# Optimal First-Order Boolean Masking for Embedded IoT Devices

Alex Biryukov, Daniel Dinu, Yann Le Corre, Aleksei Udovenko

`first-name.last-name@uni.lu`
SnT and CSC, University of Luxembourg

**Abstract.** Boolean masking is an effective side-channel countermeasure that consists in splitting each sensitive variable into two or more shares which are carefully manipulated to avoid leakage of the sensitive variable. The best known expressions for Boolean masking of bitwise operations are relatively compact, but even a small improvement of these expressions can significantly reduce the performance penalty of more complex masked operations such as modular addition on Boolean shares or of masked ciphers. In this paper, we present and evaluate new secure expressions for performing bitwise operations on Boolean shares. To this end, we describe an algorithm for efficient search of expressions that have an optimal cost in number of elementary operations. We show that bitwise AND and OR on Boolean shares can be performed using less instructions than the best known expressions. More importantly, our expressions do no require additional random values as the best known expressions do. We apply our new expressions to the masked addition/subtraction on Boolean shares based on the Kogge-Stone adder and we report an improvement of the execution time between 14% and 19%. Then, we compare the efficiency of first-order masked implementations of three lightweight block ciphers on an ARM Cortex-M3 to determine which design strategies are most suitable for efficient masking. All our masked implementations passed the t-test evaluation and thus are deemed secure against first-order side-channel attacks.

**Keywords:** Boolean masking, side-channel attack, IoT, embedded device

## 1 Introduction

The Internet of Things (IoT) is one of the technical revolutions of our time, with many IoT devices being deployed every day to create a global network of smart objects. According to Gartner, 8.4 billion connected things will be in use worldwide by the end of 2017 [14]. From 2018 onwards, Gartner forecasts that devices such as those targeted at smart buildings (LED lighting, HVAC, and physical security systems) will have the biggest market share [14]. In light of the very recent security vulnerabilities [8, 24] discovered in such devices, immediate action is required to prevent large-scale security incidents similar to the Mirai botnet [1].

The attack surface of IoT devices is considerably larger than the attack surface of classical Internet-connected systems due to the various use cases these gadgets, sensors, and actuators are built for. Most of the IoT systems are characterized by low physical security, with devices being deployed in easily accessible places. As a consequence, attack vectors that exploit these weaknesses came to light. Side-channel attacks, such as EM and power analysis attacks, fall in this category of attack vectors that require physical proximity to the target system. If the target system uses an unprotected implementation of a cryptographic algorithm, the adversary can determine the secret key used by the system from the leakage generated during the execution of the algorithm. Hence, countermeasures against side-channel attacks are mandatory for the security of IoT devices.

There are two main categories of countermeasures against side-channel attacks: masking and hiding [18]. One of the main advantages of masking over hiding is that the security of masking schemes can be proved under certain assumptions on the device leakage model and the attacker capabilities [17]. However, if the masking scheme is not correctly implemented, the implementation can leak and therefore it is not secure against side-channel attacks [4,21].

Boolean masking is one of the most widely used masking schemes. An $(n-1)$-th order Boolean masking scheme with $n \geq 2$ is based on the principle of secret sharing, splitting each variable $x$ into at least $n$ shares $x_i$ such that $x = x_1 \oplus x_2 \oplus \ldots x_n$. Then, the protected algorithm processes the shares $x_i$ in such a way that no information about the sensitive value $x$ can be learned by an adversary which can probe up to $n-1$ wires. Yet, an $(n-1)$-th order masking scheme can be broken with an $n$-th order attack. The complexity of a such an attack grows exponentially with the number of shares since the attacker has to combine $n$ points to reconstruct the leakage of the sensitive variable [7].

There are two main requirements an implementation of a cryptographic algorithm to be deployed in the IoT has to satisfy. On the one hand, the implementation must be *lightweight* (i.e. consume few resources) because of the limited computational resources of embedded devices for the IoT. On the other hand, the implementation must be secure against side-channel attacks given the attack surface specific to the IoT. Most implementations of the existing lightweight ciphers do not satisfy the second requirement, either because the cipher was not designed to facilitate masking, or because the best existing masking schemes add significant performance penalties to the unprotected implementation of the cipher. Therefore, there is a need for more efficient masking schemes. Any improvement of the existing masking schemes brings us closer to the goal of a secure IoT.

Conceptually, Boolean masking of a block cipher is done by replacing each unprotected operation by its masked counterpart. The most common operations used by lightweight block ciphers are logical operations (NOT, AND, OR, XOR), rotations, and modular addition/subtraction. Masked NOT is equivalent to the negation of a single share, while masked XOR and rotations can be realized by simply applying the operation to each pair of shares independently. To our knowledge, the best known expression for first-order Boolean masking of bitwise

AND is based on the Trichina AND gate [28]. The same expression of the masked AND was latter used by Coron *et al.* in their algorithm for masked addition on Boolean shares [9]. Since there is almost no reference to a masked OR expression in the literature, one might try to derive such an expression by applying De Morgan's laws to the masked AND expression. Hence, we consider the derived expression using De Morgan's laws as the best known expression for masked OR, although Baek and Noh [3] proposed a masked OR gate that requires six elementary operations and no random value. The best known masked expressions of AND and OR require an additional random value.

Groß [16] showed how to design and implement a general purpose arithmetic logic unit using provably secure threshold implementations. He used an exhaustive search to find the best expression for efficient masking of AND and OR in hardware using three shares.

The best known algorithm for secure addition on Boolean shares is based on the Kogge-Stone adder [9]. Won and Han [29] presented a method to improve the execution time of this algorithm when the register size of the target microcontroller is smaller than the operand size. Schneider *et al.* described efficient hardware modules that perform addition on Boolean shares [26].

In this paper, we study the efficiency of Boolean masking for embedded IoT devices. Although our work is not limited to a specific microprocessor architecture, we evaluate our implementations on a 32-bit ARM Cortex-M3 since these microcontrollers are widely used for IoT applications [22].

