Efficient Low-Latency Masking of Ascon without Fresh Randomness

Srinidhi Hari Prasad
Infineon Technologies, Germany

Florian Mendel
Infineon Technologies, Germany

Martin Schläffer
Infineon Technologies, Germany

Rishub Nagpal
Graz University of Technology, Austria

Abstract

In this work, we present the first low-latency, second-order masked hardware implementation of Ascon that requires no fresh randomness using only $d+1$ shares. Our results significantly outperform any publicly known second-order masked implementations of AES and Ascon in terms of combined area, latency and randomness requirements. Ascon is a family of lightweight authenticated encryption and hashing schemes selected by NIST for standardization. Ascon is tailored for small form factors. It requires less power and energy while attaining the same or even better performance than current NIST standards.

We achieve the reduction of latency by rearranging the linear layers of the Ascon permutation in a round-based implementation. We provide an improved technique to achieve implementations without the need for fresh randomness. It is based on the concept of changing of the guards extended to the second-order case. Together with the reduction of latency, we need to consider a large set of additional conditions which we propose to solve using a SAT solver.

We have formally verified both, our first- and second-order implementations of Ascon using CocoAlma for the first two rounds. Additionally, we have performed a leakage assessment using t-tests on all 12 rounds of the initial permutation. Finally, we provide a comparison of our second-order masked Ascon implementation with other results.

1 Introduction

Motivation. In 1999, Paul Kocher published differential power analysis (DPA) that allows to extract the secret key of a block cipher from a device's physical properties, if the device is not properly protected [38]. Many attempts have been made to provide techniques to counteract such attacks. For symmetric ciphers, a typical countermeasure is Boolean masking, which splits computations on the secret into several shares [11, 26]. While Boolean masking is relatively easy for linear (Boolean) functions, it is not straight-forward for the non-linear parts of a cipher. This is especially the case for hardware implementations of high-degree S-boxes like in the AES [17]. The low-degree S-boxes of SHA-3 [5] or Ascon [21] seem to be easier to mask.

Several attempts have been made to protect non-linear functions in hardware. Many have failed in practice or have high costs in terms of area, latency or fresh randomness. Threshold implementations (TI) [43] is the first approach to provide a general framework but requires more shares and is difficult to generalize for higher orders [47]. While it is possible to mask non-linear functions using the minimum number of shares [48], this has the drawback of high randomness requirements. The high randomness requirements can be drastically reduced using the changing of the guards technique [15] in the first order case. However, a generalization to the higher-order case seems difficult since the randomness requirements typically increases quadratic with the number of shares.

Domain-oriented masking (DOM) is a seemingly easy approach to mask simple non-linear functions like AND gates using $d+1$ shares [29] for an arbitrary number of shares. However, implementing DOM for a complete cipher can be difficult and several flaws have been shown in practice. This ranges from violating the independence assumption [1] to insufficient refreshing, especially in the higher-order case [40].

Implementations of masking schemes like TI and DOM introduce additional latency and pose a requirement for fresh randomness every cycle to securely operate on the shares. For instance, a DOM AND-gate would require $d(d+1)/2$ fresh random bits where d is the order of masking [29]. Generation of such fresh randomness requires dedicated RNG circuits which consume additional chip area and power, resulting in a large overhead, particularly for higher-order implementations.

Related work. Over the last couple of years, significant research has been conducted in the area of optimizing fresh randomness requirements for various masked implementations. One line of research is to define low-level gadgets and optimize or reuse the randomness among them while maintaining the security of the masked circuit [24, 25, 35, 36, 50].

Ideally, this would result in a generic method to convert an unprotected implementation into a side-channel robust implementation with the desired order of masking. The main drawback of such generic methods is usually a higher cost in terms of area, latency or the required number of shares.

Another study [41] aims to reduce latency and improve performance by introducing a generic approach for designing single-cycle, glitch-resistant hardware. This is done by incorporating a partial dual-rail encoding of masked signals that replace register stages. The introduction of this technique provides synchronization within the circuit. While this work optimized the latency and reduces the randomness requirements, the area requirements are still quite high.

On the other hand, several state-of-the-art publications try to directly reduce the latency and randomness requirements for complete encryption schemes. One approach by [27] provides a generic concept for low-latency masking in hardware and successfully demonstrates its application to Ascon. With this approach, it becomes feasible to eliminate the need for additional register stages which reduces the latency. However, this approach exponentially increases the number of required shares, randomness and domains when consecutive non-linear layers are used.

Another idea is to minimize the latency of a masked cipher by unrolling several rounds of an implementation. A first-order SCA-protected implementation of Keccak with two unrolled rounds has been published in [1]. The authors propose a method using TI in unrolled implementations without violating the non-completeness property. This method increases the required number of shares to 6, still resulting in high area costs.

In the case of AES, a first-order robust AES implementation using TI and changing of the guards has been shown in [54]. This implementation requires no fresh randomness at the cost of high latency and using 4 shares. Similarly, [52] show a first-order protected 3-share TI of the AES S-box without fresh randomness. A second-order protected 4-share AES implementation with low-randomness requirements has been published in [6]. Recently, the second-order low-randomness implementation of AES has been improved to 3-shares in [19]. All these masked AES implementations have a high latency due to the high-degree S-box.

Our Contribution. In this work, we present first- and second-order protected implementations of Ascon [21] using $d+1$ shares, with two cycles of latency per round and zero online randomness. Most notably, these results significantly outperform any publicly known second-order masked implementations of AES and Ascon in terms of combined area, latency and randomness requirements. The technique is generic but requires a dedicated adaptation for each cipher. It works best for algorithms containing S-boxes of low algebraic degree. Our technique allows to efficiently higher-order mask complete ciphers with $d+1$ shares and zero online randomness, while maintaining a low-latency implementation.

We start from a $d+1$ masked hardware implementation of Ascon with three cycles of latency and $d(d+1)/2$ bits of randomness per round [46]. This implementation requires 320 bits of randomness for the first-order and 960 bits of randomness for the second order case per cycle. Using our technique, we are able to reduce the online randomness to zero and the latency to two cycles in both, the first- and second-order case, with minimal area overhead. We have verified two rounds of the first- and second-order implementations of Ascon using the formal side-channel verification tool CocoAlma [30]. Additionally, we have performed a leakage assessment of our implementations using uni- and bivariate t-tests. We plan to publish the source code of our implementations and the scripts to obtain the randomness assignment.

To achieve these results, we use a first- and second-order probing secure AND gate based on domain-oriented masking (DOM). Our main idea is to extend the changing of the guards technique and systematically reuse randomness to refresh the shares in the non-linear functions. In a DOM AND gate, we need to take care that the fresh randomness is independent of its masked inputs. However, especially in the higher-order case bits from neighboring S-boxes may not be independent anymore. We show that the linear layer plays a crucial role in reusing the randomness and provide a method based on SAT solvers to find valid assignments.

Our findings on reducing the randomness requirements have versatile applications. With some modifications, this technique might also be used in algorithms like SHA-3, Xoodyak, but also AES. Furthermore, it can be seamlessly incorporated into the single-cycle glitch-resistant masked hardware of cryptographic S-boxes, which was previously introduced in [41]. This integration can lead to significant gains by reducing the fresh randomness requirements to zero for the implementations.

2 Background

2.1 Side-channel Analysis

Side-channel analysis (SCA) targets the device specific implementation of a cryptographic algorithm, rather than the algorithm itself. An attacker aims to obtain information about the secret data being processed by monitoring various side-channels such as power consumption, timing, or electromagnetic radiation.

