IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 3, pp. 708–755. DOI:10.46586/tches.v2021.i3.708-755

Second-Order SCA Security with almost no Fresh Randomness

Aein Rezaei Shahmirzadi and Amir Moradi

Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
firstname.lastname@rub.de

Abstract.
Masking schemes are among the most popular countermeasures against Side-Channel Analysis (SCA) attacks. Realization of masked implementations on hardware faces several difficulties including dealing with glitches. Threshold Implementation (TI) is known as the first strategy with provable security in presence of glitches. In addition to the desired security order d, TI defines the minimum number of shares to also depend on the algebraic degree of the target function. This may lead to unaffordable implementation costs for higher orders. For example, at least five shares are required to protect the smallest nonlinear function against second-order attacks. By cutting such a dependency, the successor schemes are able to achieve the same security level by just $d + 1$ shares, at the cost of high demand for fresh randomness, particularly at higher orders.

In this work, we provide a methodology to realize the second-order glitch-extended probing-secure implementation of a group of quadratic functions with three shares and no fresh randomness. This allows us to construct second-order secure implementations of several cryptographic primitives with very limited number of fresh masks, including Keccak, SKINNY, Midori, PRESENT, and PRINCE.

Keywords: Side-Channel Analysis · Masking · Hardware · Threshold Implementation

1 Introduction

Physical attacks are a serious threat for many security-critical devices, where the attacker tries to gain sensitive information by monitoring the physical properties of the target device. Physical observations like power consumption or electromagnetic radiations can be exploited to recover sensitive data, if no proper countermeasure is employed. Hence, resistance against Side-Channel Analysis (SCA) attacks – as a sort of physical attack – is an important requirement for almost any deployed cryptographic device.

Kocher et al. [KJJ99] exploited the relation between the calculations performed on intermediate values and power consumption of the target device for the first time. Introduction of such a seminal work opened a new line of research into SCA attacks leading to more sophisticated attack strategies like Correlation Power Analysis (CPA) [BCO04], Mutual Information Analysis (MIA) [GBTP08], and Moments-Correlating DPA (MC-DPA) [MS16b]. Recently, several studies focus on applying Deep Learning (DL) to improve state-of-the-art SCA attacks [Tim19, RAD20]. This highlights the necessity of employing appropriate countermeasures to ensure physical security in applications, where the adversary has a chance to control the device. A wide range of strategies and theories have been proposed in the open literature to limit or eliminate the amount of SCA leakages, some focusing on software implementations, others on hardware platforms. Masking schemes, due to their sound theoretical basis and a good understanding of their requirements, are among the

Licensed under Creative Commons License CC-BY 4.0.

Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-07-09

most common methods applied in practice and studied by the researchers. In masking schemes, sensitive intermediate values are randomized during the execution of the cipher breaking the relation between the processed secret-dependent data and physical properties of the underlying device.

Generally, in a masking scheme, every key-dependent variable is split into several shares (defining the order of sharing/masking), and all computations are performed on the masked data which can be seen as performing the computations on secret-shared data. Boolean masking is surely the most popular approach, although other masking schemes like Multiplicative masking [MRB18] or Inner Product masking [BFG^{+}17] can be beneficial depending on the application. It has been shown that masking increases the measurement complexity of an SCA attack exponentially in the number of shares provided that the leakage of each share is noisy enough and that each power sample depends on a bounded number of shares. This level of protection does not come for free; the area overhead and the latency of an implementation realizing a masking scheme grows approximately quadratically with respect to the number of shares [FGP^{+}18].

In the context of masking, the seminal contribution has been made by Trichina [Tri03] where a first-order secure AND gate was presented. As a follow-up work, Ishai et al. [ISW03] introduced a general methodology to mask a 2-input AND gate at the desired security order. However, the existing research exhibits leakage in their hardware implementations due to a well-known phenomenon in hardware platforms called glitches [MPO05, MME10]. Glitches are unwanted signal transitions at the output of a combinatorial circuit due to the unbalanced delay of its inputs. Hence, the result of the calculation also depends on the timing of the inputs and can potentially cause exploitable leakage in practice.

To assess the security of a given design some sort of abstraction should be made. The most convincing approach in this context is *probing security model*, firstly introduced in [ISW03]. In this method, the security order of the design is defined by the maximum number of probes that the attacker can place on intermediate signals of the circuit and observe their values simultaneously. It appears consistent with software implementations where the instructions are executed sequentially and each of them can be considered as an atomic gate. In contrast, this abstraction does not consider physical defaults, and hence the designs may exhibit leakage even though they are shown secure under the probing model. Over time, extensive research has been devoted to understand how to adjust this model covering hardware platforms. After introducing several models, the most convincing one seems to be *robust probing model* presented in [FGP^{+}18]. In order to take glitches into account, each probe in the *glitch-extended* probing model is extended to multiple probes implying an even stronger adversary. By probing a signal in a combinatorial circuit, the adversary gains information about all intermediate values and input signals involved in the calculation of the probed signal.