**Our Contributions.** Firstly, we present an algorithm for efficient search of Boolean masking expressions (Section 2). Thanks to several algorithmic optimizations, the search is very fast. As a second contribution, we propose concrete expressions for Boolean masking of the AND and OR operations (Section 2.2). Our expressions use fewer elementary operations than the best known expressions in the literature. At the same time, unlike the best known expressions, our expressions for secure AND and OR on Boolean shares do no require any randomness. Thirdly, we improve the Kogge-Stone algorithm for addition/subtraction on Boolean shares [9] by using our masking expressions and by processing the shares in a clever way that does not require any randomness (Section 3.1). When implemented on an ARM Cortex-M3 processor (Section 4.1), the addition/subtraction of 32-bit values using the new algorithm is between 14% and 19% faster than similar implementations using the original algorithm [9]. Finally, we use our Boolean masking expressions to write first-order masked implementations of three lightweight block ciphers, namely SIMON, SPECK, and RECTANGLE (Section 4.2). By comparing the performance figures of the masked and unmasked implementations of the three ciphers, we learn which design strategies facilitate efficient masked implementations.

All software presented in this paper will is placed in the public domain [1] to support reproducibility of results and to maximize reusability.

[1] `https://github.com/cryptolu/ofom`

## 2 Search Algorithm

In this section we describe our algorithm for searching optimal masking expressions. We start with a high-level description, then we dive into details. We give a pseudocode of the full search algorithm. Finally, we provide the optimal expressions we found using the algorithm.

### 2.1 The Algorithm

The algorithm takes as input a set of variables representing the input shares, a set of sensitive functions (i.e. functions that combine the shares of a sensitive value and thus leak the sensitive value) and a target function; it outputs the shortest sequences of operations required to compute the Boolean shares of the target function. A *sequence* is represented as a tuple that contains all intermediate terms of an expression in the order they are required to compute the expression. Moreover, any single intermediate value computed in a sequence does not leak any information about the sensitive functions. Multiple intermediate values may be considered for higher-order masking.

At its core, the algorithm performs a breadth-first search with several cut-off conditions. The functions are represented by their truth tables and are stored as integers for efficiency reasons. Initially, there is only an empty sequence available. The algorithm expands it into multiple sequences of length one. Afterwards, all sequences of length one are expanded into sequences of length two and so on, until the algorithm finds a sequence for which some of its intermediate values are the Boolean shares of the target function. The search is illustrated in Fig. 1.

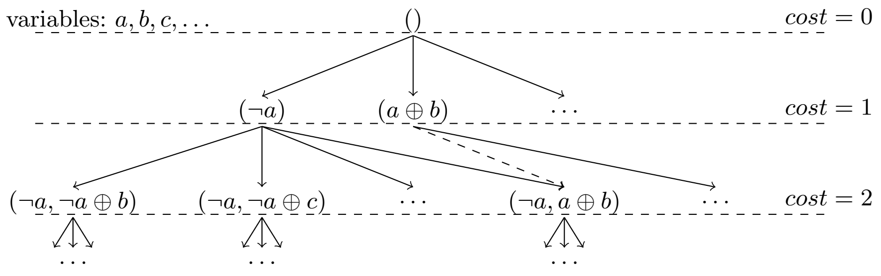

Fig. 1: High-level scheme of the search algorithm.

The core function of the algorithm is the extension step, where a given sequence is extended with one operation using all possible combinations to get the *extended sequences*. The new operation may take as its inputs either variables of the input shares or intermediate values computed in the current sequence. The cut-off conditions used to reduce the complexity of the algorithm are described next.

1. **Leakage test.** For each new function, the algorithm checks if the new function leaks information about the sensitive functions. If the function is leaking, the extended sequence is omitted and not considered anymore. In this way, the search space is effectively reduced only to non-leaking functions. This significantly improves the efficiency of the search algorithm and guarantees that the resulting sequences do not leak.
   The check is very efficient since it consists of 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

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

   where $\mathsf{HW}(g)$ denotes the Hamming weight of the truth table of function $g$.
2. **Ignoring the order of operations.** For any sequence of operations, we exclude all other sequences that compute the same *set* of intermediate functions. Indeed, such sequences are *equivalent* in terms of extension, because an extension depends only on the *set* of intermediate functions but not on the way they are computed. From each such equivalence class we keep the representative that the algorithm reaches first. Note that this condition also excludes sequences that compute some function multiple times.
   Due to this cut-off, we may miss some optimal sequences. More precisely, from each such equivalence class we will preserve only one representative sequence. Since we do not allow to compute the same function twice in a sequence, the representative will have the shortest length. Hence, the algorithm will find at least one sequence of optimal length. If all optimal sequences are required, the full equivalence class can be recovered from its representative.
3. **Exploiting the symmetries of shares.** The Boolean shares are naturally symmetric: permuting the shares of a masked value does not change the masked value. Moreover, when we are masking a symmetric operation (e.g. AND, OR), swapping the input operands in the whole circuit will still give a correct circuit. We can exploit these symmetries and explore only one of the equivalent sequences, similarly to the cut-off condition 2. Again, the same reasoning shows that we do not miss an optimal sequence.

The pseudocode of the search algorithm is given in Algorithm 1. For simplicity and efficiency reasons, the algorithm keeps track only of computed functions but not of the applied operations. After the optimal function sequences are found, it is easy to recover the corresponding expressions. This approach also reduces the memory usage of the algorithm.

**Optimality.** We would like to stress that the algorithm is designed to find *optimal* expressions, not just to improve the existing ones. It is easy to show that the algorithm yields optimal expressions when it reaches the optimal cost level. Any optimal expression has (at least one) non-leaking sequence of operations to compute it. The algorithm explores all sequences except those omitted during