It was first shown by [37], that differences in execution time can be exploited to successfully retrieve RSA keys. This was later followed by [38] who demonstrated that differential power analysis reveals even more information about secret keys by exploiting the correlation between multiple power traces.

Numerous research works, such as those mentioned in [44], [31], [33], [13] have demonstrated how inexpensive and highly effective these attacks are in compromising the security of cryptographic implementations. In order to protect the implementation against such side-channel attacks, countermeasures such as masking [26], shuffling [32], and random delay insertion [12] have been proposed.

2.2 Masking

One fundamental countermeasure against DPA attacks is masking. In principle, the idea is to make the intermediate data independent of any sensitive information that is being processed. The most common masking schemes are Boolean masking and arithmetic masking. Boolean masking uses an XOR operation over a binary field in contrast to arithmetic masking, which utilizes addition or multiplication in a modular ring. Using Boolean masking, d-th order security can be achieved by splitting the secret data into $s = d + 1$ shares using an XOR operation over a binary field:

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_s$$

Ideally, each share is then processed individually throughout the computation in a device. The fundamental principle of domain oriented masking (DOM) introduced by [29] is based on shared domains. Here, the objective is to keep the shares of each domain separate from the shares of other domains. For instance, a DOM implementation with $d+1$ shares for each variable will result in $d+1$ domains and should provide d^{th}-order security. However, it is essential to note that implementing linear and non-linear operations in this setup is different. While it is relatively easy to separate domains in linear operations, non-linear operations require special considerations. To handle terms crossing the domains, additional fresh random shares and register stages are needed to prevent glitches from crossing the domains.

The authors present two types of multipliers called *DOM-indep* and *DOM-dep*. The *DOM-indep* multiplier operates on independently shared inputs, which has the advantage of requiring less fresh randomness and has a smaller size. The *DOM-dep* multiplier does not require the inputs to be shared independently but is more costly to implement. Moreover, the authors of [40] observed a specific vulnerability of the *DOM-dep* multiplier. They found out that the *DOM-dep* multiplier does not offer adequate protection against probing attacks for masking orders of two or higher.

Within the scope of our work, we only consider the *DOM-indep* multiplier. Therefore, we need to make sure that the input shares are independently shared in our implementation. We focus on the 1-bit *DOM-indep* multiplier with two inputs, realizing a masked AND operation or AND gate. This *DOM-indep* AND gate consists of the DOM calculation phase, resharing phase, and integration phase, that generates the final shared outputs in $d+1$ domains. The resharing phase requires $d(d+1)/2$ fresh random bits, at least $d(d+1)$ registers and adds additional latency due to the register stage. The 2nd-order

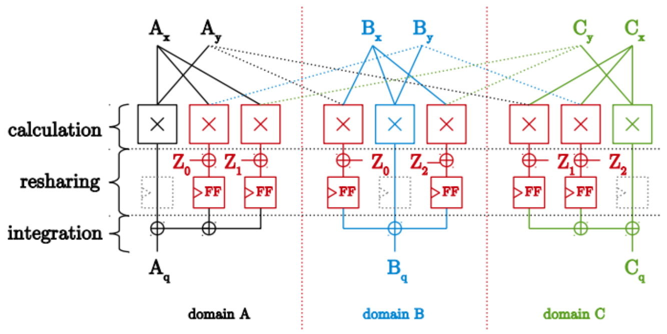


Figure 1: 2nd-order DOM-indep multiplier [29]

DOM-indep multiplier is shown on Figure 1. It is evident that the cost of implementing masking significantly increases due to this processing of shares. For the sake of simplicity, from this point forward we will use the term "DOM AND gate" to refer to the "DOM-indep AND gate".

2.3 Changing of the Guards

Changing of the Guards [15] is a technique that suggests using shares of neighboring S-boxes as additional randomness to arrive at a correct, incomplete, and uniform sharing in a Threshold Implementation (TI) [43]. The authors demonstrate the successful implementation of this concept on the TI masked implementation of Keccak. The method has also been effective in implementing first-order TI of KETJE, a CAESAR finalist, without the need for fresh randomness [3]. Additionally, it has been used to realize a first-order robust TI masked implementation of the AES with zero randomness per-round [54].

Changing of the guards works very well for first-order implementations. Extending it to second- or higher-order masked implementations has not been extensively investigated. One reason is the need for $d(d+1)/2$ additional randomness, which cannot easily be provided using only shares of neighboring S-boxes. In our work, we will discuss challenges and methodologies for extending the concept of *Changing of the Guards* to second-order masked implementations.

2.4 Formal Side-channel Verification

Formal side-channel verification techniques offer several advantages over traditional simulation-based verification approaches. They thoroughly examine all possible leakage scenarios using mathematical models to find potential vulnerabilities or, alternatively provide proof of security often based on the (robust) d-probing models [23, 34]. In addition, they take into consideration hardware effects such as glitches. This makes them particularly suitable for scenarios where hardware designs change frequently and require frequent verification, as they outperform traditional methods that use statistical analysis of leakage traces to identify vulnerabilities, which can be time-consuming and may not cover all scenarios.

Despite these advantages, formal side-channel analysis tools have a significant drawback in the form of complexity issue that arises during the verification of larger designs. Ongoing research is focused on developing formal side-channel verification tools that cover all security properties and still manage to verify reasonably large designs. VerMI [2], MaskVerif [4], REBECCA [9], CocoAlma [30] are some of the available state-of-the-art formal side-channel verification tools. Within the premise of this paper, we use CocoAlma to verify our implementations.

2.5 Ascon

Ascon is a family of authenticated encryption and hashing schemes [21, 22]. Ascon was selected as a winner in the CAESAR competition [14] and was recently chosen by NIST for standardization [42]. While Ascon was designed to enable efficient SCA-resistant implementations in hardware and software, implementing such countermeasures typically requires a lot of area, fresh randomness and increases latency affecting the overall performance.

All Ascon schemes use the same 320-bit permutation, only parameterized by a different umber of rounds. In our masked implementations of Ascon, we focus on the permutation and more specifically, on the round operations of the permutations. In Ascon, the 320-bit state S is split into five 64-bit registers words x_i, which we refer to throughout the paper:.

$$S = x_0 \mid\mid x_1 \mid\mid x_2 \mid\mid x_3 \mid\mid x_4.$$

The permutations p^a and p^b apply the round transformation p iteratively for a or b rounds. Each round consists of a round constant addition (p_C), a substitution layer (p_S), and a linear diffusion layer (p_L). In the constant addition, the round constant c_r is xored to the register word x_2 of the state. The specific value of the round constant depends on the round i of p^a and p^b.

The substitution layer p_S is the only non-linear component of the round transformation. It updates the state S with a parallel application of 64 5-bit S-boxes. The S-box is constructed by applying a lightweight linear transformations to the input and an affine layer to the output of the χ mapping of Keccak [5]. It operates on those five bits with the same bit position in the five state words. The S-box is illustrated in Figure 2, together with the three mentioned components *S-box linear layer*, non-linear χ *mapping* of Keccak and *S-box affine layer*. The S-box has an algebraic degree of 2, a linear and differential branch number of 3 and can be implemented efficiently in hardware and software.

