oplus \quad y_0 \quad \oplus ((x_0 \wedge y_0) \ll 3) \oplus ((x_0 \wedge y_1) \ll 3) \oplus ((x_1 \wedge y_0) \ll 3) \\ z'_1 = R_2 \oplus z_1 \oplus \quad y_1 \quad \oplus ((x_1 \wedge y_1) \ll 3) \end{cases} \tag{1}$$

The correctness of this equation can simply be checked by xor'ing the two shares of each variable. For example:

$$\begin{aligned} x' &= x'_0 \oplus x'_1 \\ &= (x_0 \oplus x_1) \oplus ((z_0 \oplus z_1) \ll 1) \oplus (((y_0 \wedge z_0) \oplus (y_0 \wedge z_1) \oplus (y_1 \wedge z_0) \oplus (y_1 \wedge z_1)) \ll 2) \\ &= x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2) \end{aligned}$$

If the subscript(s) of a term have 0, then we split that term into the first share, the others are combined into the second share. Because in this way, we can optimize the memory operations in assembly code more easily. $12 \times 24 = 288$ random numbers are required for the 24 rounds of GIMLI. While STM32F4 devices feature a true random number generator, it takes approximately 60 to 70 clock cycles to generate one 32-bit random integer. Hence we implement a ChaCha-8 pseudo-random number generator which reduces the clock cycle count to around 25 to 30 per random number. We choose the ChaCha-8 pseudo-random number generator because it is fast and currently considered to be 2^{256} bit security and highly unlikely to be less secure than Gimli's design security level [22], but of course, we can substitute any secure and fast stream cipher.

3.2 2-Share with Optimal Masking

In [10], a state-of-the-art masking mechanism was proposed. We can replace the non-linear operations in Eq. 1 by the operations of 2-share Threshold implementation, according to Table 1.

Table 1. Expressions for different operations.

Operation	Expression
SecAnd	$z_1 = (x_1 \wedge y1) \oplus (x_1 \oplus \neg y_2)$ $z_2 = (x_2 \wedge y1) \oplus (x_2 \oplus \neg y_2)$
SecOr	$z_1 = (x_1 \wedge y1) \oplus (x_1 \oplus \neg y_2)$ $z_2 = (x_2 \wedge y1) \oplus (x_2 \oplus \neg y_2)$

By eliminating the requirement of fresh randomness, it outperformed the classical Boolean masking schemes on software platforms. Here we utilize their result and construct a 2-share optimal masking of GIMLI.

$$\begin{cases} x_0' = x_0 \oplus (z_0 \lll 1) \oplus (((y_0 \wedge z_0) \oplus (\neg y_1 \vee z_0)) \lll 2) \\ x_1' = x_1 \oplus (z_1 \lll 1) \oplus (((y_0 \wedge z_1) \oplus (\neg y_1 \vee z_1)) \lll 2) \\ y_0' = y_0 \oplus \quad x_0 \quad \oplus (((x_0 \wedge z_0) \oplus (x_0 \vee z_1)) \lll 1) \\ y_1' = y_1 \oplus \quad x_1 \quad \oplus (((x_1 \vee z_0) \oplus (x_1 \wedge z_1)) \lll 1) \\ z_0' = z_0 \oplus \quad y_0 \quad \oplus (((x_0 \wedge y_0) \oplus (x_0 \vee \neg y_1)) \lll 3) \\ z_1' = z_1 \oplus \quad y_1 \quad \oplus (((x_1 \wedge y_0) \oplus (x_1 \vee \neg y_1)) \lll 3) \end{cases} \quad (2)$$

Notice that we only put the negation gate before y shares because ARM's flexible second operand only works on the second operand (the negated one) in `ORN` instructions. Since the y shares must be shifted, they need to be used as the second operands. Also, the method does require a register stage for the operations of the AND and OR gates. For the application on hardware platforms, this would be a big trade-off in terms of speed and area.

To prove that the modified function does not leak any information about any sensitive variable, we notice that in the formula of $\mathbf{x}$ and $\mathbf{y}$ shares, the only shares that contain both subscripts are only in the AND and OR gates. Since they don't leak the information about y and z, respectively, the whole calculation will not leak either since different shares are independent.

For the $\mathbf{z}$ shares, however, there is a chance that it might leak the information about y because both shares are present in both parts. We can check this very efficiently by performing a few bitwise operations on the truth tables and computing the Hamming weight. For example, a non-constant function f leaks information about function k if and only if

$$\frac{HW(k \wedge f)}{HW(f)} \neq \frac{HW(k \wedge \neg f)}{HW(\neg f)},$$

where $\mathrm{HW}(g)$ denotes the Hamming weight of the truth table of function g [10].

To confirm the $\mathbf{z}$ shares, we may then set z'_0 as f and y as k and calculate the Hamming weight for all variables indexed from 0 to 15 since the formula shifts variables to the left at most 3. Therefore, we can prove that the function of 2-share optimal masking does not leak information about the sensitive state.