A critical question more than a decade ago was how to make a design secure considering physical defaults like glitches. In order to properly address this question, three comprehensible properties, associated with an implementation strategy called Threshold Implementation (TI), have been introduced in [NRR06]. This strategy is immune to glitches and guarantees the security of the design if all properties are fulfilled. In the underlying methodology, the number of shares is defined based on the desired security order and the algebraic degree of the function. It has been shown that the same level of security can also be achieved with the minimum possible shares, i.e., independent of the algebraic degree of the underlying function [RBN^{+}15, GMK16] even in the presence of glitches. In this technique, the masked realization is split into two parts, where registers should be placed in between to avoid the propagation of glitches, and fresh randomness should be used to avoid the leakages. In [SM20] a methodology is presented which avoids using fresh randomness in the first-order secure hardware implementations with two shares. The authors provided a first-order secure 2-input AND gate without fresh randomness for the first time and presented first-order secure implementation of a couple of ciphers with no fresh randomness under glitch-extended probing model.

The situation is a bit different at higher orders. Classical TI forces to use at least five shares to protect the smallest non-linear function against second-order attacks [BGN+14]. In contrast, when using minimum number of shares of three for such a security level, the use of a relatively high number of fresh masks is mandatory. For example, a second-order masked AES S-box requires 162 fresh mask bits [CRB+16] (alternatively 84 bits [GMK17]).

1.1 Our Contributions

In this work, we pass a further step and introduce three-share hardware constructions which can provide second-order security without fresh randomness. Due to the complexity of the algorithms as well as the constructions, we limit ourselves to quadratic functions, i.e., with algebraic degree of two. In short, we present a group of quadratic functions whose three-share second-order secure hardware implementation can be realized without fresh randomness. The other not-supported functions need to be decomposed to such quadratic functions, which necessitates refreshing the intermediate shares to avoid multivariate leakages. As an outcome of our research, we provide three-share second-order glitch-extended probing-secure implementations of Keccak with no fresh randomness and the S-box of SKINNY, Midori, PRESENT, and PRINCE ciphers using only 8-bit fresh masks per clock cycle. We would like to highlight that we confirm the second-order security of our constructions by SILVER [KSM20] under glitch-extended probing model and by FPGA-based practical experiments. Our programs as well as the hardware implementations (HDL codes) are fully provided in the GitHub.

2 Background

In the following, after giving the used notations and basic definitions, we review the concept behind probing and glitch-extended probing security and restate the fundamental concepts of hardware masking, which are required to follow the rest of the paper.

2.1 Notations and Definitions

We denote binary variables $\in \mathbb{F}_2$ with lower-case italic x and vectors $\in \mathbb{F}_2^{n>1}$ with upper-case italic X. We represent j-th element in a vector X with superscripts x^j, i-th share of a variable with subscripts x_i, coordinate functions with lower-case italic sans-serif $\mathsf{f}(.)$, vectorial Boolean functions by upper-case italic sans-serif $\mathsf{F}(.)$, and sets with calligraphic font $\mathcal{F}$.

A Boolean function of n variables is a function of the form $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$, where $\mathbb{F}_2^n$ is the n-dimensional vector space over $\mathbb{F}_2$. We denote $\mathsf{F} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ to show a vector of Boolean functions.

$$\mathsf{F}(X) = \begin{pmatrix} \mathsf{f}^1(X) \\ \mathsf{f}^2(X) \\ \vdots \\ \mathsf{f}^m(X) \end{pmatrix} \quad \text{such that} \quad \forall i, \mathsf{f}^i : \mathbb{F}_2^n \to \mathbb{F}_2 \quad \text{and} \quad X \in \mathbb{F}_2^n \tag{1}$$

Definition 1. The weight of a Boolean function is defined as $wt(\mathsf{f}) = \big|\{X|\mathsf{f}(X) = 1\}\big|$.

Definition 2. A Boolean function $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$ is balanced, if

$$wt(\mathsf{f}) = 2^{n-1} \iff \big|\{X|\mathsf{f}(X) = 1\}\big| = \big|\{X|\mathsf{f}(X) = 0\}\big|.$$

In this paper, we use Algebraic Normal Form (ANF), which is a representation of a Boolean function with a polynomial of n variables of $X = \langle x^1, \ldots, x^n \rangle$. In other words, every Boolean function $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$ can uniquely be expressed by an element in $\mathbb{F}_2[x^1, \ldots, x^n]$, where $\mathbb{F}_2[x^1, \ldots, x^n]$ is the ring of all polynomials with coefficients in $\mathbb{F}_2$.

Definition 3 (Algebraic degree of a Boolean function)**.** The algebraic degree of a Boolean function $\mathsf{f} : \mathbb{F}_2^n \to \mathbb{F}_2$, where $\mathsf{f}(X) = \underset{V \in \mathbb{F}_2^n}{+} \alpha_V \prod_{v^i=1} x^i$, is defined as:

$$deg(\mathsf{f}) = \max_V \Big\{ \sum_{i=1}^{n} v^i \Big| \alpha_V = 1 \Big\},$$

where $\forall V, \alpha_V \in \mathbb{F}_2$ and by v^i we refer to the i-th element of V. In other words, the maximum number of variables that have to be multiplied determines the algebraic degree of a given Boolean function. Further, algebraic degree of a vectorial Boolean function is determined by the maximum algebraic degree of its coordinate functions.

In order to apply Boolean masking to provide s-th order security, we first split a variable x into at least $s+1$ shares x_i where $i \in \{0, 1, \ldots, s\}$ such that the sum of these shares is equal to the original value, i.e., $x = \underset{\forall i}{+} x_i$. As an initial sharing, we naturally can draw x_1 to x_s from a uniform distribution at random and form the first share $x_0 = x + \underset{1 \leq \forall i \leq s}{+} x_i$. The application of Boolean masking on linear functions is straightforward, since the same function can be applied on each share independently. However, implementing the masked realization of a non-linear Boolean function is non-trivial and special care should be taken to avoid any leakage. This is actually the main difficulty and the core of major publications in the areas of Boolean masking.