The linear diffusion layer p_L operates on 64-bit register word x_i and applies the linear function $\Sigma_i(x_i)$ to each state word. It rotates each register x_i by fixed rotation constants and

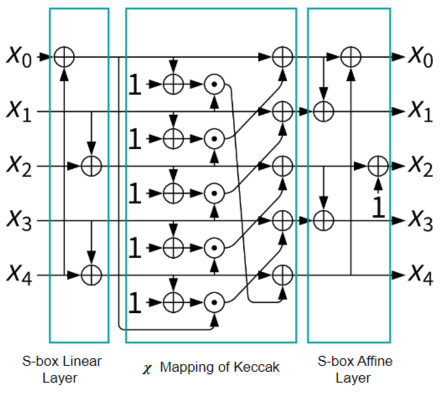


Figure 2: Ascon's 5-bit S-box S(x) with different components highlighted [21]

xors the results to the input register. The linear layer provides a high diffusion within state words and is defined as follows:

$$\begin{aligned}
x_0 &= \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &= \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
x_2 &= \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)\\
x_3 &= \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &= \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}$$

3 Minimizing the Latency in Masked Implementations

The latency of a cipher depends on the number of registers stages needed. While plain implementations can be implemented even within one cycle, this is in general not the case for masked implementations. Although it is possible to minimize the latency at a high cost in area (number of shares) or high randomness requirements, this is not the goal of our work. In our case, we use the DOM-indep multiplier or DOM AND gate, which requires at least one register stage for each function of algebraic degree 2.

Additionally, the inputs of the DOM-indep multiplier need to be independent. This is in general not the case, since the inputs are mixed by the linear functions preceeding the DOM-indep multipliers. Especially in high-diffusion linear layers like in Ascon, inputs cannot be considered independent anymore. To prevent data-dependent glitches propagating into the DOM-indep multiplier, we require an additional register stage at its input.

To summarize, we would need two register stages considering the permutation structure of Ascon. However, we propose to use just one register stage within the permutation structure and hence reduce the latency, while also improving the randomness requirements. In the following, we apply this technique to the cipher Ascon. However, the proposed method has the significant advantage of being generic and can be employed in other ciphers as well. It works best for lightweight ciphers with low algebraic degree such as ISAP [20], Xoodyak [16] or Keccak-based designs, but can also be applied to AES.

3.1 Masking the Ascon S-box

The Ascon S-box is designed by adding a lightweight linear and affine transformations to the input and the output of the χ mapping of Keccak. These transformations are illustrated in Figure 2 as *S-box linear layer* and *S-box affine layer*. These linear transformations can simply be masked by duplicating the operation for each share.

The χ *mapping* xors each bit with a non-linear AND of two other input bits. This 3-bit function is given by $x_i + (x_{i+1} + 1) \cdot x_{i+2}$. This is basically an AND-XOR operation with one inverted input bit. In order to apply masking to the χ *mapping*, we use the DOM AND gate to mask the non-linear operation. Additionally, we include the XOR operation, to get a DOM AND-XOR function.

The structure of the DOM AND-XOR gate to mask one component of $x_i + (x_{i+1} + 1) \cdot x_{i+2}$ is shown in Figure 3. Here, the shares of x_{i+1} and x_{i+2} are multiplied with each other. The corresponding shares of x_i are added to the inner-domain terms prior to the DOM register stage. This is highlighted in green colour in the Figure 3.

For the sake of simplicity, the figure shows the first-order DOM scheme applied to the AND-XOR operation. The same approach can be applied to the second-order case as well. This masking approach results in a functionally correct operation of the χ *mapping* as the XORs are only moved via the linear DOM integration layer into the resharing layer.

This approach offers several advantages. It reduces the depth of the logic between the register stages, as well as minimizes the glitch propagation. Including the XOR into the resharing layer has the advantage of reducing the randomness, which we will show in the subsequent sections.

3.2 Rearranging Linear Functions

To mask the linear diffusion layer, we essentially duplicate the XOR operations for each share, which allows us to mask a single round of permutation once the Ascon S-box has been masked.

We propose a method of reducing latency by rearranging the linear layers among different rounds of the Ascon permutation. For the sake of simplicity, let's consider an implementation of Ascon [46] with three cycles of latency for one round of permutation as shown in Figure 4a. The reason for the three cycles of latency is that one round of the permutation requires three register stages.

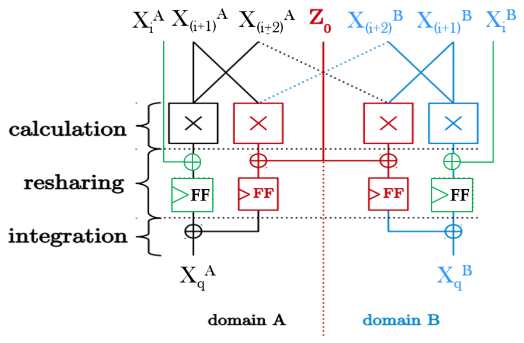


Figure 3: DOM AND gate structure to mask the AND-XOR operation based on [29]

As shown in Figure 4a, we see a state register to register the output of multiple rounds of permutation, register stage 1 is required to provide independence of the inputs to the DOM AND gates, and register stage 2 comes from the DOM AND gate implementation itself. As mentioned before, the register stages 1 and 2 are crucial for the security of the masking implementation and thus cannot be avoided.

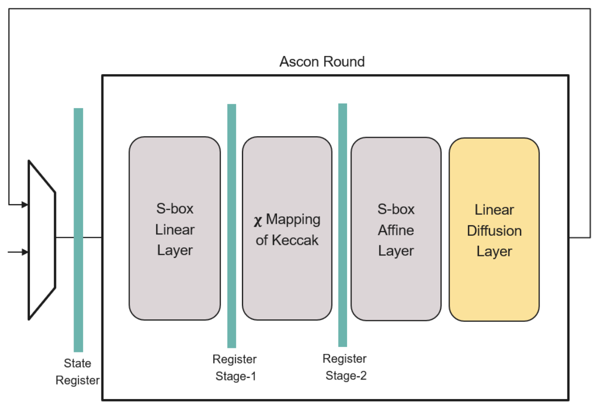
In order to reduce latency, we restructure the round, as shown in Figure 4b. Here we reorder the linear layers by moving the *S-box affine layer* after the *linear diffusion layer* to obtain the new low-latency permutation. The resulting permutation layer just has two register stages: register stage 2 for the DOM implementation and the stage register. This restructuring effectively reduces the latency by one cycle.

It is worth noting that the optimization technique proposed in [46] can also be applied here by clocking one of the register stages in the permutation with a falling-edge clock. By using this technique in our reduced latency implementation, we can achieve a 1-cycle latency for a single round of the permutation.

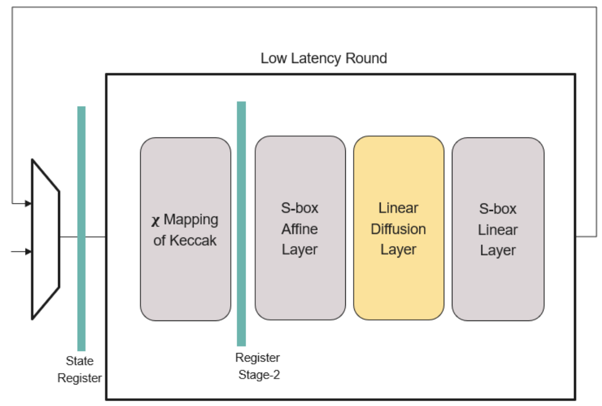
4 Reducing Fresh Randomness Requirements

To reduce PRNG/TRNG circuit costs and energy consumption associated with hardware, it is essential to decrease randomness requirements for constraint devices. Although various methods have been explored extensively to achieve first-order robust masked implementations, second-order scenarios still pose a challenge, with no apparent easy solutions.