3.3 3-Share Threshold Implementation

To resist-first order DPA in hardware security, at least $t+1$ shares of masking is required [9] for the TI method, where t is the degree of a function and is 2 for Gimli. We begin by constructing a threshold implementation of a 3-share Gimli permutation.

$$\begin{cases} x'_0 = x_1 \oplus (z_1 \lll 1) \oplus (((y_1 \wedge z_1) \oplus (y_1 \wedge z_2) \oplus (y_2 \wedge z_1)) \lll 2) \\ x'_1 = x_2 \oplus (z_2 \lll 1) \oplus (((y_2 \wedge z_2) \oplus (y_2 \wedge z_0) \oplus (y_0 \wedge z_2)) \lll 2) \\ x'_2 = x_0 \oplus (z_0 \lll 1) \oplus (((y_0 \wedge z_0) \oplus (y_0 \wedge z_1) \oplus (y_1 \wedge z_0)) \lll 2) \\ y'_0 = y_1 \oplus \quad x_1 \quad \oplus (((x_1 \wedge z_1) \oplus (x_1 \wedge \neg z_2) \oplus (\neg x_2 \wedge z_1)) \lll 1) \\ y'_1 = y_2 \oplus \quad x_2 \quad \oplus (((x_2 \vee z_2) \oplus (\neg x_2 \wedge z_0) \oplus (x_0 \wedge \neg z_2)) \lll 1) \\ y'_2 = y_0 \oplus \quad x_0 \quad \oplus (((x_0 \wedge z_0) \oplus (x_0 \wedge z_1) \oplus (x_1 \wedge z_0)) \lll 1) \\ z'_0 = z_1 \oplus \quad y_1 \quad \oplus (((x_1 \wedge y_1) \oplus (x_1 \wedge y_2) \oplus (x_2 \wedge y_1)) \lll 3) \\ z'_1 = z_2 \oplus \quad y_2 \quad \oplus (((x_2 \wedge y_2) \oplus (x_2 \wedge y_0) \oplus (x_0 \wedge y_2)) \lll 3) \\ z'_2 = z_0 \oplus \quad y_0 \quad \oplus (((x_0 \wedge y_0) \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0)) \lll 3) \end{cases} \tag{3}$$

Theorem 1. *Equation 3 constructs a threshold implementation of* Gimli *permutation. That is, it meets the definition of* ***Correctness***, ***Non-Completeness*** *and* ***Uniformity***.

Proof. For **Correctness**, we need to make sure that $(\mathbf{x}', \mathbf{y}', \mathbf{z}') = (\bigoplus x'_i, \bigoplus y'_i, \bigoplus z'_i)$. For instance, the y part can be proved: $\bigoplus_{i=0}^{2} y'_i = (x_0 \oplus x_1 \oplus x_2) \oplus (y_0 \oplus y_1 \oplus y_2) \oplus ((x_0 \oplus x_1 \oplus x_2) \vee (z_0 \oplus z_1 \oplus z_2)) \lll 1) = \mathbf{x} \oplus \mathbf{y} \oplus ((\mathbf{x} \vee \mathbf{z}) \lll 1)$. The x and z part is simple since it only contains an and gate.

For **Non-Completeness**, it can be seen that the computations of x'_0, y'_0, z'_0 do not involve components of x_0, y_0, z_0, the ones of x'_1, y'_1, z'_1 do not involve components of x_1, y_1, z_1 and the ones of x'_2, y'_2, z'_2 do not involve components of x_2, y_2, z_2.

For **Uniformity**, if the mapping of Eq. 3 is an invertible mapping from (**x**, **y**, **z**) to (**x**', **y**', **z**'), it implies that if (**x**, **y**, **z**) is a uniform sharing, then (**x**', **y**', **z**') is an uniform sharing as well [8]. It is, therefore, sufficient to show that the mapping of Eq. 3 is invertible. We will do that by giving a method to compute (**x**, **y**, **z**) from (**x**', **y**', **z**').

We do this by recovering $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ from 0-bit (rightmost) to 31-bit (leftmost) with the output $(\mathbf{x}', \mathbf{y}', \mathbf{z}')$. We denote the k-bit of x_0, say, by $x_{0,k}$. We rewrite the first equation by switching the output term to the right hand side:

$$x_1 = x'_0 \oplus (z_1 \ll 1) \oplus (((y_1 \wedge z_1) \oplus (y_1 \wedge z_2) \oplus (y_2 \wedge z_1)) \ll 2)$$

Because the terms after x'_0 have been shifted, $x_{1,0}$ can be derived from $x'_{0,0}$. Then we rewrite the equations as:

$$y_1 = y'_0 \oplus x_1 \oplus (((x_1 \wedge z_1) \oplus (x_1 \wedge \neg z_2) \oplus (\neg x_2 \wedge z_1)) \ll 1)$$

$$z_1 = z'_0 \oplus y_1 \oplus (((x_1 \wedge y_1) \oplus (x_1 \wedge \quad y_2) \oplus (\quad x_2 \wedge y_1)) \ll 3)$$

To compute $y_{1,0}$ and $z_{1,0}$, the last term is also irrelevant. Since we already knew $x_{1,0}$, we simply calculate $y_{1,0} = y'_{0,0} \oplus x_{1,0}$. And then $z_{1,0} = z'_{0,0} \oplus y_{1,0}$. We can use the same method to get the 0-bit of the other 6 shares.

Assume that $\forall a \in \{x, y, z\}, i \in \{0, 1, 2\}, n < 0, a_{i,n} = 0$, with the bit 0 to $k-1$ of all shares known, bits k of $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ can be derived easily via:

$$\begin{aligned}
x_{1,k} &= x'_{0,k} \oplus z_{1,k-1} \oplus ((y_{1,k-2} \wedge z_{1,k-2}) \oplus (y_{1,k-2} \wedge z_{2,k-2}) \oplus (y_{2,k-2} \wedge z_{1,k-2}))\\
y_{1,k} &= y'_{0,k} \oplus x_{1,k} \oplus ((x_{1,k-1} \wedge z_{1,k-1}) \oplus (x_{1,k-1} \wedge \neg z_{2,k-1}) \oplus (\neg x_{2,k-1} \wedge z_{1,k-1}))\\
z_{1,k} &= z'_{0,k} \oplus y_{1,k} \oplus ((x_{1,k-3} \wedge y_{1,k-3}) \oplus (x_{1,k-3} \wedge y_{2,k-3}) \oplus (x_{2,k-3} \wedge y_{1,k-3}))
\end{aligned}$$

This way, we can restore $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ from the output $(\mathbf{x}', \mathbf{y}', \mathbf{z}')$. Thus all possible inputs and outputs are mapped one-to-one to each other, which implies the uniformity property.