2.2 Probing Security

Masking, as the most promising countermeasure against SCA attacks, has been widely applied in practice. Consequently, many different schemes have been proposed over the years, considering different applications, assumptions and security requirements [Tri03, ISW03, NRR06, RBN⁺15, GMK16, NRS11, GM18, GIB18]. Some of the interesting questions in this context are how to evaluate the proposed masking schemes under different adversary models, and how to consider execution environments and physical defaults in the evaluation process.

One of the first attempts to address such questions was presented in [ISW03], where d-probing security model was proposed. In this model, the adversary is allowed to observe (probe) up to d intermediate values during the execution of the cipher. It has been repeatedly shown (e.g., in [MPO05, MME10]) that hardware implementations of such d-probing secure schemes fail to deliver security in practice. Namely, this model fits best into software platforms, where instructions can be viewed as atomic gates and there is no data-dependent activation timing. However, it is an inaccurate assumption for hardware platforms due to a common fact in CMOS technologies called glitches. In fact, the d-probing model does not cover specific physical defaults, such as glitches, couplings, or transitions [FGP⁺18]. These undesired effects, which are inherent to the nature of physical implementations, may occur during the execution of the cipher on a device, hence violating the security assumptions leading to exploitable leakages.

To consider physical characteristics in the verification model, the relevant scientific communities have conducted extensive research on the development of formal models. After some trial and error, Faust et al. [FGP⁺18] addressed the aforesaid questions by proposing an extension of the d-probing model called *robust probing* model, which can cover inherent physical properties of hardware platforms when evaluating the security of a design. Focusing on *glitch-extended* feature of such a comprehensive model, each probe, placed on a combinatorial circuit, propagates backward to the last synchronization point (registers). In other words, by placing a single probe, the adversary has information about all signals that may contribute to determine the value of the probed signal. This simple but effective abstraction significantly helped reducing the implementation cost of several schemes [BGR18, SM20]. Moreover, using such glitch-extended probing model, the authors of [MMSS19] demonstrated the insecurity of previous hardware-oriented masking schemes such as [RBN+15, GMK17, GMK16]. This highlighted the importance and necessity of probing security proofs in masked implementation, leading to the development of formal verification tools to evaluate the given designs [BBC+19, KSM20].

2.3 Masking with $td + 1$ Shares

Classical TI [NRR06] is the first implementation strategy that is immune to glitches in hardware implementations. It defines the minimum number of input shares as $td + 1$, where t and d stand for the algebraic degree of the function and the desired security order, respectively. Let us suppose that the number of input shares and output shares are the same and equal to $s + 1$. Hence, a masked realization of $Y = \mathsf{F}(X)$ receives input shares $X_0, \ldots, X_s$ and provides output shares $Y_0, \ldots, Y_s$. Three essential properties were introduced in [NRS11] to guarantee first-order security. First of all, the sum of output shares should yield to the original output value for the sake of correctness, i.e., $\forall X_i, \mathsf{F}(\underset{\forall i}{+} X_i) = \underset{\forall i}{+} Y_i$. In order to fulfill the second property, the computation of each output share should be independent of at least one input share. To this end, the authors suggested to avoid giving X_i as an input to the function which generates the output share Y_i, so-called a *component function*. This property, called non-completeness, guarantees the glitch-resistance as the leakage of each component function is independent of X. Alternatively, by placing a glitch-extended probe on any component function, the adversary observes only a non-complete shared input, hence no information about X. The third and last property implies that for each value of X, all possible input shares $X_0, \ldots, X_s$ lead to a set of $Y_0, \ldots, Y_s$ which can be represented as $\mathsf{F}(X) = Y$ being shared by masks uniformly selected at random. Note that uniformity itself is neither a necessary nor a sufficient condition to achieve security of an implementation [MBR19]. It actually becomes important when secure gadgets are composed. In other words, we need to guarantee that the second gadget receives a uniformly-shared input. Otherwise, the essential underlying assumption of masking (secret sharing) is violated.

While non-completeness can be easily achieved by a methodology called direct sharing [NRS11], no systematic way is known to fulfill uniformity except remasking, i.e., refreshing the sharing of the output using fresh randomness. Daemen [Dae17] introduced a trick called changing of the guards to relax the need for fresh masks at every clock cycle. The underlying concept is based on re-using the unrelated parts of the cipher, e.g., shares of the neighboring S-box(es), as fresh masks. We should highlight that this technique can fulfill the uniformity of a correct and non-complete shared function, but cannot be beneficial if non-completeness is violated.

In short, constructing secure implementations becomes costly when the algebraic degree of the function increases or high-order security is desired. To cope with this issue, the target function is decomposed into smaller functions as their masked variants are easier to achieve. The secure constructions of the smaller functions are composed with register stages in between to avoid the propagation of glitches, or let say to avoid the propagation of glitch-extended probes on all shares of a variable. TI also covers higher-order security [BGN+14], where the main difference to the first order is the adjusted definition of non-completeness. Considering probing security, in a d-order secure implementation, every d probes placed on any part of the circuit should be independent of at least one share of every variable. As highlighted in [Rep15], the output sharing of a higher-order secure gadget should be refreshed before being fed into the next higher-order secure gadget.