While *changing of the guards* offers a generic way to select guard offsets from neighboring S-boxes, this method may not be suitable for all permutation structures, such as in Ascon. This is because the *linear diffusion layer* in Ascon combines the outputs of different S-boxes and subsequent neighboring S-boxes may not be independent anymore. Any generic method for selecting guards could compromise the security of the



(a) 3 cycles per round



(b) 2 cycles per round

Figure 4: Round-based implementation of Ascon.

masked implementation. As a result, choosing guards must be done carefully, and it may present a significantly greater challenge for higher-order scenarios.

This section delves into the difficulties of choosing guards for first and second-order cases in Ascon permutation and outlines the criteria for selecting guards. The goal is to achieve masked implementations of Ascon that do not require fresh randomness.

Selection of the guards. The *changing of the guards* technique proposes to select the guards from neighbouring S-boxes. When selecting guards, it's important to keep in mind that they are the input shares of different S-boxes. In our particular implementation of Ascon, there are 64 S-boxes implemented in parallel and each S-box has 5 DOM AND gates requiring 5 guards.

We define two types of offsets to refer to the position from where we select the guard. The first one is the *column-offset* (c) where we select the S-box position from where we want to use the guard. The next one is the *row-offset* (r) where we select which of the 5 bits (x_i) of the S-box at position c will be used as the guard. For example, for every AND gate i in an S-box j generating the result for x_{ij}, we use $x_{i+r,j+c}$ as the guard. Here, c and r are the *column* and *row-offsets*, respectively, with $0 \leq j \leq 63$ and $0 \leq i \leq 5$. Traditional guard selection would simply use $c = 1$ and $r = 0$.

After defining the two types of offsets, we proceed to establish rules for selecting them for the first and second-order cases. Our goal is to define rules and select offsets while keeping dth-order probing security in mind [23]. This entails picking guards from S-boxes that do not interact with each other.

Due to Ascon's rotation symmetry, the guard selection is also rotation symmetric regarding columns. We can simplify the selection process by formulating and applying *column-* and *row-offset* conditions for one column. Note that this is not the case for rows since the linear layer differs for each row.

4.1 First-order Masked Implementation without Fresh Randomness

The first-order implementation of the DOM AND gate only requires a single fresh random bit. This means that we require only one guard with one *row* and *column-offset* per AND gate. It is intriguing to explore how we can choose these offsets by only considering the Ascon S-box itself. However, we also need to consider the impact of the reorganized Ascon linear layer we have implemented to minimize the latency.

To achieve first-order probing security, we must select guard positions in a way that avoids information leakage due to glitches in the linear layer as well. With this in mind, we simplify the offset selection process by fixing the *row-offset* to 0. In a second step, we determine the conditions for selecting the *column-offset* of the guard. It's essential to note that if we choose different *row-offsets* instead of 0, the conditions for *column-offsets* and corresponding solutions would differ.

Conditions to obtain guard column-offsets. Let R_i be the set containing the two rotation constants r_{i0} and r_{i1} of row x_i. Let G be the set containing the guard *column-offsets*.

First, we must avoid using rotation offsets as guard *column-offsets*. This is because the linear diffusion layer rotates x_i using the rotation constants r_{i1} and r_{i2} and xors them together. Therefore, we need to avoid positions that combine in the linear diffusion layer of Ascon. To fulfill this rule, the following condition applies for R_i where $i \in \{0, 1, 2, 3, 4\}$ and $r_{ij} \in R_i$ where $j \in \{0, 1\}$.

$$\forall g \in G, \quad r_{ij} \in R_i, \quad g \neq r_{ij} \tag{C1}$$

Likewise, we have to avoid the differences in rotation constants used in the linear diffusion layer for the same reason as mentioned before. Hence, the following condition applies for

Table 1: 1st-order column offsets using row offset 0.

	Column offsets for 1 guard
First-order	{2},{4},{8},**{11}**,{14}, {15}, {16},{18},{20},{24},{26}, {27},{29},{31},{32},{33},{35}, {37}, {38},{40},{44},{46},{48}, {49},{50},{53},{56},{60},{62}

R_i with $i \in \{0, 1, 2, 3, 4\}$ and $r_{ij} \in R_i$ with $j \in \{0, 1\}$:

$$\begin{aligned} &\forall g \in G, \quad \forall r_{ij} \in R_i \\ &g \neq r_{i0} - r_{i1} \pmod{64} \\ &g \neq r_{i1} - r_{i0} \pmod{64} \end{aligned} \tag{C2}$$

Furthermore, we have to also avoid the difference between the rotation constants of x_0 and x_4. These bits are xored in the *s-box linear layer* after the *linear diffusion layer*, which is added to achieve the low latency implementation as explained in Section 3.2. It is important to note that this condition may change depending on the selected *row-offset* Hence, the following condition applies for R_i with $i \in \{0, 4\}$ and $r_{ij} \in R_i$ with $j \in \{0, 1\}$:

$$\begin{aligned} &\forall g \in G, \quad \forall r_{ij} \in R_i, \quad \forall j \in \{0,1\} \\ &g \neq r_{0j} - r_{4j} \pmod{64} \\ &g \neq r_{4j} - r_{0j} \pmod{64} \end{aligned} \tag{C3}$$

After applying all the aforementioned conditions (C1)–(C3) and fixing the *row-offset* to 0, we are left with 29 positions that can be selected as a guard *column-offset*. All these positions are provided in Table 1. These guard constraints were exclusively derived due to interactions in one round of the permutation. In practice, we also avoid values which are not co-prime to 64, since these values may line up when evaluating multiple rounds of the permutation. For example we avoid using {32}, as there is a possibility of a leak in the second round for this offset value. Hence, for the implementation we select the first *column-offset* that is co-prime to 64, in out case {11} as highlighted in the Table 1.

4.2 Second-order Masked Implementation without Fresh Randomness

At a first glance, incorporating guards for the second-order scenario may seem straightforward. However, there has been a lack of discussion on expanding and selecting guards in a generic manner for second-order protected implementations. This is primarily due to the increased complexity of the second-order case, which is characterized by $d = 2$ probing locations and $d(d+1)/2 = 3$ fresh random bits per DOM AND gate (see Figure 1). As we intend to avoid using any fresh randomness, we would need guards from three distinct offsets. This significantly increases the complexity of selecting proper guards.

For example, in the second-order implementation of Ascon, we need 3 sets of *row* and *column-offsets* for each DOM AND gate. However, one cannot simply choose a random set out of the offsets obtained in the first-order case. This would neglect the interactions between the selected guards itself. The additional dependency between guards significantly complicates the selection process. In the following, we discuss this interaction of guards in more detail.

To select the guards, we start from the same approach as in the first-order case, where we set the *row-offset* to 0 and solve for the *column-offset* for a single S-box. Once we have obtained the *column-offset*, we again apply the same offsets to all the S-boxes. This will preserve the rotation symmetry in the design and will help to reduce the complexity in the verification later on (see Section 6).

For the *row-offset*, we define two distinct sets of conditions. The first applies independently for all *row-offset* values. The second depend on the selected *row-offset* value. In the following, we limit the discussion to conditions for a fixed *row-offset* value of 0. Similar conditions arise when the *row-offset* value is to a different value.