**Algorithm 1** Searching for the Optimal Shares

```
Require:
    target function $t : \mathbb{F}_2^n \to \mathbb{F}_2$;    ▷ e.g. $t(x_0, x_1, x_2, x_3) = (x_0 \oplus x_1) \wedge (x_2 \oplus x_3)$
    number of output shares $m$;
    set of sensitive functions $K = \{k_i\}, k_i : \mathbb{F}_2^n \to \mathbb{F}_2$;    ▷ e.g. $(x_0 \oplus x_1), (x_2 \oplus x_3), t$
    set of allowed operations $O = \{op_i\}, op_i : \mathbb{F}_2^2 \to \mathbb{F}_2$
Ensure:
    set of $m$ functions $S = \{s_i\}, s_i : \mathbb{F}_2^n \to \mathbb{F}_2$ such that $\bigoplus_{s_i \in S} s_i = t$;
    optimal circuits computing all $s_i$ without leaking information about the value of
    any $k_i$;
 1: $seqs_0 \leftarrow \{()\}$    ▷ empty sequence
 2: $visited \leftarrow \{()\}$
 3: for $cost := 1$ to $\infty$ do
 4:     $seqs_{cost} \leftarrow \{\}$
 5:     for all $seq \in seqs_{cost-1}$ do
 6:         for all $seq' \in$ EXTENSIONS($seq$) do
 7:             if SHOULDKEEPSEQUENCE($seq'$) then
 8:                 $seqs_{cost} \leftarrow seqs_{cost} \cup \{seq'\}$
 9:                 if CONTAINSSHARES($seq'$, $t$) then    ▷ impl. omitted for brevity
10:                     yield $seq'$
11:                 end if
12:             end if
13:             $visited \leftarrow visited \cup \{set(seq')\}$
14:         end for
15:     end for
16: end for
17: function EXTENSIONS($seq$)
18:     for all $a, b \in seq \cup \{x_0, x_1, \ldots, x_{n-1}\}$ do
19:         for all $op \in O$ do
20:             yield $seq || op(a, b)$
21:         end for
22:     end for
23: end function
24: function SHOULDKEEPSEQUENCE($seq$)
25:     if LEAKS($last(seq), K$) then    ▷ Cut-off 1
26:         return False
27:     end if
28:     $seq \leftarrow$ SYMMETRYREPRESENTATIVE($seq$) ▷ impl. omitted for brevity; Cut-off 3
29:     if $set(seq) \in visited$ then    ▷ Cut-off 2, 3
30:         return False
31:     end if
32:     return True
33: end function
34: function LEAKS($f, K$)
35:     for all $k \in K$ do
36:         if $\mathsf{HW}(k \wedge \neg f)\mathsf{HW}(f) \neq \mathsf{HW}(k \wedge f)\mathsf{HW}(\neg f)$ then ▷ fraction equality check
37:             return True
38:         end if
39:     end for
40:     return False
41: end function
```

cut-offs. The first cut-off condition reduces the search to non-leaking sequences. It is easy to see that the effect of the other two cut-off conditions can be jointly seen as collapsing large equivalence classes into single representatives. Due to the breadth-first nature of the algorithm, the representative chosen by the algorithm has minimum cost.

Note that it is important to search for sequences of operations instead of expressions. Expressions may contain repeating terms and this reduces the computational cost. Moreover, this effect spreads over the output shares as well: they also may have common terms. Because of this effect, it is unclear how long are the expressions one has to consider to find a provably optimal expression. Searching for sequences of operations solves this problem at the cost of increasing the search space. The described cut-off conditions aim to narrow this gap and bring the algorithm to feasible complexities.

**Instruction Set Architecture (ISA).** We distinguish between two classes of IoT devices depending on the operations supported by the instruction set architecture (ISA): *basic* and *enhanced* devices. Most IoT devices have instructions only for the following bitwise logical operations: NOT, AND, OR, and XOR. We call these architectures *basic* ISAs. In addition to these operations, the *enhanced* ISAs have dedicated instructions for other bitwise logical operations, such as AND NOT or OR NOT. For example, the instruction set of ARM Cortex-M3 includes the `bic` (AND NOT) and `orn` (OR NOT) instructions that perform two basic bitwise logical operations in a single clock cycle instead of two clock cycles. Most microcontrollers execute all logical instructions in a single clock cycle.

**Leakage Model.** The power consumption of most microcontrollers is proportional to the number of bits that are set in the processed sensitive value [18]. Therefore, the Hamming weight power model is a reliable method for modeling the leakage of a sensitive variable. In addition to the bit-level leakage verification performed by the search algorithm, we performed a t-test leakage assessment [15] for each valid expression returned by the algorithm to confirm the absence of any leakage.

**Extension to Higher-Order Masking.** Our algorithm can naturally be extended to search expressions for higher-order masking. However, further optimizations might be required to ensure that the algorithm scales well for higher values of the number of shares.

## 2.2 Results

We have implemented the algorithm in Python language and ran it using the fast PyPy interpreter [11]. We searched for expressions for masked AND ($\mathsf{SecAnd}$) and masked OR ($\mathsf{SecOr}$). For example, to search for masked AND on a basic platform we used the following inputs to the algorithm:

Table 1: Expressions, number of randoms (Rand) and number of operations (Cost) for different secure operations. Basic cost gives the number of elementary operations, while the ARM cost gives the number of instructions. Expressions in parentheses have priority and operations are executed from left to right.

| Source | Operation | Expression | Rand | Cost | |
|---|---|---|---|---|---|
| | | | | Basic | ARM |
| best known | SecAnd | $z_1 = r$<br>$z_2 = z_1 \oplus (x_1 \wedge y_1) \oplus (x_1 \wedge y_2) \oplus$ $(x_2 \wedge y_1) \oplus (x_2 \wedge y_2)$ | 1 | 8 | 8 |
| | SecOr | $z_1 = r$<br>$z_2 = \neg z_1 \oplus (x_1 \wedge y_1) \oplus (x_1 \wedge \neg y_2) \oplus$ $(\neg x_2 \wedge y_1) \oplus (\neg x_2 \wedge \neg y_2)$ | 1 | 11 | 10 |
| our | SecAnd | $z_1 = (x_1 \wedge y_1) \oplus (x_1 \vee \neg y_2)$<br>$z_2 = (x_2 \wedge y_1) \oplus (x_2 \vee \neg y_2)$ | 0 | 7 | 6 |
| | SecOr | $z_1 = (x_1 \wedge y_1) \oplus (x_1 \vee y_2)$<br>$z_2 = (x_2 \vee y_1) \oplus (x_2 \wedge y_2)$ | 0 | 6 | 6 |

1. target function $t(x_0, x_1, x_2, x_3) = (x_0 \oplus x_1) \wedge (x_2 \oplus x_3)$;
2. number of output shares $m = 2$;
3. set of sensitive functions $K = \{s_0, s_1, s_0 \wedge s_1, \neg s_0 \wedge s_1, s_0 \wedge \neg s_1, \neg s_0 \wedge \neg s_1, \}$, where $s_0 = x_0 \oplus x_1, s_1 = x_2 \oplus x_3$;
4. set of allowed operations $O = \{\wedge, \vee, \oplus, \neg\}$.