4 Implementation Details

In this section, assembly level optimizations of original and masked Gimli for ARM Cortex-M4 processors, aimed at both high-speed and compact code-size, are presented.

4.1 Optimization on Original Gimli

We optimized the original Gimli in two ways: (1) we first deal with the non-linear layer, namely the SP-box, of Gimli by exploiting the "flexible second operand" feature; (2) we optimized the big/small swap steps by minimizing the amount of memory operations.

SP-box: In order to exploit the flexible second operand feature, we slightly modify the SP-box function. Instead of calculating the rotations of y-states with a `ROR` instruction, we skip this and fix the missing rotations by rotating registers when they are AND-ed or XOR-ed, using the flexible second operand. Without using the flexible second operand, each round needs 15 operations. By not rotating the y-states beforehand and some rearrangement of the instructions, we can reduce that to 10 instructions, which is 33% smaller.

Notice that with 12 state words loaded into registers, we have only 2 other registers (here denoted a_0, a_1) available as scratch space.

Algorithm 2: Optimization on Original Gimli

```
input  : States before the SP-box ($x_0, y_0, z_0$)
output: States after the SP-box ($x_0, y_0, z_0$)
1  ROR $x_0$, #8
2  AND $a_0$, $z_0$, $y_0$, ROR, #23
3  EOR $a_0$, $x_0$, $a_0$, LSL, #2
4  ORR $a_1$, $x_0$, $z_0$
5  EOR $a_1$, $x_0$, $a_1$, LSL, #1
6  AND $x_0$, $x_0$, $y_0$, ROR, #23
7  EOR $x_0$, $z_0$, $x_0$, LSL, #3
8  EOR $x_0$, $x_0$, $y_0$, ROR, #23
9  EOR $z_0$, $a_1$, $z_0$, LSL, #1
10 EOR $y_0$, $a_2$, $y_0$, ROR, #23
```

Swap: To avoid the penalty of using slow memory operations, we want to minimize save and load instructions. Since the small and big swaps operate alternatively, the states will return to their former places after 2 small and 2 big swaps. With a total of 6 small and big swaps, this means that all states will return to their original places after 24 rounds of Gimli permutation.

If we can keep track of where the states are before and after the swap, we can simply continue to the next round with this order of registers without actually swapping anything. For example, let `r1-r12` be the content of the 12 state words. The first non-linear layer is performed on (`r1, r4, r7`), (`r2, r5, r8`), (`r3, r6, r9`), and (`r4, r8, r12`). After the first small swap, the next non-linear layer should be performed on (`r2, r4, r7`), (`r1, r5, r8`), (`r4, r6, r9`), and (`r3, r8, r12`). In this way, the linear layer can be simply omitted.

For software implementation, we modify the source code given in [19] by changing the C implementation of Gimli permutation to assembly code because that is the bottleneck of the performance. It is also where our three methods differ. And the C implementation acts as a baseline to our optimization that we compare to in the next section.

4.2 Implementation Details of Masked Gimli

Based on the idea shown in Sects. 3.1 and 3.2, we develop Algorithm 3 (SecSP1), a one round 2-share GIMLI SP-box refreshed with given random numbers, and Algorithm 4 (SecOptSP1), one round optimal masking of 2-share GIMLI. s^0 and s^1 are the 2 shares of one column in the state and R_i are random numbers generated by `ChaCha-8`.

Notice that the optimization on the original GIMLI can also be applied here as well: every shift of the **y**s can be delayed and effectively performed using the flexible second operand. We can actually count the number of operations used in these expressions of a 2-share SP-box: $2(\texttt{ROR}) + 24(\texttt{EOR}) + 12(\texttt{AND}/\texttt{ORR}) = 38$, which is about 4 times that of the original version. But because of the overhead of random number generation, the actual cost of Algorithm 3 is much higher.

Algorithm 3: 2-share SP-box (`SecSP1`)

```
    input  : $s^0 = (x_0, y_0, z_0), s^1 = (x_1, y_1, z_1), R_0, R_1, R_2 \in \{0,1\}^{32}$
    output: $(x'_0 \oplus x'_1, y'_0 \oplus y'_1, z'_0 \oplus z'_1) = SP(x_0 \oplus x_1, y_0 \oplus y_1, z_0 \oplus z_1)$
 1  $(x_0, x_1) \leftarrow (x_0 \lll 24, x_1 \lll 24)$
 2  $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge z_0, x_0 \wedge \neg z_1, x_1 \wedge \neg z_0, x_1 \vee z_1)$
 3  $(t_0, t_1) \leftarrow (R_0 \oplus s_0 \oplus s_1 \oplus s_2, R_0 \oplus s_3)$
 4  $(u_0, u_1) \leftarrow (x_0 \oplus (t_0 \ll 1), x_1 \oplus (t_1 \ll 1))$
 5  $(y'_0, y'_1) \leftarrow (u_0 \oplus (y_0 \lll 9), u_1 \oplus (u_1 \lll 9))$
 6  $(s_0, s_1, s_2, s_3) \leftarrow (z_0 \wedge (y_0 \lll 9), z_0 \wedge (y_1 \lll 9), z_1 \wedge (y_0 \lll 9), z_1 \wedge (y_0 \lll 9))$
 7  $(t_0, t_1) \leftarrow (R_1 \oplus s_0 \oplus s_1 \oplus s_2, R_1 \oplus s_3)$
 8  $(u_0, u_1) \leftarrow (x_0 \oplus (t_0 \ll 2), x_1 \oplus (t_1 \ll 2))$
 9  $(z'_0, z'_1) \leftarrow (u_0 \oplus (z_0 \ll 1), u_1 \oplus (z_1 \ll 1))$
10  $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge (y_0 \lll 9), x_0 \wedge (y_1 \lll 9), x_1 \wedge (y_0 \lll 9), x_1 \wedge (y_0 \lll 9))$
11  $(t_0, t_1) \leftarrow (R_2 \oplus s_0 \oplus s_1 \oplus s_2, R_2 \oplus s_3)$
12  $(u_0, u_1) \leftarrow (z_0 \oplus (t_0 \ll 3), z_1 \oplus (t_1 \ll 3))$
13  $(x'_0, x'_1) \leftarrow (u_0 \oplus (y_0 \lll 9), u_1 \oplus (y_1 \lll 9))$
```