2.4 Masking with $d+1$ Shares

The high implementation cost of TI circuits has been reported using several case studies (e.g., [MPL+11, CBR+15]), particularly at higher orders due to using a high number of shares which naturally scales to a significant amount of fresh randomness when composing the functions. In two independent works [RBN+15, GMK16], it has been tried to make the number of shares independent of the algebraic degree of the function, i.e., $d+1$ shares for d-order security. These constructions can potentially lead to lower implementation costs in terms of area overhead and latency while maintaining the same level of security and glitch resistance that TI offers. In these techniques, the masked variant of the target function consists of two separate parts, which are divided by dedicated registers. More importantly, fresh masks should be used to ensure the security of the gadgets. Precisely speaking, in contrast to $td+1$ where fresh masks might be need to fulfill the uniformity when the functions are composed, in $d+1$ the fresh masks are essential to achieve non-completeness. In other words, an standalone $td+1$ function can be secure without any fresh masks, but its $d+1$ variant demands fresh masks for its security.

Following Domain Oriented Masking (DOM) [GMK16], which needs slightly-less fresh masks compared to [RBN+15] in certain scenarios, a two-share masked variant of a 2-input AND gate $x = f(a, b)$ can be realized as:

$$\begin{array}{lllr} f_0(a_0,b_0) & = a_0b_0 & \rightarrow x'_0 & \\ f_1(a_0,b_1,r) & = a_0b_1 + r & \rightarrow x'_1 & x'_0 + x'_1 = x_0 \\ \hline f_2(a_1,b_0,r) & = a_1b_0 + r & \rightarrow x'_2 & x'_2 + x'_3 = x_1 \\ f_3(a_1,b_1) & = a_1b_1 & \rightarrow x'_3 & \end{array}, \tag{2}$$

where r is a single-bit fresh mask, a_0, a_1, b_0, b_1 are input shares, and x_0, x_1 are output shares. $f_l(.), 0 \leq l \leq 3$ are known as *component functions* whose result should be stored in registers, identified by x'_0 and x'_1. The part that XORs the registers' outputs to generate the output shares x_0 and x_1 is known as *compression layer*. It is shown in [RBN+15] that the demand for fresh randomness can be more relaxed particularly in the first-order masked implementation of quadratic functions. A methodology has been later introduced in [SM20] which avoids using fresh randomness in the first-order $d+1$ hardware implementations. We express the details of this scheme in the next section.

3 Technique

Below, we first shortly review the technique presented in [SM20] allowing to construct first-order secure implementations without any fresh randomness. Afterwards, we express our developments extending the underlying scheme to the second order.

3.1 First-Order $d+1$ Masking with no Fresh Randomness

It is shown in [SM20] that two-share first-order representation of the 2-input AND $x = f(a, b) = ab$ can be realized by four component functions $f_{0 \leq l \leq 3}(.)$, each of which receiving a combination of input shares as follow.

$$\begin{array}{lllr} f_0(a_0,b_0) = a_0b_0 & & \rightarrow x'_0 & \\ f_1(a_0,b_1) = a_0b_1 + b_1 & & \rightarrow x'_1 & x'_0 + x'_1 = x_0 \\ \hline f_2(a_1,b_0) = a_1b_0 & & \rightarrow x'_2 & x'_2 + x'_3 = x_1 \\ f_3(a_1,b_1) = a_1b_1 + b_1 & & \rightarrow x'_3 & \end{array} \tag{3}$$

Its first-order security has been guaranteed through the following observations.

- Every component function receives only one share of each input, either a_0 or a_1 and either b_0 or b_1, hence fulfilling non-completeness. Therefore, placing a probe on every gate of each component function does not leak any information about a or/and b.
- Following glitch-extended probing model, a probe placed on x_0 propagates to (x'_0, x'_1). However, simulating (x'_0, x'_1) for all possible sharings of (a_0, a_1, b_0, b_1) shows a unique joint distribution for all values of (a, b). The same holds when a probe is placed on x_1.

Further, it has been shown that (x_0, x_1) is a uniform sharing of $x = f(a, b) = ab$ if a and b are uniformly shared. Note that the above-given example is one of 16 solutions found for the 2-input AND in [SM20].

The same principle has been extended to cover up to 4-bit cubic functions allowing the authors to obtain the first-order secure implementations of coordinate functions of several 4-bit S-boxes. As the last step to construct the masked S-box, a combination of different solutions for each coordinate function should be found that fulfills the joint uniformity of the output sharing.