The hardest target was the search for SecAnd limited to 6 enhanced ISA operations which took 30 minutes and 10 GB RAM on a laptop. The optimal expressions for masked SecOr use 6 instructions on both platforms, while optimal expressions for SecAnd have a cost of 7 on a basic device and 6 on ARM.

The optimal expressions for SecOr and SecAnd using basic instructions are unique up to symmetries of the shares, whereas for ARM there are 48 different optimal expressions for SecAnd and 50 different optimal expressions for SecOr. The unique optimal expressions for a basic architecture are actually included in the optimal expressions for the ARM architecture, which makes them universal. A comparison of these two expressions with the best known expressions in the literature is given in Table 1. Besides using less operations than the best known expressions in the literature, our optimal expressions do not require a random value. Thanks to these two properties, our expressions have a significant performance advantage over the best known ones.

## 3 Applications

### 3.1 Modular Addition and Subtraction

Coron *et al.* [9] proposed a logarithmic-time algorithm for modular addition on Boolean shares based on the Kogge-Stone adder. Their algorithm for modular addition uses the following three secure operations: SecAnd, SecXor, and SecShift.

The expression of SecAnd uses 8 elementary operations, the one of SecXor needs 2 elementary operations, while SecShift can be performed using 4 elementary operations. Algorithms for all these operations are presented in [9].

Although not described in the original paper [9], the algorithm for modular subtraction can be obtained from the algorithm for modular addition on Boolean shares by making several changes. Namely, the SecShift operations from lines 7 and 15 of [9, Algorithm 6] have to be replaced by SecShiftFill (secure operation for shift to the left by $n$ followed by OR of $2^n - 1$). Similarly, SecXor operations from lines 9 and 17 of [9, Algorithm 6] must be replaced by SecOr. These changes affect the performance of the modular subtraction algorithm since operations with a lower cost are replaced by operations with a higher cost.

**Algorithm 2** Improved Kogge-Stone Masked Addition

```
Require: $x_1, x_2, y_1, y_2 \in \{0,1\}^k$ such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z_1, z_2$ such that $z = z_1 \oplus z_2 = (x + y) \bmod 2^k$
 1: $p_1, p_2 \leftarrow \mathsf{SecXor}(x_1, x_2, y_1, y_2)$
 2: $g_1, g_2 \leftarrow \mathsf{SecAnd}(x_1, x_2, y_1, y_2)$
 3: $g_1, g_2 \leftarrow \big((g_1 \oplus x_2) \oplus g_2, x_2\big)$
 4: $n \leftarrow \max\big(\lceil \log_2(k-1) \rceil, 1\big)$
 5: for $i := 1$ to $n - 1$ do
 6:     $h_1, h_2 \leftarrow \mathsf{SecShift}(g_1, g_2, 2^{i-1})$
 7:     $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
 8:     $g_1, g_2 \leftarrow \mathsf{SecXor}(g_1, g_2, u_1, u_2)$
 9:     $h_1, h_2 \leftarrow \mathsf{SecShift}(p_1, p_2, 2^{i-1})$
10:     $h_1, h_2 \leftarrow \big((h_1 \oplus x_2) \oplus h_2, x_2\big)$
11:     $p_1, p_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
12:     $p_1, p_2 \leftarrow \big((p_1 \oplus y_2) \oplus p_2, y_2\big)$
13: end for
14: $h_1, h_2 \leftarrow \mathsf{SecShift}(g_1, g_2, 2^{n-1})$
15: $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
16: $g_1, g_2 \leftarrow \mathsf{SecXor}(g_1, g_2, u_1, u_2)$
17: $z_1, z_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
18: $z_1, z_2 \leftarrow \big(\big(z_1 \oplus (g_1 \ll 1)\big) \oplus (x_2 \ll 1), y_2\big)$
```

One can improve the algorithms for modular addition/subtraction based on the Kogge-Stone adder by simply replacing the original expressions for SecAnd and SecOr with our optimal expressions. Yet, the algorithm can be improved further by replacing the expression of the SecShift operation, which requires a random variable, by a more efficient expression that does not require any randomness. Hence, the new versions of the algorithm do not require any randomness at all. The improved algorithm for addition on Boolean shares is described in Algorithm 2, while the analogous algorithm for subtraction is presented in Algorithm 3. It is important to note that lines 3, 10, and 12 from Algorithm 2 are required to prevent composition of operations that otherwise will leak. Similarly, lines 4, 10, 12, 14, and 19 of Algorithm 3 avoid composing operations that leak.

**Algorithm 3** Improved Kogge-Stone Masked Subtraction

```
Require: $x_1, x_2, y_1, y_2 \in \{0,1\}^k$ such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$
Ensure: $z_1, z_2$ such that $z = z_1 \oplus z_2 = (x - y) \bmod 2^k$
 1: $y_1, y_2 \leftarrow \mathsf{SecNot}(y_1, y_2)$
 2: $p_1, p_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
 3: $g_1, g_2 \leftarrow \mathsf{SecAnd}(x_1, x_2, y_1, y_2)$
 4: $g_1, g_2 \leftarrow \big((g_1 \oplus x_2) \oplus g_2, x_2\big)$
 5: $n \leftarrow \max\big(\lceil \log_2(k-1) \rceil, 1\big)$
 6: for $i := 1$ to $n - 1$ do
 7:     $h_1, h_2 \leftarrow \mathsf{SecShiftFill}(g_1, g_2, 2^{i-1})$
 8:     $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
 9:     $g_1, g_2 \leftarrow \mathsf{SecOr}(g_1, g_2, u_1, u_2)$
10:     $g_1, g_2 \leftarrow \big((g_1 \oplus x_2) \oplus g_2, x_2\big)$
11:     $h_1, h_2 \leftarrow \mathsf{SecShift}(p_1, p_2, 2^{i-1})$
12:     $h_1, h_2 \leftarrow \big((h_1 \oplus x_2) \oplus h_2, x_2\big)$
13:     $p_1, p_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
14:     $p_1, p_2 \leftarrow \big((p_1 \oplus y_2) \oplus p_2, y_2\big)$
15: end for
16: $h_1, h_2 \leftarrow \mathsf{SecShiftFill}(g_1, g_2, 2^{n-1})$
17: $u_1, u_2 \leftarrow \mathsf{SecAnd}(p_1, p_2, h_1, h_2)$
18: $g_1, g_2 \leftarrow \mathsf{SecOr}(g_1, g_2, u_1, u_2)$
19: $g_1, g_2 \leftarrow \big((g_1 \oplus x_2) \oplus g_2, x_2\big)$
20: $z_1, z_2 \leftarrow \mathsf{SecXor}(y_1, y_2, x_1, x_2)$
21: $z_1 \leftarrow \Big(z_1 \oplus \big((g_1 \ll 1) \vee 1\big)\Big) \oplus (x_2 \ll 1)$
22: $z_2 \leftarrow y_2$
```

**Cost.** A comparison between the cost of the secure expressions used by the original version of the algorithm and the new expressions used by the improved version of the algorithm is provided in Table 2. Based on these values, one can compute the total cost of these algorithms for different architectures and make an estimation of their performance for different values of the operand size $k$ (see Table 3).