Due to the significant number of conditions involved, a simple selection of offsets is not trivial anymore. Therefore, we use a constraint formulation approach and employ a SAT solver to find the appropriate offsets based the the defined constraints. We will discuss some of the constraints for each scenario in the following.

Generic Constraints. Here, we discuss the generic constraints that should be applied to obtain the guard column offsets given by the set $G = \{g_1, g_2, g_3\}$ for each DOM AND gate of the S-box in position 0.

First two constraints (C1) and (C2) of the first-order scenario discussed before are also needed for the second-order case $\forall g_i \in G$, where G is the set of containing the *column-offsets* of the guards. When dealing with a second-order scenario where multiple guards are necessary, it's crucial to consider the interaction between these guards. Therefore, we define the following conditions to ensure proper selection:

Any combination of two guards, when combined together should not result in the position that we are solving for, in this case, position 0.

$$\begin{aligned} &\forall g \in G, \quad \forall k,l \in 0,1,2, \quad k \neq l \\ &g_k + g_l \neq 0 \pmod{64} \end{aligned} \tag{G1}$$

When any guards g_k combine with the rotation value r_{ij} of any row x_i should not generate any other guard offset g_l.

$$\begin{aligned} &\forall g \in G, \quad \forall r_{ij} \in R_i, \quad \forall k,l \in 0,1,2, \quad k \neq l \\ &g_k + r_{ij} \neq g_l \pmod{64} \end{aligned} \tag{G2}$$

The combination of any two *column-offsets*, g_k and g_l where $0 \le k,l \le 2$ with the rotation values r_{ij} of the row x_i should not generate the same value. This constraint is applied for all x_i where $i \in \{0,1,2,3,4\}$.

$$\begin{aligned} &\forall g \in G, \quad \forall r_{ij} \in R_i, \quad k \neq l \\ &g_k + r_{i0} \neq g_l + r_{i1} \qquad \pmod{64} \end{aligned} \tag{G3}$$

Constraints that change with the row-offset. Here, we discuss the constraints that are specific to the chosen *row-offset*, in this case 0. The third and last constraint (C3) discussed in the first-order scenario, applies here as well. Along with that we need to also include the constraints which considers the differences between the rotation constants of $\{x_1, x_2\}$ and $\{x_3, x_4\}$.

$$\begin{aligned} \forall g \in G, \quad \forall r_{ij} \in R_i, \quad &\forall j \in \{0,1\} \\ g \neq r_{1j} - r_{2j} \quad &\pmod{64} \\ g \neq r_{3j} - r_{4j} \quad &\pmod{64} \end{aligned} \tag{R1}$$

To clarify, the constraints explained are a simplified version and not the complete set used for encoding. The full set of constraints will be provided for all possible *row-offsets* from 0 to 4.

Solving. After defining all the constraints, we input them into a SAT solver to obtain the assignment of guards. Any solver can be used, but in our case, we utilized the z3 solver [18] to solve the constraints.

The solver generates all possible combinations of *column-offsets* that satisfy the previously defined constraints. The list of *column-offsets* obtained by setting the *row-offset* to 0 is illustrated in Table 2. By using these offsets, we can successfully make guard connections to the DOM AND gate, thereby obtaining a second-order implementation with no additional randomness required. As previously mentioned, selecting a different *row-offset* may result in a different set of solutions.

Shifted domain trick. In our previous discussion, we noted that in the second-order case, we have the advantage of using three different domains for each *column-offset* that is obtained. Additionally, it is beneficial to consider how to connect these distinct domains in a DOM AND gate. For this, we introduce a technique we refer to as the *shifted domain trick* that provides a means of connecting the shares of the guards from the relevant domain to the location of the random bits in a second-order DOM-AND gate. Specifically, we propose to connect a guard from domain A (G_a^1) to Z_0, a guard from domain B (G_b^2) to Z_2, and a guard from domain C (G_c^3) to Z_1, using the notation of Figure 1. This approach helps to create symmetry in the design and reduces some of the constraints related to guard selection. Because of the symmetry introduced by this

Table 2: 2nd-order column offsets using row offset 0.

	Column offsets for 3 guards
Second-order	{2,16,18},{2,20,46},{2,46,48}, {2,46,50},{2,48,50},{4,18,44}, {8,24,26},{8,24,35},{8,24,48}, **{11,35,37}**,{14,16,18},{14,16,32}, {14,18,32},{15,31,35},{16,18,32}, {16,18,62},{16,27,53},{24,26,53}, {24,32,48},{24,48,62},{27,29,53}, {31,35,49},{31,46,62},{32,46,48}, {32,46,50},{32,48,50},{46,48,50}, {46,48,62}

connection technique, any permutation of the solution pair from Table 2 can be used for implementation, which increases the number of solutions for the guard offsets.

5 Implementation Results

The small state size and simple permutation layer of Ascon enable efficient SCA protected implementations with high throughput and low area. We explored multiple secured open-source implementations of Ascon to find suitable ones to improve using our technique.

We first looked into the implementation published by Primas et al. [46]. This implementation is based on the work of Gross et al. in [28, 29] and provides several first and second-order protected implementation of Ascon. It is compliant with the LWC Hardware API [53] and has been evaluated during the course of the NIST LWC project [39]. However, this implementation and the LWC Hardware API did not support hashing functionality along with authenticated encryption. We also considered the corresponding Verilog implementation [45]. This implementation is similar to the previous, a simpler interface and significantly reduces the code size. However, it does not include any SCA protected implementation. Despite this, we have chosen this implementation and extend it using an SCA-protected version using domain-oriented masking.

In line with the recommendations presented in [53] for SCA-protected implementations of Lightweight Cryptography candidates, we employ a technique involving processing inputs and outputs as shares, expanding the BUS width at the interface to handle them, and incorporating an extra random data input port to supply the SCA-protected implementation with fresh randomness. The resulting Ascon core interface, as depicted in Figure 5, is used to enable our improved SCA protected implementation.

Next, we have applied the latency reduction technique described in Section 3.2 two reduce the latency for one round of the permutation by one cycle. Finally, we implement the guard connections based on the results from Section 4 which

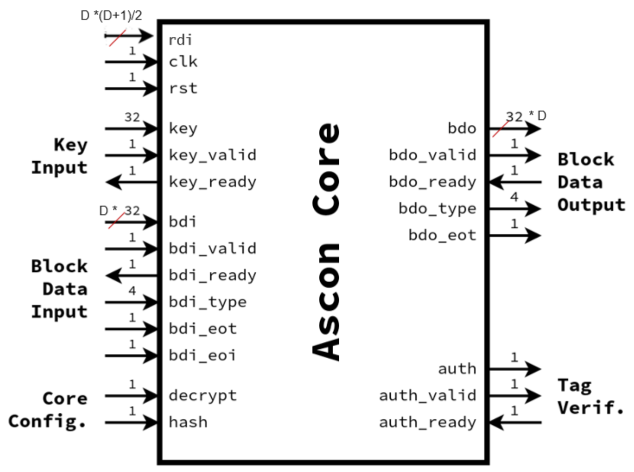


Figure 5: Lightweight Cryptography Ascon Core Implementation with SCA protection based on [53]

results in an implementation of Ascon with improved latency requiring no fresh randomness.