3.2 Extension to the Second Order

3.2.1 2-input AND

Moving toward second-order security, we start with the same simplest case, i.e., 2-input AND. Using three shares, $a : (a_0, a_1, a_2)$ and $b : (b_0, b_1, b_2)$, 9 component functions $f_{0 \leq l \leq 8}(.)$ are required to cover all 9 quadratic monomials $\forall 0 \leq i, j \leq 2, a_i b_j$. Naturally, the result of each three component functions should be compressed (after being stored in dedicated registers) to form an output share, as exemplary shown below.

$$\begin{array}{lll}
f_0(a_0, b_0) & \rightarrow x'_0 & \\
f_1(a_0, b_1) & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
f_2(a_0, b_2) & \rightarrow x'_2 & \\
\hline
f_3(a_1, b_0) & \rightarrow x'_3 & \\
f_4(a_1, b_1) & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
f_5(a_1, b_2) & \rightarrow x'_5 & \\
\hline
f_6(a_2, b_0) & \rightarrow x'_6 & \\
f_7(a_2, b_1) & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
f_8(a_2, b_2) & \rightarrow x'_8 &
\end{array} \tag{4}$$

In addition to the corresponding quadratic monomial $a_i b_j$, each component function $f_l(a_i, b_j)$ can have two other linear monomials a_i and b_j, i.e., four cases $f_l(a_i, b_j) = a_i b_j$, $f_l(a_i, b_j) = a_i b_j + a_i$, $f_l(a_i, b_j) = a_i b_j + b_j$, or $f_l(a_i, b_j) = a_i b_j + a_i + b_j$. Hence, we should search for cases whose combination fulfills the requirements for second-order security. Compared to [SM20], we need to extend the checks and examine all possible two probes which can be placed on different parts of the implementation. To this end, we constructed a procedure shown in Algorithm 1 and Algorithm 2, explaining the entire process.

We start with constructing a set $\mathcal{F}_{0,1,2}$ containing 3 component functions f_0, f_1, and f_2 that are jointly second-order secure and whose compression layer's output is balanced. As defined in Section 2.1, a Boolean function is balanced if its output yields as many zeros as ones over its input set. This is shown in lines 4 to 13 of Algorithm 1. Its second-order security is examined by three combinations of two probes: one probe at the output of the compression layer, which – under glitch-extended probing model – propagates backward to the output of registers storing the output of all three component functions, and the second probe on the input of one of such registers, which also propagates back to all inputs of the component function. For example, considering Equation (4), placing a probe on x_0 propagates to $f_0(a_0, b_0)$, $f_1(a_0, b_1)$, and $f_2(a_0, b_2)$. If the second probe is placed on the first component function $f_0(a_0, b_1)$, which propagates to a_0 and b_0, the joint distribution of $\big(a_0, b_0, f_1(a_0, b_1), f_2(a_0, b_2)\big)$ should be identical for all values of (a, b) over all possible sharings. Note that it is not required to consider $f_0(a_0, b_0)$ in the joint distributions as all its inputs a_0 and b_0 are already covered. This check and other 2-probe combinations are shown in lines 6 to 8 of Algorithm 1. The final check is the balancedness of the compression layer's output, which is an essential condition for uniform sharing of the final construction [KSM20, § 4.6]. In the rest of Algorithm 1, this process is repeated to construct two other sets $\mathcal{F}_{3,4,5}$ and $\mathcal{F}_{6,7,8}$ for other tuples of component functions, identified by lines 14 to 23 and lines 24 to 33, respectively.

The next step, shown in Algorithm 2, is to find an element in each aforementioned set, that jointly i) realize the sharing of $f(a, b) = ab$, ii) are second-order secure, and iii) form a uniform sharing for the output. In order to ease the first check, i.e., the correctness of sharing, we store the function made by each output of the compression layer. For example, in line 10 of Algorithm 1, in addition to component functions f_0, f_1, and f_2, we store $f_{0,1,2}$ in set $\mathcal{F}_{0,1,2}$. For all elements of $f_{0,1,2}$ and all elements of $f_{3,4,5}$, we calculate the XOR of the corresponding outputs of the compression layer, i.e., $f_{0,1,2} + f_{3,4,5}$. Since the desired output of the target function over input sharing, i.e., f^*, can be easily achieved by replacing a by $a_0 + a_1 + a_2$ and b by $b_0 + b_1 + b_2$ in $f(a, b) = ab$, the expecting third output of the compression layer can be calculated as $f^*_{6,7,8} = f^* + f_{0,1,2} + f_{3,4,5}$. If $f^*_{6,7,8}$ exists in $\mathcal{F}_{6,7,8}$, we already found a solution that fulfills the correctness property. In order to accelerate this process, we enumerate the ANF of such functions and use sorted arrays (or sorted link lists) to rapidly find out whether the expecting function exists in a set.

After finding a correct solution, we need to examine its second-order security. Since through Algorithm 1, we included only those component functions in each set $\mathcal{F}_{0,1,2}$, $\mathcal{F}_{3,4,5}$, and $\mathcal{F}_{6,7,8}$, that are second-order secure, we need to just examine the cases where two probes are placed on functions of different output shares. For example, one probe on output share x_0 and another one on x_1, which means examining the identical joint distribution of $\big(f_0(a_0, b_0), f_1(a_0, b_1), f_2(a_0, b_2), f_3(a_1, b_0), f_4(a_1, b_1), f_5(a_1, b_2)\big)$. Lines 6 to 8 of Algorithm 2 show this check and that of two other combinations where probes are placed on (x_0, x_2) and (x_1, x_2). In fact, many other probe combinations should be examined, where a probe is placed on an output share, e.g., x_0, and the other one on a component function of a different output share, e.g., $f_3(a_1, b_0)$ which propagates to (a_1, b_0). This means examining the identical joint distribution of $\big(a_1, b_0, f_0(a_0, b_0), f_1(a_0, b_1), f_2(a_0, b_2)\big)$ as shown in line 9 of Algorithm 2. There are 17 other such combinations that should be checked as given in lines 10 to 26. Note that we do not need to examine the combinations where both probes are placed on different parts of an output share, since they are already covered during the generation of the sets $\mathcal{F}_{0,1,2}$, $\mathcal{F}_{3,4,5}$, and $\mathcal{F}_{6,7,8}$ in Algorithm 1. Further, it is not necessary to examine the cases, where probes are placed on different component functions, as only one share of each input variable is involved in every component function, i.e., second-order non-completeness. We also do not require to examine the first-order probing security, since having a second-order probing-secure design implies its first-order security as well. If the found correct and second-order probing-secure solution forms a uniform sharing, examined in line 27 of Algorithm 2, the found solution is a valid one.