**Security.** We evaluated the secure operations presented in this section, including the two improved algorithms for addition and subtraction on Boolean shares, against first-order attacks using Welch's t-test [15]. Welch's t-test is a fast and robust way to verify the soundness of a masking scheme [12, 25]. To determine if there is any leakage in our first-order implementations, we used a simple tool similar to the ones described in [20, 21, 23]. Firstly, we validated the correctness of our tool by performing evaluations against a set of masking schemes known

Table 2: Comparison of the number of instructions required to perform different secure operations.

| Platform | Source | Cost | | | | | |
|---|---|---|---|---|---|---|---|
| | | SecNot | SecXor | SecAnd | SecOr | SecShift | SecShiftFill |
| Basic | best known | 1 | 2 | 8 | 11 | 4 | 6 |
| | our | 1 | 2 | 7 | 6 | 2 | 4 |
| | **gain** | **0** | **0** | **1** | **5** | **2** | **2** |
| ARM | best known | 1 | 2 | 8 | 10 | 4 | 6 |
| | our | 1 | 2 | 6 | 6 | 2 | 4 |
| | **gain** | **0** | **0** | **2** | **4** | **2** | **2** |

Table 3: Cost and random numbers (Rand) required for Kogge-Stone addition/subtraction on Boolean shares for different values of the operand size $k$. Basic cost gives the number of elementary operations, while the ARM cost gives the number of instructions.

| Operation | Platform | Expressions | Rand | $k$ | $k=8$ | $k=16$ | $k=32$ | $k=64$ |
|---|---|---|---|---|---|---|---|---|
| SecAdd | Basic | best known | 2 | $28 \cdot \log_2 k + 4$ | 88 | 116 | 144 | 172 |
| | | our | 0 | $22 \cdot \log_2 k + 6$ | 72 | 94 | 116 | 138 |
| | | **gain** | **2** | $6 \cdot \log_2 k - 2$ | **16** | **22** | **28** | **34** |
| | ARM | best known | 2 | $28 \cdot \log_2 k + 4$ | 88 | 116 | 144 | 172 |
| | | our | 0 | $22 \cdot \log_2 k + 4$ | 70 | 92 | 114 | 136 |
| | | **gain** | **2** | $6 \cdot \log_2 k$ | **18** | **24** | **30** | **36** |
| SecSub | Basic | best known | 2 | $41 \cdot \log_2 k + 4$ | 127 | 168 | 209 | 250 |
| | | our | 0 | $32 \cdot \log_2 k + 6$ | 102 | 134 | 166 | 198 |
| | | **gain** | **2** | $9 \cdot \log_2 k - 2$ | **25** | **34** | **43** | **52** |
| | ARM | best known | 2 | $40 \cdot \log_2 k + 4$ | 124 | 164 | 204 | 244 |
| | | our | 0 | $30 \cdot \log_2 k + 6$ | 96 | 126 | 156 | 186 |
| | | **gain** | **2** | $10 \cdot \log_2 k - 2$ | **28** | **38** | **48** | **58** |

to be either secure or broken. Then, we carefully applied the t-test to avoid false negatives [27]. All our secure implementations passed a set of *fixed vs. random* evaluations with up to $10^6$ traces using both Hamming weight and Hamming distance models for the simulated leakage. See Appendix A for more details.

## 3.2 Other Applications

The optimal expressions for secure computation of AND and OR can be used to mask more complex structures such as S-boxes. They can also be used to efficiently mask ciphers that use only logical bitwise operations such as Simon [6], as well as bit-sliced designs such as Noekeon [10], RECTANGLE [30], or RoadRunneR [5]. In Section 4, we evaluate how these expressions can be applied to unprotected implementations of several lightweight block ciphers and we determine the performance penalty of the resulting first-order protected implementations.

## 4 Implementations

In this section we describe our efficient implementations of several first-order secure algorithms and block ciphers. All our implementations are written in assembly language for a Cortex-M3 processor for two reasons. Firstly, we wanted to avoid accidental leakages introduced by the transformations made by the `gcc` compiler which is not optimized for masked implementations, but only for efficiency [4]. On the other hand, when coding in assembly language, the implementer has full control of the register allocation and the sequence of instructions executed by the microcontroller. Hence, she can avoid combining instructions and registers in a way that leaks [4, 21]. Secondly, we wanted to get a clear picture of the performance figures of our implementations in order to conduct a fair comparison of the first-order implementations. Hence, the effort spent by a programmer on a more demanding assembly implementation is paid off in the end by a better (i.e. more secure and efficient) implementation.

In line with previous work, we do not include the cost of random number generation for the implementations that need randomness since the cost of random number generation is different from one device to the other and we want a device-independent comparison. We report the execution time and the code size for protected implementations that do not leak in the Hamming weight model. The leakage of these implementations in the Hamming distance model can be fixed with minor changes. These changes have a similar effect on the performance of the implementations based on our expressions and the implementations based on the best known expression.

### 4.1 Masked Addition

We implemented the original algorithms for addition and subtraction on Boolean shares as well as the improved algorithms presented in this paper. For each algorithm we wrote a straightforward implementation and an implementation that unrolls the main loop of the Kogge-Stone adder. The execution time and code size of our implementations are given in Table 4.

The improved algorithms are between 14% and 19% faster than the original ones. At the same time, the code size of the improved algorithms is between 12% and 21% smaller than the code size of the original algorithms. Unlike the original algorithms, which require two random values, the improved algorithms do not require any random value. The generation of a 32-bit random number takes between 37 cycles for a XorShift RNG [19] and 85 cycles for the built-in TRNG [2]. Hence, the improved algorithms for addition and subtraction on Boolean shares outperform the original algorithms in all categories: execution time, code size, and required randomness.