5.1 Latency Reduction

In our implementation, the *Ascon Core* module contains a state machine responsible for performing the authenticated encryption and hashing calculations. Additionally, the module includes an instance of the permutation module, *Asconp*, which implements the permutation that runs through multiple rounds. Now we will discuss the modifications to both the *Asconp* and *Ascon Core* module to implement the the previously mentioned latency reduction technique. Furthermore, it's worth noting that we have utilized this optimization technique for both the first-order and second-order implementations.

Changes in Asconp module. To achieve a permutation with a 2-cycle latency, the implementation of the *Asconp* module has been modified. The *S-box linear layer* has been moved and placed after the *Linear diffusion layer*, and the permutation starts from the χ mapping of the S-box. The round constant of the next round is also calculated and added to the *S-box linear layer*. Furthermore, we have added specific logic to prevent the integration of the next *S-box linear layer* during the final round of the permutation.

Changes in Ascon Core module. To achieve functional correctness, we redesign the *Ascon Core* module to include the *S-box linear layer* before initiating the first round of permutation. This is due to the modified structure of the low-latency *Asconp* module. Since each phase of the algorithm, such as initialization, processing associated data, processing plaintext/ciphertext, and finalization, triggers a specific number of permutation rounds, each phase requires some modifications to integrate the *S-box linear layer*.

5.2 Implementing the Guards

After incorporating the low-latency permutation, we proceed to add guards that allow us to obtain the masked implementation of Ascon without the need for fresh randomness. From the options available in Table 1 and Table 2, we choose one of the *column-offset* possibilities and substitute it for the $d(d+1)/2$ random bits in the DOM AND gate. The connections are made according to the *shifted domain technique* described in Figure 1.

For the purpose of verification, we have implemented all possible options for both first-order and second-order scenarios from Table 1 and Table 2. In our implementation, we simply select the first *column-offset* that is co-prime to 64, which is $\{11\}$ in the first-order case and $\{11, 35, 37\}$ in the second-order case to implement the guard connections accordingly. We use this co-prime selection as it provides better security considering multiple rounds.

5.3 Performance and Area

To reduce latency, our proposed method entails restructuring the permutation layer and introducing additional control logic to maintain functional correctness, which results in lower overhead in terms of area cost as the linear layers are just moved around. Since, the latency is reduced by one cycle, we remove one register stage which also helps to reduce the area cost. Importantly for randomness optimization, implementing guards will not require additional area overhead as they are simply connections from different S-box inputs. It's worth noting that this approach won't require any additional modifications to the interface.

After comparing our implementation with other available implementations, we can say that we have achieved lower latency and no requirement for fresh randomness at a comparatively lesser area overhead which can be seen in Table 3. The Synopsys Design Compiler Version T-2022.03-SP2, along with the *lsi_10k* library, was used to synthesize the design and compute the area numbers displayed for our module. Although we acknowledge that the area numbers referenced in the table are obtained from diverse synthesis tools utilizing different technologies, we attempt to make a rough comparison.

6 Formal Verification

The benefits of formal verification techniques are already outlined in Section 2.4. This section focuses on the specific tool we use for verification, CocoAlma [30], and elaborates on the

Table 3: Ascon Permuation Synthesis Result

Masking Order	**Cycles /round**	**Randomness (bits/round)**	**Area (kGE)**	**Ref.**
1	3	320	28.89	[28, 29]
	2*	-	**26.10**	**ours**
	1	320	50.40	[41]
	1	2048	42.75	[27]
2	3	960	53.00	[28, 29]
	2*	-	**52.63**	**ours**
	1	960	102.39	[41]
	1	4608	90.94	[27]

* Latency can be reduced to 1 cycle using rising and falling edge clock.

verification results for first and second-order implementations with reduced latency requiring no fresh randomness.

CocoAlma is a formal verification tool that verifies the side-channel resistance of masked implementations. It is designed to verify masked hardware implementations while taking into account hardware effects such as glitches.

The workflow of CocoAlma includes Yosys [55], an open-source synthesis tool for obtaining the synthesized netlist from the design files. It provides a parsing script, for obtaining the circuit graph of the netlist and additionally creates a labeling template that can be used to label the inputs as secrets or randomness. An execution trace is generated by utilizing Verilator [51] and the user-provided testbench. This provides a *value change dump* of the internal signal variations during the execution of the implementation. The data gathered from the circuit graph, execution trace, and labeling file is encoded by a verification script into a SAT problem, which is then solved using a SAT solver like CaDiCaL [7] or Kissat [8]. The unsatisfiability of the problem shows that the observation of any intermediate computation would not leak any information about the secret for a given security level and the respective protection order.

6.1 Setup

Our verification setup is designed to verify the Ascon permutation for both first and second-order implementations. To simplify the process, we exclude the interface and the *Ascon Core* and create a wrapper around the permutation module, *Asconp* that runs it for a selected number of rounds. We then verify the permutation for single and multiple rounds, while maintaining a consistent verification setup for all of our implementations.

We start by providing the hardware design files of the selected implementation into the parsing script. This script synthesizes the design and generates the netlist and label template. In the label template, all $320 \times (d+1)$ input bits to the Ascon permutation, are labeled as *secrets*. As the implementation does not use any random bits, there is no need to label them.

To generate an execution trace, a C++ testbench is created by assigning values to the inputs and control signals. The trace generation script uses this testbench to generate the execution trace. The resulting netlist and execution trace are then provided to the verification script, along with the label definitions. This script can take additional parameters, such as the protection order to be verified, the number of cycles of verification, and the mode of verification. Two modes of verification are available: stable case and transient case. The transient case is particularly interesting to verify because it covers glitches. As a result, we have used this mode to verify all of our masked implementations of the Ascon permutation.

Apart from the aforementioned parameters, the user can also specify the checking mode and probing model when using the tool. The tool offers two checking modes: per-location and per-secret. For first-order designs, per-location is recommended because it builds a formula for every potentially leaking location, which is faster in the first-order case. On the other hand, per-secret is faster for higher-order designs, because it builds a formula for every secret. Since the primary objective is to validate second-order design, the per-secret checking mode is chosen specifically for our implementation.

6.2 Verification of a Single Round

First, we verify the low-latency implementation using CocoAlma, without any randomness optimization, to ensure that the latency-reducing restructuring of the linear layers does not introduce any security issues. Subsequently, we proceed to verify a single round of our first and second-order Ascon permutation implementations without any fresh randomness.

First-order masked implementation. We utilize the same CocoAlma setup as described earlier to verify the first-order implementation with guards, which requires no fresh randomness. We label all 320 secrets, while random bits need not be labeled. The verification is performed for three cycles, as the permutation necessitates two cycles, and the wrapper adds an extra cycle. We verify all the 29 candidates from Table 1 for the first-order case. Our analysis shows that all these first-order implementation with guards are considered secure for a single round of permutation. It took less than a minute to verify each implementation with CocoAlma on a core of AMD EPYC 74F3 with a 3.2GHz base clock.

Second-order masked implementation. In the case of second-order designs, the complexity of formal side-channel verification increases significantly. This is mainly because, in the case of Ascon, which has a state size of 320 bits, every bit must be verified for each cycle. This adds to the complexity of building the correlation sets and formulating the conditions required to check the leakage of each secret bit, taking into account multiple probing locations across all cycles. Moreover, in our implementation, the use of guards, which are essentially the secrets from a different S-boxes, further increases the complexity by introducing interactions between the secrets and making the overall SAT formula more complex.