We can count the operations on these expressions of the optimal 2-share SP-box as well: $2(\texttt{ROR}) + 18(\texttt{EOR}) + 12(\texttt{AND}/\texttt{ORR}) = 32$, which is about 3 times the non-shared version without additional memory manipulation because of the increased variables.

For further optimization, we notice that the linear layers (swap) happen every two rounds, it means that during the consecutive rounds where no swap happens in-between, the inputs to non-linear layers are the same three 32-bit states. Therefore, we can apply SP-box twice to the states without loading and saving to further reduce the number of memory instructions. Algorithm 5 shows the process of this idea applied to 2-share optimal masking, where $\mathbf{s}_{i,j}$ represents the 2 shares of 3 states (s_i, s_{4+j}, s_{8+j}) and `SP2` is one round SP-box applied twice in a row.

Algorithm 4: Optimal 2-share SP-box (`SecOptSP1`)

```
input  : $s^0 = (x_0, y_0, z_0), s^1 = (x_1, y_1, z_1)$
output: $(x'_0 \oplus x'_1, y'_0 \oplus y'_1, z'_0 \oplus z'_1) = SP(x_0 \oplus x_1, y_0 \oplus y_1, z_0 \oplus z_1)$
1  $(x_0, x_1) \leftarrow (x_0 \lll 24, x_1 \lll 24)$
2  $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge z_0, x_0 \vee z_1, x_1 \vee z_0, x_1 \wedge z_1)$
3  $(t_0, t_1) \leftarrow (s_0 \oplus s_1, s_2 \oplus s_3)$
4  $(u_0, u_1) \leftarrow (x_0 \oplus (t_0 \ll 1), x_1 \oplus (t_1 \ll 1))$
5  $(y'_0, y'_1) \leftarrow (u_0 \oplus (y_0 \lll 9), u_1 \oplus (u_1 \lll 9))$
6  $(s_0, s_1, s_2, s_3) \leftarrow (z_0 \wedge (y_0 \lll 9), z_0 \vee \neg(y_1 \lll 9), z_1 \wedge (y_0 \lll 9), z_1 \vee \neg(y_0 \lll 9))$
7  $(t_0, t_1) \leftarrow (s_0 \oplus s_1, s_2 \oplus s_3)$
8  $(u_0, u_1) \leftarrow (x_0 \oplus (t_0 \ll 2), x_1 \oplus (t_1 \ll 2))$
9  $(z'_0, z'_1) \leftarrow (u_0 \oplus (z_0 \ll 1), u_1 \oplus (z_1 \ll 1))$
10 $(s_0, s_1, s_2, s_3) \leftarrow (x_0 \wedge (y_0 \lll 9), x_0 \vee \neg(y_1 \lll 9), x_1 \wedge (y_0 \lll 9), x_1 \vee \neg(y_0 \lll 9))$
11 $(t_0, t_1) \leftarrow (s_0 \oplus s_1, s_2 \oplus s_3)$
12 $(u_0, u_1) \leftarrow (z_0 \oplus (t_0 \ll 3), z_1 \oplus (t_1 \ll 3))$
13 $(x'_0, x'_1) \leftarrow (u_0 \oplus (y_0 \lll 9), u_1 \oplus (y_1 \lll 9))$
```