### 4.2 Lightweight Block Ciphers

We selected the top-3 block ciphers that use a 64-bit block from the performance evaluation conducted using the FELICS benchmarking framework [13] and we

Table 4: Execution time and code size for secure addition and subtraction on Boolean shares using the Kogge-Stone adder.

| Impl. | Expressions | Rand | Time (cycles) | | Code size (bytes) | |
|---|---|---|---|---|---|---|
| | | | Addition | Subtraction | Addition | Subtraction |
| rolled | best known | 2 | 275 | 388 | 292 | 416 |
| | our | 0 | 228 | 333 | 232 | 332 |
| | **gain** | 2 | **47 (17%)** | **55 (14%)** | **60 (21%)** | **84 (20%)** |
| unrolled | best known | 2 | 203 | 296 | 544 | 812 |
| | our | 0 | 173 | 241 | 480 | 692 |
| | **gain** | 2 | **30 (15%)** | **55 (19%)** | **64 (12%)** | **120 (15%)** |

protected them against first-order attacks using the best known algorithms for secure operations on Boolean shares as well as the ones introduced in this paper. Besides their very lightweight software implementations, these three ciphers (Speck, Simon, and RECTANGLE) have different design strategies. Hence, they facilitate an analysis of the relationship between their design strategies and the performance figures of their masked implementations.

**Speck.** Speck [6] is an ARX-based family of lightweight block ciphers designed for performance in software. Nevertheless, all ciphers of this family perform very well in hardware also. Speck-64/128 refers to the version of Speck characterized by a 64-bit block, a 128-bit key, and 27 rounds. The round function of Speck-64/128 uses only bitwise XOR, addition modulo $2^{32}$, and rotations:

$$R_k(x, y) = \Big( \big((x \ggg 8) \boxplus y\big) \oplus k, (y \lll 3) \oplus \big((x \ggg 8) \boxplus y\big) \oplus k \Big),$$

where $x$ and $y$ are the two 32-bit branches of a Feistel network.

While the unprotected implementation of Speck requires only four registers in order to process the cipher's state, the protected implementations need all 13 general-purpose registers of the Cortex-M3 microcontroller. Moreover, the rolled implementations have to save the content of a register onto the stack at the beginning of the secure addition/subtraction. The initial value of this register is recovered at the end of the addition/subtraction operation. A pair of stack operations (i.e. `push` and `pop`) adds 4 cycles to the total execution time of the algorithm.

The implementations of Speck based on the improved algorithms for modular addition and subtraction on Boolean shares are faster and use less code space than the implementations of Speck based on the original versions of the same algorithms as can be seen in Table 5. When comparing the gain of the improved expressions over the original ones for rolled and unrolled implementations, we can see that the gain in the case of rolled implementations is higher than the gain in the case of unrolled ones. For example, the gain of rolled decryption is 27%, while the gain of unrolled decryption is only 17%.

Table 5: Execution time, code size and performance penalty factor for different secure implementations of SPECK-64/128. For each set of expressions (best known, our) we wrote two implementations that correspond to the two implementation strategies of the Kogge-Stone adder (KSA): rolled/unrolled KSA.

| Impl./Expr. | Rand | Time (cycles) | | Code size (bytes) | | Penalty factor | |
|---|---|---|---|---|---|---|---|
| | | Enc | Dec | Enc | Dec | Enc | Dec |
| unprotected | 0 | 318 | 530 | 44 | 52 | 1 | 1 |
| rolled KSA/best known | 2 | 7131 | 11368 | 340 | 488 | 22.42 | 21.44 |
| rolled KSA/our | 0 | 5686 | 8258 | 272 | 400 | 17.88 | 15.58 |
| **gain** | 2 | **1445** | **3110** | **68** | **88** | **4.54** | **5.86** |
| **%** | | **21%** | **27%** | **20%** | **18%** | | |
| unrolled KSA/best known | 2 | 4945 | 7431 | 588 | 876 | 15.55 | 14.02 |
| unrolled KSA/our | 0 | 4666 | 6188 | 536 | 712 | 14.67 | 11.67 |
| **gain** | 2 | **279** | **1243** | **52** | **164** | **0.87** | **2.34** |
| **%** | | **6%** | **17%** | **9%** | **19%** | | |

**Simon.** SIMON [6] is a family of lightweight block ciphers designed primarily for optimal performance in hardware, but its instances perform very good in software as well. The round function of SIMON uses only bitwise XOR, bitwise AND, and rotations:

$$R_k(x, y) = \big(y \oplus f(x) \oplus k, x\big),$$

where $f(x) = (x \lll 1) \wedge (x \lll 8) \oplus (x \lll 2)$. SIMON-64/128 is the instance of SIMON that processes a 64-bit block using a 128-bit key in 44 rounds.

The two protected implementations of SIMON are very efficient since the operations used by the cipher can be masked with a little impact on the execution time and code size. The most costly operation is secure bitwise AND which, depending on its expression, can be evaluated using 6 or 8 instructions. The other secure operations require only 2 instructions each. The unprotected implementation of SIMON needs only four registers. The first-order protected implementation based on the best known expression of AND requires ten registers, while the one based on our optimal expression of AND takes nine registers. Consequently, the gain in execution time of the implementation based on the improved expression of AND over the implementation based on the best known expression of AND is modest (i.e. 5%). Nevertheless, the gain in code size is about 25%. The results of these implementations is presented in Table 6.