To test the second-order case, there are 28 candidates that need to be implemented and verified as indicated in Table 2. Due to the verification complexity involved, symmetry in the Ascon implementation is exploited to reduce the runtime, and only five secrets belonging to a single S-box are verified. Our proposal for guard assignment for each S-box also preserves the symmetry because of the offset-based selection method. However, all 320 secrets are labeled and the verification is run again for 3 cycles. As we verified only 5 secret bits, the verification took only about 100s on a core of AMD EPYC 74F3 with a 3.2GHz base clock. Despite exploiting symmetry and verifying only 5 secrets, the runtime of the verification process for the second-order case significantly increases, highlighting the complexity involved in verifying second-order implementations.

6.3 Verification of Multiple Rounds

While prior research has primarily focused on formally verifying a single round, it is worthwhile to extend the verification process to multiple rounds of the Ascon permutation, particularly because of its simple structure and heavily optimized randomness requirements. Although some recent work investigates the security of multiple rounds of masked AES with reduced randomness, they only provide a pen-and-paper approach to argue about the multi-round leakage [19].

In our work, we extended the formal verification of side-channel security to the second round of the permutation. This is particularly relevant for our implementation, where the guards offsets are obtained by encoding constraints that consider only the interactions during the first round of permutation. On the other hand, it's worth highlighting that exploiting the leakage after the second round is extremely challenging due to the high diffusion of the Ascon permutation. In this section, we discuss the verification of two rounds of both first and second-order masked implementation of the Ascon permutation.

First-order masked implementation. In this section, we present the verification results of the first-order masked implementation of Ascon without fresh randomness. We repeat the verification using a similar setup as before but increase the number of cycles for which the CocoAlma is executed to cover the two rounds of permutation.

The results demonstrate that the all the 29 first-order implementations with the guards from Table 1 requiring no fresh randomness are considered secure for the first two rounds of permutation except for the value $\{32\}$ as expected. For each implementation with the guard, we verified all the 320 secrets, and the verification takes about 108 minutes on an average for each candidate on an AMD EPYC 74F3 24-Core Processor with a 3.2GHz base clock.

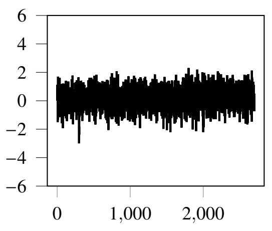
It is also essential to note the significant increase in verification time when extending the first-order implementation to two rounds. While verifying the first-order designs for one round only took a few seconds to minutes, verifying the two complete rounds took almost an hour. This observation already suggests potential complexity issues that will be encountered in the second-order case, which we will discuss in the next section.

Second-order masked implementation. The complexity involved in second-order verification is considerable, and extending it to multiple rounds only exacerbates the issue, as it requires considering an increased number of cycles for verification purposes. We have noticed that it can take two to four days for CocoAlma to verify each secret for two rounds in a second-order case when running on a core of AMD EPYC 74F3 with a 3.2GHz base clock. Since CocoAlma checks the secrets sequentially, verifying the entire implementation would take a considerable amount of time. To address this, as mentioned before, we take advantage of the symmetry in the Ascon permutation, and verify just one S-box which has 5 secrets.

Furthermore, since the formulas are built separately for each secret, we can build the formula for each secret and run them in parallel to reduce the verification time even further. To select the secrets from a specific S-box or column, the labeling file provided to CocoAlma needs to be altered accordingly. Additionally, we apply a local patch to the CocoAlma code to enable the selection of a set of secrets from a given S-box for verification. It's important to highlight that even though we select only one S-box for verification, all the secrets are labeled, and the formula is built considering the entire implementation. Only a selected set of secrets are verified, but they are done so in parallel, reducing the overall verification time.

By taking advantage of symmetry in the design and verifying the secrets concurrently, the second-order implementation was verified for one of the *column-offset* combinations from Table 2. Here we selected the set of guard offsets that are co-prime to 64 and the first occurrence of such a combination in our list is $\{11, 35, 17\}$. This verification process took about two to four days. Additionally, for comprehensive verification, we also verified all 320 secrets for this particular design implementing the highlighted guard offset from Table 2. This was achieved by running a larger number of secrets in parallel on multiple machines for a couple of weeks. After verifying all the 320 secrets, the implementation was seen to show no second-order leakage for two rounds of permutation.

We anticipate that all the guard offset possibilities mentioned in Table 2 do not show leakage for two rounds, but we cannot claim it since we could not verify all of them for two rounds due to the time-consuming nature of the verifica-


(a) 1st-order t-test

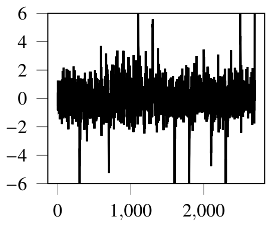
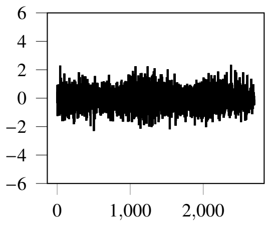

(b) 2nd-order t-test

Figure 6: First-order implementation showing no leakage in the 1st-order t-test and expected leakage in the 2nd-order t-test.


(a) 1st-order t-test

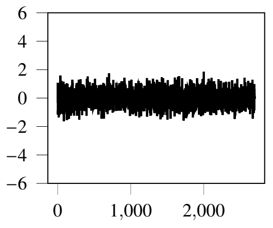

(b) 2nd-order t-test

Figure 7: Second-order implementations showing no leakage.

tion process, despite having verified them for one round of permutation. Also, it's challenging to argue the security for two rounds because the constraints were derived based on the structure of the linear layers in the first round.

7 Experimental Evaluation

To further assess the security of our designs, we employ a statistical analysis based on the leakage assessment methodology of [49] on both of our first and second-order implementations. Concretely, we performed the non-specific fixed vs. random Welch's t-test over 10 million traces for both designs. For both first and second-order designs, we report the univariate analysis of the 1st and 2nd statistical moments. For the second-order design, we additionally report the result of bivariate analysis.

Experimental setup and results. We measured the first 12 permutation rounds of an encryption, corresponding to the initialization of the Ascon sponge prior to encryption. We limit our analysis to this operation as this function directly operates on the secret key and is the most suitable target for an attack by an adversary.

We performed our measurements on the CW305 Artix-7 target board from NewAE and recorded the traces with a Picoscope 6000-series. The FPGA is clocked at 1.5625MHz which is provided to the Picoscope as a reference clock. We measure 100 samples of each clock cycle on the FPGA, corresponding to 156.25M samples per second. For each recorded trace,

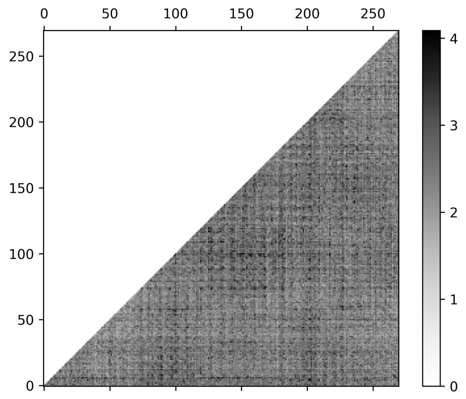


Figure 8: Bivariate t-test on the second-order implementation showing no significant leakage.

a fixed or random key is selected and shared with random masks. The plaintext (or nonce) is fixed to a masked vector of zeros. We compute the t-scores online using SCALib [10]. For the first-order implementation, the 1st-order t-test given in Figure 6a is below the accepted ±4.5-sigma bound, whereas the 2nd-order t-test exhibits expected leakage in Figure 6b. For the bivariate analysis of the second-order implementation (Figure 8), we compute the bivariate t-score over every pairwise combination of sample points in a trace and report the absolute value. The second-order implementation does not show any leakage for both the 1st and 2nd-order t-tests.