Algorithm 5: 24 rounds of Gimli permutation

```
input  : $\mathbf{s}^0 = (s^0_{i,j}), \mathbf{s}^1 = (s^1_{i,j}) \in W^{3\times 4}$
output: $\text{GIMLI}(\mathbf{s}^0 \oplus \mathbf{s}^1) = (s^0_{i,j}, s^1_{i,j}) \in W^{3\times 4,2}$
$\mathbf{s}_{0,0}, \mathbf{s}_{1,1}, \mathbf{s}_{2,2}, \mathbf{s}_{3,3} \leftarrow \texttt{SP1}(\mathbf{s}_{0,0}), \texttt{SP1}(\mathbf{s}_{1,1}), \texttt{SP1}(\mathbf{s}_{2,2}), \texttt{SP1}(\mathbf{s}_{3,3})$
$s^0_{1,0} \leftarrow s^0_{1,0} \oplus \texttt{0x9e377900} \oplus 24$
$\mathbf{s}_{1,0}, \mathbf{s}_{0,1}, \mathbf{s}_{3,2}, \mathbf{s}_{2,3} \leftarrow \texttt{SP2}(\mathbf{s}_{1,0}), \texttt{SP2}(\mathbf{s}_{0,1}), \texttt{SP2}(\mathbf{s}_{3,2}), \texttt{SP2}(\mathbf{s}_{2,3})$
$\mathbf{s}_{3,0}, \mathbf{s}_{2,1}, \mathbf{s}_{1,2}, \mathbf{s}_{0,3} \leftarrow \texttt{SP2}(\mathbf{s}_{3,0}), \texttt{SP2}(\mathbf{s}_{2,1}), \texttt{SP2}(\mathbf{s}_{1,2}), \texttt{SP2}(\mathbf{s}_{0,3})$
$s^0_{2,0} \leftarrow s^0_{2,0} \oplus \texttt{0x9e377900} \oplus 20$
$\mathbf{s}_{2,0}, \mathbf{s}_{3,1}, \mathbf{s}_{0,2}, \mathbf{s}_{1,3} \leftarrow \texttt{SP2}(\mathbf{s}_{2,0}), \texttt{SP2}(\mathbf{s}_{3,1}), \texttt{SP2}(\mathbf{s}_{0,2}), \texttt{SP2}(\mathbf{s}_{1,3})$
$\mathbf{s}_{0,0}, \mathbf{s}_{1,1}, \mathbf{s}_{2,2}, \mathbf{s}_{3,3} \leftarrow \texttt{SP2}(\mathbf{s}_{0,0}), \texttt{SP2}(\mathbf{s}_{1,1}), \texttt{SP2}(\mathbf{s}_{2,2}), \texttt{SP2}(\mathbf{s}_{3,3})$
$s^0_{1,0} \leftarrow s^0_{1,0} \oplus \texttt{0x9e377900} \oplus 16$
$\mathbf{s}_{1,0}, \mathbf{s}_{0,1}, \mathbf{s}_{3,2}, \mathbf{s}_{2,3} \leftarrow \texttt{SP2}(\mathbf{s}_{1,0}), \texttt{SP2}(\mathbf{s}_{0,1}), \texttt{SP2}(\mathbf{s}_{3,2}), \texttt{SP2}(\mathbf{s}_{2,3})$
$\mathbf{s}_{3,0}, \mathbf{s}_{2,1}, \mathbf{s}_{1,2}, \mathbf{s}_{0,3} \leftarrow \texttt{SP2}(\mathbf{s}_{3,0}), \texttt{SP2}(\mathbf{s}_{2,1}), \texttt{SP2}(\mathbf{s}_{1,2}), \texttt{SP2}(\mathbf{s}_{0,3})$
$s^0_{2,0} \leftarrow s^0_{2,0} \oplus \texttt{0x9e377900} \oplus 12$
$\mathbf{s}_{2,0}, \mathbf{s}_{3,1}, \mathbf{s}_{0,2}, \mathbf{s}_{1,3} \leftarrow \texttt{SP2}(\mathbf{s}_{2,0}), \texttt{SP2}(\mathbf{s}_{3,1}), \texttt{SP2}(\mathbf{s}_{0,2}), \texttt{SP2}(\mathbf{s}_{1,3})$
$\mathbf{s}_{0,0}, \mathbf{s}_{1,1}, \mathbf{s}_{2,2}, \mathbf{s}_{3,3} \leftarrow \texttt{SP2}(\mathbf{s}_{0,0}), \texttt{SP2}(\mathbf{s}_{1,1}), \texttt{SP2}(\mathbf{s}_{2,2}), \texttt{SP2}(\mathbf{s}_{3,3})$
$s^0_{1,0} \leftarrow s^0_{1,0} \oplus \texttt{0x9e377900} \oplus 8$
$\mathbf{s}_{1,0}, \mathbf{s}_{0,1}, \mathbf{s}_{3,2}, \mathbf{s}_{2,3} \leftarrow \texttt{SP2}(\mathbf{s}_{1,0}), \texttt{SP2}(\mathbf{s}_{0,1}), \texttt{SP2}(\mathbf{s}_{3,2}), \texttt{SP2}(\mathbf{s}_{2,3})$
$\mathbf{s}_{3,0}, \mathbf{s}_{2,1}, \mathbf{s}_{1,2}, \mathbf{s}_{0,3} \leftarrow \texttt{SP2}(\mathbf{s}_{3,0}), \texttt{SP2}(\mathbf{s}_{2,1}), \texttt{SP2}(\mathbf{s}_{1,2}), \texttt{SP2}(\mathbf{s}_{0,3})$
$s^0_{2,0} \leftarrow s^0_{2,0} \oplus \texttt{0x9e377900} \oplus 4$
$\mathbf{s}_{2,0}, \mathbf{s}_{3,1}, \mathbf{s}_{0,2}, \mathbf{s}_{1,3} \leftarrow \texttt{SP2}(\mathbf{s}_{2,0}), \texttt{SP2}(\mathbf{s}_{3,1}), \texttt{SP2}(\mathbf{s}_{0,2}), \texttt{SP2}(\mathbf{s}_{1,3})$
$\mathbf{s}_{0,0}, \mathbf{s}_{1,1}, \mathbf{s}_{2,2}, \mathbf{s}_{3,3} \leftarrow \texttt{SP1}(\mathbf{s}_{0,0}), \texttt{SP1}(\mathbf{s}_{1,1}), \texttt{SP1}(\mathbf{s}_{2,2}), \texttt{SP1}(\mathbf{s}_{3,3})$
return $(\mathbf{s}^0, \mathbf{s}^1)$
```

The details for a 3-share threshold implementation are much the same in Algorithm 6. The input is now 3 shared states $(s^0 = (x_0, y_0, z_0), s^1 = (x_1, y_1, z_1), s^2 = (x_2, y_2, z_2))$ and the output is $SP(x_0 \oplus x_1 \oplus x_2, y_0 \oplus y_1 \oplus y_2, z_0 \oplus$