**RECTANGLE.** RECTANGLE [30] is a block cipher designed to facilitate lightweight and fast implementations, both in hardware and software, using bit-slicing. RECTANGLE processes a 64-bit block in 25 rounds and supports keys of 80 and 128 bits. We refer to the 128-bit version of RECTANGLE as RECTANGLE-64/128. The cipher's state is represented as a matrix of $4 \times 16$ bits. Each round of RECTANGLE uses three transformations: `AddRoundKey` (bitwise XOR), `SubColumn` (application of a 4-bit S-box to the state columns), and

Table 6: Execution time, code size and performance penalty factor for different secure implementations of SIMON-64/128.

| Impl./Expr. | Rand | Time (cycles) | | Code size (bytes) | | Penalty factor | |
|---|---|---|---|---|---|---|---|
| | | Enc | Dec | Enc | Dec | Enc | Dec |
| unprotected | 0 | 1068 | 1069 | 60 | 64 | 1 | 1 |
| best known | 1 | 1736 | 1737 | 152 | 156 | 1.62 | 1.62 |
| our | 0 | 1648 | 1649 | 136 | 140 | 1.54 | 1.54 |
| **gain** | 1 | **88 (5%)** | **88 (5%)** | **16 (27%)** | **16 (25%)** | **0.08** | **0.08** |

Table 7: Execution time, code size and performance penalty factor for different secure implementations of RECTANGLE-64/128.

| Impl./Expr. | Rand | Time (cycles) | | Code size (bytes) | | Penalty factor | |
|---|---|---|---|---|---|---|---|
| | | Enc | Dec | Enc | Dec | Enc | Dec |
| unprotected | 0 | 945 | 994 | 200 | 160 | 1 | 1 |
| best known | 1 | 3661 | 3422 | 632 | 444 | 3.87 | 3.44 |
| our | 0 | 2584 | 2954 | 564 | 372 | 2.73 | 2.97 |
| **gain** | 1 | **1077 (19%)** | **468 (14%)** | **68 (11%)** | **72 (16%)** | **1.13** | **0.47** |

`ShiftRow` (rotations of the state rows by 1, 12 and 13 bits). The S-box of RECTANGLE can be described using a sequence of 12 basic logical instructions and hence the `SubColumn` transformation can be implemented in a bit-sliced fashion.

The unprotected implementation of RECTANGLE requires seven registers for encryption and eight for decryption. The protected implementations use all available registers of the microcontroller and several pairs of stack operations (i.e. `push` and `pop`). The protected implementation based on the best known expressions uses five pairs of stack operations, while the one based on our optimal expressions uses only three pairs for encryption and four pairs for decryption. The stack operations are necessary because the protected implementations have to keep track of more intermediate variables than they can fit into the registers of the ARM microcontroller.

In summary, the implementation based on our optimal expressions uses less instructions and less stack operations compared to the implementation based on the best known expression. The performance figures given in Table 7 show that the gain in execution time is 19% for encryption and 14% for decryption.

**Comparison.** When comparing the performance results of the unprotected implementations of the three ciphers (see Fig. 2), one can see that SPECK is the fastest, followed by RECTANGLE and SIMON; each of them takes about three times more cycles than SPECK. On the other hand, when comparing first-order protected implementations, the implementations of SIMON and RECTANGLE take the lead, while the implementation of SPECK is the last one. The performance degradation of the first-order protected implementation of SPECK stems from the high overhead associated with masking modular addition (see Table 4).

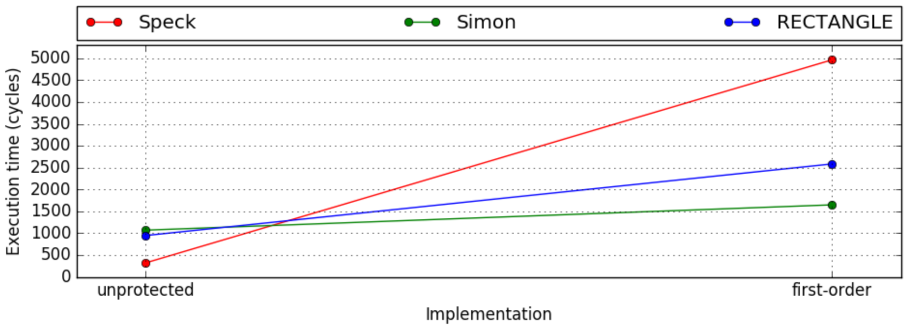

Fig. 2: Performance comparison of unprotected and first-order protected implementations of Speck, Simon, and RECTANGLE.

The protected implementation of RECTANGLE is roughly three times slower than its unprotected implementation. Finally, the protected implementation of Simon is only 54% slower than its unprotected implementation.

From this analysis, we learn that lightweight block ciphers that are very fast in unprotected software implementations (e.g. Speck), might not be the most suitable ones for first-order masking in software. A second key remark is that a cipher that uses only bitwise operations can have an efficient first-order masked implementation only if it has a small number of intermediate variables.

**Discussion.** Our implementations explored how far one can push the optimization level in Boolean masking of various algorithms and ciphers. Consequently, we lost the benefit of being able to provide strong security proofs for our implementations. In other words, one can insert a random value in our expressions for masked AND and OR and they will still be a little bit more efficient than the best known ones, but provably secure. On the other hand, if one removes the randomness from the best know expressions for masked AND and OR, they will leak. We kept the amount of randomness at a minimum level (i.e. one or two random values for algorithms using the best known expressions and no random for our expressions). In these settings, the composition problem (i.e. chaining basic secure operations in an unsecure way) is similar for algorithms and ciphers masked using the previously best known expressions and our expressions. Finally, we stress that we put a similar effort in all our implementations.

## 5 Conclusion

We described an efficient algorithm for searching of optimal Boolean masking expressions. Then, we proposed optimal expressions for the first-order masking of bitwise AND and OR. They require less elementary operations and no random values compared to the best known expressions in the literature. Based on these optimal expressions, we presented an improved version of the algorithm for modular addition on Boolean shares proposed by Coron *et al.* [9]. We implemented the original and improved algorithms for modular addition/subtraction of 32-bit values on an ARM Cortex-M3. Our results show that the improved algorithm is between 14% and 19% faster than the original algorithm of Coron *et al.* [9]. Finally, we used our optimal Boolean masking expressions to write first-order protected implementations of three lightweight block ciphers, namely SIMON, SPECK, and RECTANGLE. The evaluation of these implementations revealed that ciphers with a simple structure, based solely on bitwise logical operations and rotations, facilitate efficient software implementations of first-order masking.

## 6 Acknowledgements

The authors thank the anonymous reviewers for their valuable comments. The work of Aleksei Udovenko is supported by the *Fonds National de la Recherche*, Luxembourg (project reference 9037104).