Note that the above-given procedure is dedicated to the configuration shown in Equation (4). However, there is no must to place component functions f_0, f_1, and f_2 in the compression layer of the first output share x_0. The component functions can be freely assigned to one of the output shares, but assigning more than three component functions to one output share would reduce the chance of having a valid solution since placing a probe on that output share would propagate to more than three component functions.

Algorithm 1 Search for 2nd-order 3-share rep. of 2-input quadratic function (part one)

```
Output:                                                              ▷ component functions
    $\mathcal{F}_{0,1,2} : \{(f_0(a_0,b_0),\ f_1(a_0,b_1),\ f_1(a_0,b_2), f_{0,1,2}(a_0,b_0,b_1,b_2))\}$,
    $\mathcal{F}_{3,4,5} : \{(f_3(a_1,b_0),\ f_4(a_1,b_1),\ f_5(a_1,b_2), f_{3,4,5}(a_1,b_0,b_1,b_2))\}$,
    $\mathcal{F}_{6,7,8} : \{(f_6(a_2,b_0),\ f_7(a_2,b_1),\ f_8(a_2,b_2), f_{6,7,8}(a_2,b_0,b_1,b_2))\}$

 1: for $\forall i, j \in \{0,1,2\}$ do
 2:     $\mathcal{F}_{3i+j} \leftarrow \forall\ f_{3i+j}(a_i, b_j) : \mathbb{F}_2 \times \mathbb{F}_2 \mapsto \mathbb{F}_2$          ▷ only quadratic functions
 3: end for

 4: $\mathcal{F}_{0,1,2} \leftarrow \emptyset$
 5: for $\forall(f_0, f_1, f_2) \in \mathcal{F}_0 \times \mathcal{F}_1 \times \mathcal{F}_2$ do
 6:     if $\exists\alpha; \forall a, b; P(\ a_0, b_0,\ f_1(a_0,b_1), f_2(a_0,b_2)) = \alpha$ and
 7:         $\exists\beta; \forall a, b; P(f_0(a_0,b_0),\ a_0, b_1,\ f_2(a_0,b_2)) = \beta$ and
 8:         $\exists\gamma; \forall a, b; P(f_0(a_0,b_0), f_1(a_0,b_1),\ a_0, b_2) = \gamma$ then ▷ identical joint distribution
 9:         if $f_{0,1,2}(a_0,b_0,b_1,b_2) : f_0(a_0,b_0) + f_1(a_0,b_1) + f_1(a_0,b_2)$ is balanced then
10:             $\mathcal{F}_{0,1,2} \leftarrow \mathcal{F}_{0,1,2} \cup (f_0, f_1, f_2, f_{0,1,2})$
11:         end if
12:     end if
13: end for

14: $\mathcal{F}_{3,4,5} \leftarrow \emptyset$
15: for $\forall(f_3, f_4, f_5) \in \mathcal{F}_3 \times \mathcal{F}_4 \times \mathcal{F}_5$ do
16:     if $\exists\alpha; \forall a, b; P(\ a_1, b_0,\ f_4(a_1,b_1), f_5(a_1,b_2)) = \alpha$ and
17:         $\exists\beta; \forall a, b; P(f_3(a_1,b_0),\ a_1, b_1,\ f_5(a_1,b_2)) = \beta$ and
18:         $\exists\gamma; \forall a, b; P(f_3(a_1,b_0), f_4(a_1,b_1),\ a_1, b_2) = \gamma$ then ▷ identical joint distribution
19:         if $f_{3,4,5}(a_1,b_0,b_1,b_2) : f_3(a_1,b_0) + f_4(a_1,b_1) + f_5(a_1,b_2)$ is balanced then
20:             $\mathcal{F}_{3,4,5} \leftarrow \mathcal{F}_{3,4,5} \cup (f_3, f_4, f_5, f_{3,4,5})$
21:         end if
22:     end if
23: end for

24: $\mathcal{F}_{6,7,8} \leftarrow \emptyset$
25: for $\forall(f_6, f_7, f_8) \in \mathcal{F}_6 \times \mathcal{F}_7 \times \mathcal{F}_8$ do
26:     if $\exists\alpha; \forall a, b; P(\ a_2, b_0,\ f_7(a_2,b_1), f_8(a_2,b_2)) = \alpha$ and
27:         $\exists\beta; \forall a, b; P(f_6(a_2,b_0),\ a_2, b_1,\ f_8(a_2,b_2)) = \beta$ and
28:         $\exists\gamma; \forall a, b; P(f_6(a_2,b_0), f_7(a_2,b_1),\ a_2, b_2) = \gamma$ then ▷ identical joint distribution
29:         if $f_{6,7,8}(a_2,b_0,b_1,b_2) : f_6(a_2,b_0) + f_7(a_2,b_1) + f_8(a_2,b_2)$ is balanced then
30:             $\mathcal{F}_{6,7,8} \leftarrow \mathcal{F}_{6,7,8} \cup (f_6, f_7, f_8, f_{6,7,8})$
31:         end if
32:     end if
33: end for
```

In total, there are 280 different configurations of how to assign each three component functions to one output share.