8 Conclusion

Our implementation of Ascon outperforms other implementations in several aspects. Specifically, it has lower latency, requires no fresh randomness, and has low area overhead compared to existing implementations, making it an interesting choice for use in constrained IoT devices. We have formally verified both first and second-order implementations using CocoAlma for two rounds of permutation, and conducted statistical t-tests on the hardware which indicated no leakage for either implementation. The technique suggested for selecting guards can be modified and adapted for use with other cryptographic schemes like SHA-3 and Xoodyak.

References

[1] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic Keccak: SCA Security and Low Latency in HW. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):269–290, 2018.

[2] Victor Arribas, Svetla Nikova, and Vincent Rijmen. Vermi: Verification tool for masked implementations. In *ICECS*, pages 381–384. IEEE, 2018.

[3] Victor Arribas, Svetla Nikova, and Vincent Rijmen. Guards in action: First-order SCA secure implementations of KETJE without additional randomness. *Microprocess. Microsystems*, 71, 2019.

[4] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS (1)*, volume 11735 of *LNCS*, pages 300–318. Springer, 2019.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *EUROCRYPT*, volume 7881 of *LNCS*, pages 313–314. Springer, 2013.

[6] Tim Beyne, Siemen Dhooghe, Adrián Ranea, and Danilo Sijacic. A low-randomness second-order masked AES. In *SAC*, volume 13203 of *LNCS*, pages 87–110. Springer, 2021.

[7] Armin Biere. CADICAL at the SAT Race 2019. 2019.

[8] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING Entering the SAT Competition 2020. 2020.

[9] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *EUROCRYPT (2)*, volume 10821 of *LNCS*, pages 321–353. Springer, 2018.

[10] Gaëtan Cassiers and Olivier Bronchain. Scalib: A side-channel analysis library. *Journal of Open Source Software*, 8(86):5196, 2023.

[11] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[12] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *CHES*, volume 1965 of *LNCS*, pages 252–263. Springer, 2000.

[13] Christophe Clavier, Jean-Luc Danger, Guillaume Duc, M. Abdelaziz Elaabid, Benoît Gérard, Sylvain Guilley, Annelie Heuser, Michael Kasper, Yang Li, Victor Lomné, Daisuke Nakatsu, Kazuo Ohta, Kazuo Sakiyama, Laurent Sauvage, Werner Schindler, Marc Stöttinger, Nicolas Veyrat-Charvillon, Matthieu Walle, and Antoine Wurcker. Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. *Journal of Cryptographic Engineering*, 4(4):259–274, 2014.

[14] D. J. Bernstein. The CAESAR competition: CAESAR submission format. `https://competitions.cr.yp.to/caesar-submissions.htm`, 2017. Accessed: May 25, 2023.

[15] Joan Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In *CHES*, volume 10529 of *LNCS*, pages 137–153. Springer, 2017.

[16] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology*, 2020(S1):60–87, 2020.

[17] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020.

[18] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[19] Siemen Dhooghe, Aein Rezaei Shahmirzadi, and Amir Moradi. Second-order low-randomness d + 1 hardware sharing of the AES. In *CCS*, pages 815–828. ACM, 2022.

[20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. ISAP v2.0. *IACR Transactions on Symmetric Cryptology*, 2020(S1):390–416, 2020.

[21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the NIST Lightweight Cryptography project, 2019. `https://ascon.iaik.tugraz.at`.

[22] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *journal of Cryptology*, 34(3):33, 2021.

[23] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, 2018.

[24] Sebastian Faust, Clara Paglialonga, and Tobias Schneider. Amortizing randomness complexity in private circuits. In *ASIACRYPT (1)*, volume 10624 of *LNCS*, pages 781–810. Springer, 2017.

[25] Jakob Feldtkeller, David Knichel, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Randomness optimization for gadget compositions in higher-order masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):188–227, 2022.

[26] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In *CHES*, volume 1717 of *LNCS*, pages 158–172. Springer, 1999.

[27] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):1–21, 2018.

[28] Hannes Gross and Stefan Mangard. Reconciling d+1 Masking in Hardware and Software. Cryptology ePrint Archive, Paper 2017/103, 2017. https://eprint.iacr.org/2017/103.

[29] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Paper 2016/486, 2016. https://eprint.iacr.org/2016/486.

[30] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *FMCAD*, pages 1–10. IEEE, 2021.

[31] Yu Han, Xuecheng Zou, Zhenglin Liu, and Yi-Cheng Chen. Efficient DPA attacks on AES hardware implementations. *International Journal of Communications, Network and System Sciences*, 1(1):68–73, 2008.

[32] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In *ACNS*, volume 3989 of *LNCS*, pages 239–252, 2006.

[33] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205. IEEE Computer Society, 2013.

[34] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[35] David Knichel and Amir Moradi. Composable gadgets with reused fresh masks - first-order probing-secure hardware circuits with only 6 fresh masks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):114–140, 2022.

[36] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits towards automated generation of composable secure gadgets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):323–344, 2022.

[37] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[38] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[39] Kamyar Mohajerani, Luke Beckwith, Abubakr Abdulgadir, Eduardo Ferrufino, Jens-Peter Kaps, and Kris Gaj. SCA evaluation and benchmarking of finalists in the NIST lightweight cryptography standardization process. Cryptology ePrint Archive, Paper 2023/484, 2023. https://eprint.iacr.org/2023/484.

[40] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):256–292, 2019.

[41] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. Riding the waves towards generic single-cycle masking in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):693–717, 2022.

[42] National Institute of Standards and Technology. Lightweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography. Accessed: May 25, 2023.

[43] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[44] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *CT-RSA*, volume 3860 of *LNCS*, pages 192–207. Springer, 2006.

[45] Robert Primas. ascon-verilog. GitHub repository, 2023.

[46] Robert Primas and Rishub Nagpal. ascon-hardware-sca. GitHub repository, 2022.

[47] Oscar Reparaz. A note on the security of higher-order threshold implementations. Cryptology ePrint Archive, Paper 2015/001, 2015. https://eprint.iacr.org/2015/001.

[48] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *CRYPTO (1)*, volume 9215 of *LNCS*, pages 764–783. Springer, 2015.

[49] Tobias Schneider and Amir Moradi. Leakage assessment methodology - extended version. *Journal of Cryptographic Engineering*, 6(2):85–99, 2016.

[50] Aein Rezaei Shahmirzadi, Siemen Dhooghe, and Amir Moradi. Low-latency and low-randomness second-order masked cubic functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):113–152, 2023.

[51] Wilson Snyder. Veripool — veripool.org. https://www.veripool.org/verilator/. [Accessed 09-Oct-2022].

[52] Takeshi Sugawara. 3-share threshold implementation of AES s-box without fresh randomness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):123–145, 2019.

[53] Michael Tempelmeier, Farnoud Farahmand, Ekawat Homsirikamol, William Diehl, Jens-Peter Kaps, and Kris Gaj. Implementer's guide to hardware implementations compliant with the hardware API for lightweight cryptography. 2019.

[54] Felix Wegener and Amir Moradi. A first-order SCA resistant AES without fresh randomness. In *COSADE*, volume 10815 of *LNCS*, pages 245–262. Springer, 2018.

[55] Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.