## References

1. M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.
2. Random number generator (TRNG) API. `https://forum.arduino.cc/index.php?topic=129083.0`, October 2012. Accessed: 2017-07-03.
3. Y.-J. Baek and M.-J. Noh. Differential power attack and masking method. *Trends in Mathematics*, 8(1):1–15, June 2005.
4. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert. On the cost of lazy engineering for masked software implementations. In M. Joye and A. Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.
5. A. Baysal and S. Sahin. Roadrunner: A small and fast bitslice block cipher for low cost 8-bit processors. In T. Güneysu, G. Leander, and A. Moradi, editors, *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany, September 10-11, 2015, Revised Selected Papers*, volume 9542 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2015.
6. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015.
7. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara,*

*California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

8. L. Constantin. Hackers found 47 new vulnerabilities in 23 IoT devices at DEF CON. `http://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html`, September 2016. Accessed: 2017-07-03.
9. J. Coron, J. Großschädl, M. Tibouchi, and P. K. Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In G. Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.
10. J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen. Nessie proposal: Noekeon. In *First Open NESSIE Workshop*, pages 213–230, 2000.
11. T. P. Developers. *PyPy interpreter, version 5.1.2*, 2016. `https://pypy.org/`.
12. A. A. Ding, C. Chen, and T. Eisenbarth. Simpler, faster, and more robust t-test based leakage detection. In F. Standaert and E. Oswald, editors, *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, volume 9689 of *Lecture Notes in Computer Science*, pages 163–183. Springer, 2016.
13. D. Dinu, Y. L. Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov. Triathlon of Lightweight Block Ciphers for the Internet of Things. *IACR Cryptology ePrint Archive*, 2015:209, 2015.
14. Gartner. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. `http://www.gartner.com/newsroom/id/3598917`, February 2017. Accessed: 2017-07-03.
15. B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST Non-invasive attack testing workshop*, 2011.
16. H. Groß. Sharing is caring - on the protection of arithmetic logic units against passive physical attacks. In S. Mangard and P. Schaumont, editors, *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015, New York, NY, USA, June 23-24, 2015, Revised Selected Papers*, volume 9440 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2015.
17. Y. Ishai, A. Sahai, and D. A. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.
18. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007.
19. G. Marsaglia et al. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.
20. D. McCann, E. Oswald, and C. Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 199–216. USENIX Association, 2017.
21. K. Papagiannopoulos and N. Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In S. Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

22. Public Comments Received on "Profiles for the Lightweight Cryptography Standardization Process". `https://www.nist.gov/sites/default/files/documents/2017/06/20/public-comments-profiles-i-ii-june2017.pdf`, June 2017. Accessed: 2017-07-03.
23. O. Reparaz. Detecting flawed masking schemes with leakage detection tests. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2016.
24. E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 195–212. IEEE Computer Society, 2017.
25. T. Schneider and A. Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.
26. T. Schneider, A. Moradi, and T. Güneysu. Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware. In T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 559–578. Springer, 2015.
27. F.-X. Standaert. How (not) to use Welch's t-test in side-channel security evaluations. Cryptology ePrint Archive, Report 2017/138, 2017. `http://eprint.iacr.org/2017/138`.
28. E. Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.
29. Y. Won and D. Han. Efficient conversion method from arithmetic to boolean masking in constrained devices. *IACR Cryptology ePrint Archive*, 2016:664, 2016.
30. W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede. RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. *SCIENCE CHINA Information Sciences*, 58(12):1–15, 2015.

## A Leakage Assessment

The tool we used to assess the security of our implementations against first-order attacks is inspired from similar tools such as ELMO [20], ASCOLD [21], and the one described in [23]. The simulated leakages are computed as follows. For each register $r_i$ we store its previous value $r_i^{j-1}$ and its current value $r_i^j$. At each step $j$ we dump the leakage as $\mathsf{HW}(r_i^j)$ or $\mathsf{HD}(r_i^{j-1}, r_i^j) = \mathsf{HW}(r_i^{j-1} \oplus r_i^j)$, where $\mathsf{HW}(\mathsf{r})$ is the Hamming weight of $r$.

The result of the t-test applied to $10^6$ simulated traces (using the $\mathsf{HW}$ model) from our first-order protected implementation of Speck is exemplarily shown in Fig. 3. Similar results for Simon and RECTANGLE are given in Fig. 4 and Fig. 5, respectively. All results use our expressions to compute secure AND and OR. We can see that the value of the t-statistic is inside the $\pm 4.5$ interval for each point in time, which implies that the protected implementations are secure against first-order attacks.

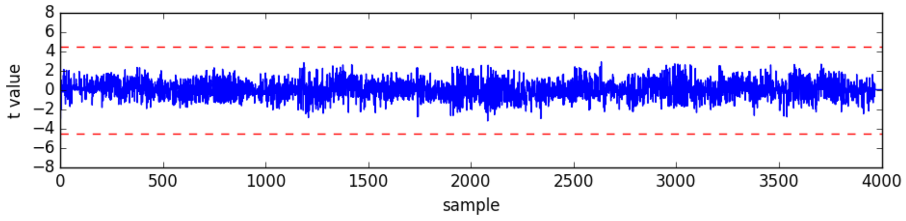

Fig. 3: The result of the t-test applied to our implementation of Speck.

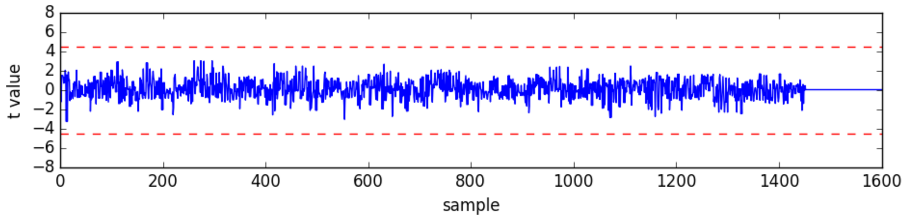

Fig. 4: The result of the t-test applied to our implementation of Simon

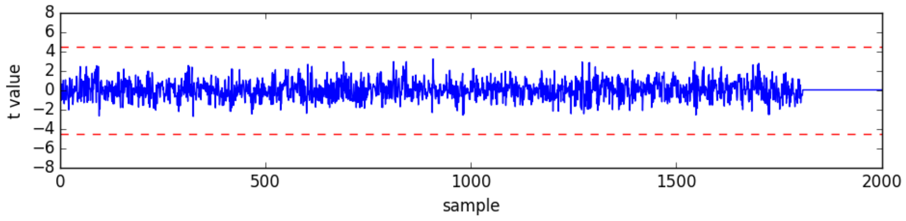

Fig. 5: The result of the t-test applied to our implementation of RECTANGLE.