We wrote the programs in C++ to implement these algorithms and realized that there is no solution satisfying all the above-explained criteria. In fact, Algorithm 1 reports empty sets $\mathcal{F}_{0,1,2}$, $\mathcal{F}_{3,4,5}$, and $\mathcal{F}_{6,7,8}$ for any of these 280 configurations. Getting back to DOM, three fresh mask bits are used to construct the second-order probing-secure 2-input AND with three shares. Hence, we adopted our algorithms to include fresh mask bits. Namely, we require to adjust line 2 in Algorithm 1 to include linear terms (as fresh masks) to each component function. Subsequently, to construct $\mathcal{F}_{0,1,2}$, the fresh masks should be considered in lines 6 to 10 when the second-order security of construction and the

Algorithm 2 Search for 2nd-order 3-share rep. of 2-input quadratic function (part two)

```
Input: $f(a,b)$,                                        ▷ target function
       $\mathcal{F}_{0,1,2}$, $\mathcal{F}_{3,4,5}$, $\mathcal{F}_{6,7,8}$                ▷ component functions
Output: $\mathcal{F}_{0-8} : \{(f_0(a_0,b_0),\ f_1(a_0,b_1),\ \ldots,\ f_7(a_2,b_1),\ f_8(a_2,b_2))\}$

 1: $\mathcal{F}_{0-8} \leftarrow \emptyset$
 2: $f^* \leftarrow$ sharing $f(a,b)$                                ▷ direct sharing
 3: for $\forall((f_0,f_1,f_2,f_{0,1,2}),(f_3,f_4,f_5,f_{3,4,5})) \in \mathcal{F}_{0,1,2} \times \mathcal{F}_{3,4,5}$ do
 4:     ${f^*}_{6,7,8} \leftarrow f^* + f_{0,1,2} + f_{3,4,5}$
 5:     if $\exists(f_6,f_7,f_8,f_{6,7,8}) \in \mathcal{F}_{6,7,8}$ s.t. $f_{6,7,8} = {f^*}_{6,7,8}$ and     ▷ correct sharing
 6:         $\exists\alpha_0;\forall a,b; P(f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2),\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_0$ and
 7:         $\exists\alpha_1;\forall a,b; P(f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2),\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_1$ and
 8:         $\exists\alpha_2;\forall a,b; P(f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2),\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_2$ and
 9:         $\exists\alpha_3;\forall a,b; P(a_1,b_0,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_3$ and
10:         $\exists\alpha_4;\forall a,b; P(a_1,b_1,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_4$ and
11:         $\exists\alpha_5;\forall a,b; P(a_1,b_2,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_5$ and
12:         $\exists\alpha_6;\forall a,b; P(a_2,b_0,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_6$ and
13:         $\exists\alpha_7;\forall a,b; P(a_2,b_1,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_7$ and
14:         $\exists\alpha_8;\forall a,b; P(a_2,b_2,\ \ f_0(a_0,b_0),f_1(a_0,b_1),f_2(a_0,b_2)) = \alpha_8$ and
15:         $\exists\alpha_9\ ;\forall a,b; P(a_0,b_0,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_9$ and
16:         $\exists\alpha_{10};\forall a,b; P(a_0,b_1,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_{10}$ and
17:         $\exists\alpha_{11};\forall a,b; P(a_0,b_2,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_{11}$ and
18:         $\exists\alpha_{12};\forall a,b; P(a_2,b_0,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_{12}$ and
19:         $\exists\alpha_{13};\forall a,b; P(a_2,b_1,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_{13}$ and
20:         $\exists\alpha_{14};\forall a,b; P(a_2,b_2,\ \ f_3(a_1,b_0),f_4(a_1,b_1),f_5(a_1,b_2)) = \alpha_{14}$ and
21:         $\exists\alpha_{15};\forall a,b; P(a_0,b_0,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{15}$ and
22:         $\exists\alpha_{16};\forall a,b; P(a_0,b_1,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{16}$ and
23:         $\exists\alpha_{17};\forall a,b; P(a_0,b_2,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{17}$ and
24:         $\exists\alpha_{18};\forall a,b; P(a_1,b_0,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{18}$ and
25:         $\exists\alpha_{19};\forall a,b; P(a_1,b_1,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{19}$ and
26:         $\exists\alpha_{20};\forall a,b; P(a_1,b_2,\ \ f_6(a_2,b_0),f_7(a_2,b_1),f_8(a_2,b_2)) = \alpha_{20}$ then
27:         if $(f_{0,1,2}, f_{3,4,5}, f_{6,7,8})$ forms a uniform sharing then          ▷ uniform sharing
28:             $\mathcal{F}_{0-8} \leftarrow \mathcal{F}_{0-8} \cup (f_0, f_1, \ldots, f_7, f_8)$
29:         end if
30:     end if
31: end for
```

balancedness of the compression layer's output is being checked. In fact, when a probe is placed on a component function, all its inputs are probed including the fresh mask (if any). Since this process is repeated to generate $\mathcal{F}_{3,4,5}$ and $\mathcal{F}_{6,7,8}$, lines 14 to 23 and lines 24 to 33 should also be adjusted accordingly.