Algorithm 6: 3-share SP-box

```
    input  : $s^0 = (x_0, y_0, z_0), s^1 = (x_1, y_1, z_1), s^2 = (x_2, y_2, z_2)$
    output: $(x'_0 \oplus x'_1 \oplus x'_2, y'_0 \oplus y'_1 \oplus y'_2, z'_0 \oplus z'_1 \oplus z'_2) =$
            $SP(x_0 \oplus x_1 \oplus x_2, y_0 \oplus y_1 \oplus y_2, z_0 \oplus z_1 \oplus z_2)$
 1  $(x_0, x_1, x_2) \leftarrow (x_0 \lll 24, x_1 \lll 24, x_2 \lll 24)$
 2  $y'_1 \leftarrow ((y_2 \lll 9) \oplus x_2 \oplus (((x_2 \vee z_2) \oplus (\neg x_2 \wedge z_0) \oplus (x_0 \wedge \neg z_2)) \ll 1))$
 3  $z'_1 \leftarrow (z_2 \oplus (y_2 \lll 9) \oplus (((x_2 \wedge (y_2 \lll 9)) \oplus (x_2 \wedge (y_0 \lll 9)) \oplus (x_0 \wedge (y_2 \lll 9))) \ll 3))$
 4  $y'_2 \leftarrow ((y_0 \lll 9) \oplus x_0 \oplus (((x_0 \wedge z_0) \oplus (x_0 \wedge z_1) \oplus (x_1 \wedge z_0)) \ll 1))$
 5  $z'_2 \leftarrow (z_0 \oplus (y_0 \lll 9) \oplus (((x_0 \wedge (y_0 \lll 9)) \oplus (x_0 \wedge (y_1 \lll 9)) \oplus (x_1 \wedge (y_0 \lll 9))) \ll 3))$
 6  $x'_2 \leftarrow (x_0 \oplus (z_0 \ll 1) \oplus ((((y_0 \lll 9) \wedge z_0) \oplus ((y_0 \lll 9) \wedge z_1) \oplus ((y_1 \lll 9) \wedge z_0)) \ll 2))$
 7  $x'_1 \leftarrow (x_2 \oplus (z_2 \ll 1) \oplus ((((y_2 \lll 9) \wedge z_2) \oplus ((y_2 \lll 9) \wedge z_0) \oplus ((y_0 \lll 9) \wedge z_2)) \ll 2))$
 8  $y'_0 \leftarrow ((y_1 \lll 9) \oplus x_1 \oplus (((x_1 \wedge z_1) \oplus (x_1 \wedge \neg z_2) \oplus (\neg x_2 \wedge z_1)) \ll 1))$
 9  $z'_0 \leftarrow (z_1 \oplus (y_1 \lll 9) \oplus (((x_1 \wedge (y_1 \lll 9)) \oplus (x_1 \wedge (y_2 \lll 9)) \oplus (x_2 \wedge (y_1 \lll 9))) \ll 3))$
10  $x'_0 \leftarrow (x_1 \oplus (z_1 \ll 1) \oplus ((((y_1 \lll 9) \wedge z_1) \oplus ((y_1 \lll 9) \wedge z_2) \oplus ((y_2 \lll 9) \wedge z_1)) \ll 2))$
```

$z_1 \oplus z_2)$, where the SP box computes the result of Eq. 3. Then, we follow the same procedure as in Algorithm 5 to construct the 24 rounds of 3-share GIMLI.

The total number of operations for the 3-share threshold implementation is 3(`ROR`) + 36(`EOR`) + 27(`AND/ORR`) = 66, which is about 7 times the non-shared version.

5 Experiments and Results

The software was cross-compiled using the GNU Compiler Collection for ARM Embedded Processors version 9.2.1 with the options `-mthumb -mcpu=cortex-m4` and tested on a STM32F407 discovery board. The length was measured by the number of assembly code instructions, while the cycle count was measured using the internal clock cycle counter. Note that the reported cycles include the overheads for calling/returning from the considered functions, while all input data was assumed to be already word aligned.

5.1 Comparison of the Implementations

Table 2 provides a comparative overview of the implementation results. The cycles are counted from the beginning of AEAD encryption of 1024 bytes associated data and 1024 bytes plaintext, while the lengths are only counted by the assembly code for 24 rounds of GIMLI permutation. In Table 2, the **original** method is pure C implementation [19] and the **non-shared** method represents the assembly implementation in Sect. 4.1.

Our GIMLI implementation is much quicker than the C implementation. Even with 3-share protection, our result is comparable with the original unprotected C implementation. The growth in cycles as the number of shares increases is meeting our prediction as well. The cycle count of the 2-share with optimal masking implementation is about 4 times that of non-masked, and that of the 3-share TI implementation is about 7 times the ones of non-shared. These reference assembly codes can be found online[1]. On the other hand, according to Table 2, a 2-share implementation with ChaCha-8 is slower than a 3-share Threshold implementation. If a 2-share Threshold implementation cannot meet the required security level, a 3-share Threshold implementation is a better choice compared with the 2-share implementation with ChaCha-8.

We also tried implementing only by M3 instructions, and the results are shown in Table 3. The main difference between M3 and M4 is the memory manipulation instructions. Without the floating-point registers as the temporary memory, the cost of store and load instructions increases as the number of shares increases.

Table 4 shows the performance of masked algorithms compared with the baseline unmasked ones. The numbers are how much slower the masked version was than the unmasked version. Our methods have a significant improvement on both scenarios compared with [29], even accounting for the Cortex-M4 to Cortex-M3 difference.

Table 2. The results for Gimli-AEAD (1024 message bytes and 1024 ad bytes) under benchmark clock (24 MHz).

Methods	Cycles	Speed (cycles/byte)	Length
Original	1037161	506.4	–
Non-shared	151551	74.0	987
2-share (chaha-8 randomness)	2213676	1080.9	336 (not unrolled)
2-share (optimal masking)	615383	300.5	4247
3-share (TI)	1159939	566.4	8378

5.2 Leakage Detection of Side-Channel Analysis

We adopt the test vector leakage assessment (TVLA) methodology to perform leakage detection. All the experiments here are based on ChipWhisperer-Lite Two-Part Version [26]. The program ChipWhisperer Capture [27] retrieves power samples from the control board, storing power traces and input data.

To complete the DPA test at Security Level 4 of ISO/IEC 17825 [1], we focus on the first permutation and capture two sets of $l = 100000$ power traces corresponding to the selected plaintexts and randomly plaintexts and compute the Welch's t-test to identify the differentiating features between the trace sets.

Figure 1 shows the T-test of three different versions of our implementation. Each picture can be separated into three parts by the black lines: 1. loading key and plaintext and applying the random mask to the state; 2. the 24-round Gimli permutation; and 3. recovering the state and return. To reduce leakage introduced by pipeline registers, we rearranged the parallel instructions in 2-share TI [13]. Figure 1c shows the t-test results where the parallel instructions

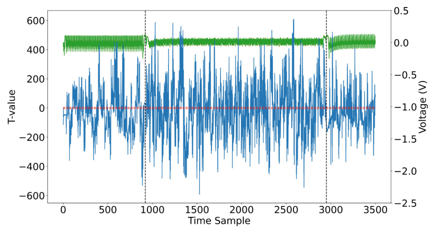
[1] https://github.com/kuruwa2/pqm4/tree/master/gimli24v1-aead.

Table 3. The results for Gimli-AEAD (1024 message bytes and 1024 ad bytes) on M3 under benchmark clock (24 MHz).

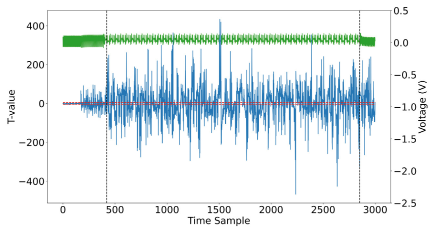
Methods	Cycles	Speed (cycles/byte)	Length
Non-shared	151551	74.0	987
2-share (chaha-8 randomness)	2286875	1116.6	335 (not unrolled)
2-share (optimal masking)	615826	300.7	4250
3-share (TI)	1247448	609.1	8276

Table 4. The amount of overhead for masking.

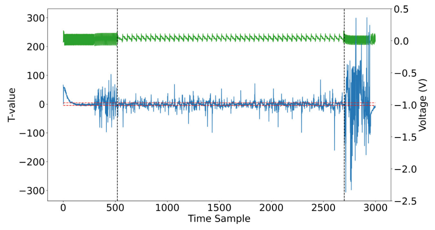
Methods	2 shares	3 shares
Reference [29]	10.50	31.41
Ours	4.06	7.65



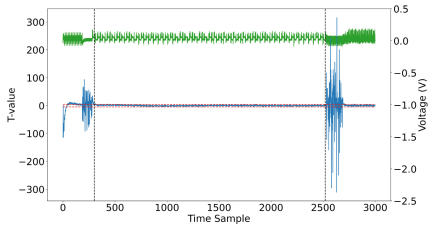
(a) Unprotected Gimli



(b) 2-share without rearrangement



(c) 2-share



(d) 3-share threshold implementation

Fig. 1. (a) T-test of unprotected Gimli (b) T-test of 2-share with optimal masking Gimli without rearrangement (c) T-test of 2-share with optimal masking Gimli (d) T-test of 3-share threshold implementation Gimli

are rearranged. However, 2-share TI still has some unintentional information leakage. We suspect that there are some unexpected buffers producing unintentional leakage in Cortex-M3 and -M4 [4]. On the other hand, We can see that the t-statistic value of the 3-share masked Gimli permutation is inside the ±4.5 [30] interval, corresponding to 99.999% confidence that a difference shown is not due

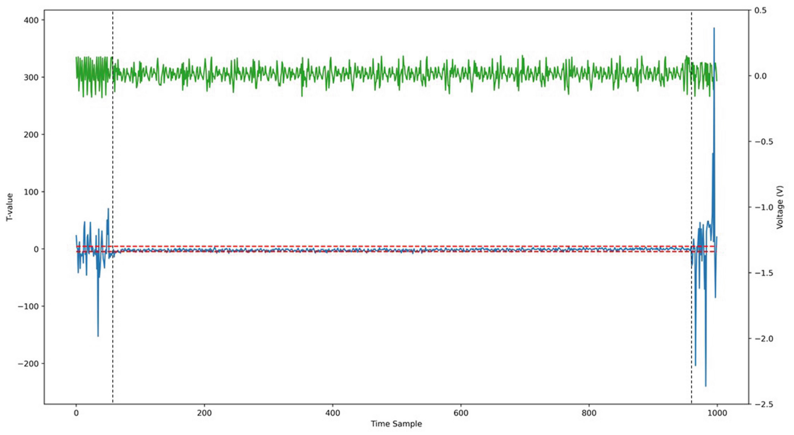


Fig. 2. T-test of 3-share threshold implementation GIMLI with l = 1000000 power traces

to random chance. To detect the leakage of 3-share masked GIMLI permutation, we capture with l = 1,000,000 power traces and use TVLA to perform a leakage assessment. Figure 2 shows the T-test of the first 12 round of 3-share Threshold implementation with l = 1,000,000 power traces. We can see that the t-statistic value is inside the ±4.5 interval. In addition, since the first and third parts contain the public value, such as the ciphertext, it is normal that the t-statistic value exceeds ±4.5 [2].

6 Conclusion

Our results significantly improve the performance of GIMLI implementations on Cortex-M3 and -M4 processors, especially NORX ciphers such as GIMLI are popular for their simplicity in preventing timing attacks. Compared to the original C implementation, the instruction count is reduced by 85%. In addition to this, the overhead of masking is reduced by 61% and 76% for 2-shared and 3-shared, respectively. Even if we rearranged the parallel instructions, the currently widely used 2-shared, GIMLI still exists for unintentional information leakage. Finally, we have completed a 3-share Threshold Implementation and passed the safety inspection of ISO/IEC 17825 Level 4.

References

1. ISO/IEC 17825:2016 information technology - security techniques - testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Standard, International Organization for Standardization, Geneva, CH (2016)
2. Abdulrahman, A., Chen, J.P., Chen, Y.J., Hwang, V., Kannwischer, M.J., Yang, B.Y.: Multi-moduli NTTS for saber on cortex-m3 and cortex-m4. Cryptology ePrint Archive, Report 2021/995 (2021). https://ia.cr/2021/995

3. Aumasson, J.-P., Jovanovic, P., Neves, S.: NORX: parallel and scalable AEAD. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8713, pp. 19–36. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_2
4. Barenghi, A., Pelosi, G.: Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pp. 1–6 (2018)
5. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: Proceedings of the 52nd Annual Design Automation Conference, pp. 1–6 (2015)
6. Bernstein, D.J., et al.: Gimli: a cross-platform permutation. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 299–320. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_15
7. Bernstein, D.J., et al.: Chacha, a variant of salsa20. In: Workshop Record of SASC, vol. 8, pp. 3–5 (2008)
8. Bilgin, B.: Threshold implementations as countermeasure against higher-order differential power analysis (2015)
9. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 326–343. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_18
10. Biryukov, A., Dinu, D., Le Corre, Y., Udovenko, A.: Optimal first-order boolean masking for embedded IoT devices. In: Eisenbarth, T., Teglia, Y. (eds.) CARDIS 2017. LNCS, vol. 10728, pp. 22–41. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75208-2_2
11. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28632-5_2
12. Chen, C., Farmani, M., Eisenbarth, T.: A tale of two shares: why two-share threshold implementation seems worthwhile—and why it is not. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 819–843. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_30
13. Corre, Y.L., Großschädl, J., Dinu, D.: Micro-architectural power simulator for leakage assessment of cryptographic software on arm cortex-m3 processors. Cryptology ePrint Archive, Report 2017/1253 (2017). https://ia.cr/2017/1253
14. Daemen, J.: Changing of the guards: a simple and efficient method for achieving uniformity in threshold sharing. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 137–153. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_7
15. Gruber, M., et al.: DOMREP - an orthogonal countermeasure for arbitrary order side-channel and fault attack protection. IEEE Trans. Inf. Forensics Secur. **16**, 4321–4335 (2021)
16. Gruber, M., Probst, M., Tempelmeier, M.: Statistical ineffective fault analysis of GIMLI. In: 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 252–261 (2020)
17. Gupta, N., Jati, A., Chattopadhyay, A., Sanadhya, S.K., Chang, D.: Threshold implementations of GIFT: a trade-off analysis. Cryptology ePrint Archive, Report 2017/1040 (2017). http://eprint.iacr.org/2017/1040
18. Jungk, B., Petri, R., Stöttinger, M.: Efficient side-channel protections of ARX ciphers. Cryptology ePrint Archive, Report 2018/693 (2018). https://eprint.iacr.org/2018/693

19. Kannwischer, M.J.: m4-crypto-eng-assignments (2020). https://github.com/mkannwischer/m4-crypto-eng-assignments/tree/master/gimli24v1-aead
20. Khan, S., Lee, W.K., Hwang, S.O.: A flexible Gimli hardware implementation in FPGA and its application to RFID authentication protocols. IEEE Access **9**, 105327–105340 (2021)
21. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
22. Miyashita, S., Ito, R., Miyaji, A.: PNB-focused differential cryptanalysis of ChaCha stream cipher. Cryptology ePrint Archive, Report 2021/1537 (2021). https://ia.cr/2021/1537
23. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 529–545. Springer, Heidelberg (2006). https://doi.org/10.1007/11935308_38
24. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of non-linear functions in the presence of glitches. In: Lee, P.J., Cheon, J.H. (eds.) ICISC 2008. LNCS, vol. 5461, pp. 218–234. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00730-9_14
25. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of nonlinear functions in the presence of glitches. J. Cryptol. **24**(2), 292–321 (2011)
26. O'Flynn, C.: Chipwhisperer-lite (cw1173) two-part version (2016)
27. O'Flynn, C.: ChipWhisperer - the complete open-source toolchain for side-channel power analysis and glitching attacks (2018)
28. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 764–783. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_37
29. Weatherley, R.: Performance of masked algorithms. In: Lightweight Cryptography Primitives Documentation (2020). https://rweather.github.io/lightweight-crypto/performance_masking.html
30. Whitnall, C., Oswald, E.: A critical analysis of ISO 17825 ('Testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019. LNCS, vol. 11923, pp. 256–284. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34618-8_9
31. Yan, Y., Oswald, E., Vivek, S.: An analytic attack against ARX addition exploiting standard side-channel leakage. Cryptology ePrint Archive, Paper 2020/1455 (2020). https://eprint.iacr.org/2020/1455. https://eprint.iacr.org/2020/1455