With two fresh masks, our programs found 156 672 solutions only for the default configuration shown in Equation (4), each of which is second-order probing-secure with uniform output sharing. One of such solutions is shown below.

$$
\begin{array}{lll|l}
f_0(a_0,b_0) & = a_0b_0 + b_0 & \rightarrow x'_0 & \\
f_1(a_0,b_1) & = a_0b_1 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
f_2(a_0,b_2,r_0) & = a_0b_2 + b_2 + r_0 & \rightarrow x'_2 & \\
\hline
f_3(a_1,b_0,r_1) & = a_1b_0 + a_1 + r_1 & \rightarrow x'_3 & \\
f_4(a_1,b_1) & = a_1b_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
f_5(a_1,b_2) & = a_1b_2 + a_1 + b_2 & \rightarrow x'_5 & \\
\hline
f_6(a_2,b_0,r_1) & = a_2b_0 + b_0 + r_1 & \rightarrow x'_6 & \\
f_7(a_2,b_1) & = a_2b_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
f_8(a_2,b_2,r_0) & = a_2b_2 + r_0 & \rightarrow x'_8 &
\end{array}
\tag{5}
$$

3.2.2 AND-XOR

As shown in [SM20], two-share first-order probing secure implementation of $f(a,b,c) = ab+c$ can be easily achieved by replacing the fresh mask bit of the masked AND implementation by c_0 and c_1. However, it is not trivially possible in the second order. Therefore, we adopted our algorithms to include one more shared input $c_{0\leq i\leq 2}$ in each component function. Since c does not contribute to any quadratic monomials, every component function can have an additional linear monomial c_0, c_1, or c_2, hence 16 cases for each component function. Staying with the default configuration (Equation (4)), our programs found 73 728 solutions for $f(a,b,c) = ab + c$, without any fresh randomness[1]. One of the solutions is given below.

$$
\begin{array}{lll|l}
f_0(a_0,b_0) & = a_0b_0 + b_0 & \rightarrow x'_0 & \\
f_1(a_0,b_1) & = a_0b_1 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
f_2(a_0,b_2,c_0) & = a_0b_2 + c_0 & \rightarrow x'_2 & \\
\hline
f_3(a_1,b_0) & = a_1b_0 + b_0 & \rightarrow x'_3 & \\
f_4(a_1,b_1) & = a_1b_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
f_5(a_1,b_2,c_1) & = a_1b_2 + b_2 + c_1 & \rightarrow x'_5 & \\
\hline
f_6(a_2,b_0,c_2) & = a_2b_0 + a_2 + c_2 & \rightarrow x'_6 & \\
f_7(a_2,b_1) & = a_2b_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
f_8(a_2,b_2) & = a_2b_2 + a_2 + b_2 & \rightarrow x'_8 &
\end{array}
\tag{6}
$$

3.2.3 Quadratic Bijections

Our findings with respect to the possibility of realizing the second-order probing-secure and uniform sharing of the AND-XOR function, motivated us to examine the applicability of our algorithms on larger yet quadratic functions. Below, borrowed from [DC07], we restate the definition of *Affine Equivalent*, which is helpful to follow the rest of the paper.

Definition 4 (Affine Equivalent)**.** Two S-boxes S and S' are affine/linear equivalent if there exists a pair of invertible affine/linear permutation A and A', such that $S = A' \circ S' \circ A$.

Every invertible affine permutation $A(X)$ can be written as $\boldsymbol{A} \cdot X + A$, where $\boldsymbol{A}$ and A are refereed to an $n \times n$ invertible matrix over $GF(2)$ and an n-bit constant, respectively. Based on Definition 4, the authors of [BNN+15] classified all 4-bit quadratic bijections in six classes, namely $\mathcal{Q}^4_4$, $\mathcal{Q}^4_{12}$, $\mathcal{Q}^4_{293}$, $\mathcal{Q}^4_{294}$, $\mathcal{Q}^4_{299}$, and $\mathcal{Q}^4_{300}$. In other words, for any 4-bit quadratic bijection F, there exist two affine functions A and A' such that $\exists i \in \{4, 12, 293, 294, 299, 300\}$, $F = A' \circ \mathcal{Q}^4_i \circ A$. Below we explain how our scheme is applied to each of such quadratic classes.

[1] For all configurations, in total we found 3 207 168 such solutions.

$\mathcal{Q}_4^4$: `0123456789ABDCFE` is the simplest one with the following coordinate functions.

$$\langle x = cd + a, \qquad y = b, \qquad z = c, \qquad t = d\rangle,$$

with $\langle a, b, c, d\rangle$ the 4-bit input, $\langle x, y, z, t\rangle$ the 4-bit output, and a and x the least significant bits. The first coordinate function is the AND-XOR, which we have studied in Section 3.2.2. However, all solutions we found for AND-XOR do not necessarily make a jointly uniform 4-bit output sharing. We found out that 533 solutions of those reported in Section 3.2.2 fulfill the joint uniformity of the output sharing of $\mathcal{Q}_4^4$. The one given in Equation (6) is one of those 533 solutions. For the sake of completeness, full details of the shared $\mathcal{Q}_4^4$ without fresh randomness is given in Appendix A.

$\mathcal{Q}_{12}^4$: `0123456789CDEFAB` has a bit more complicated ANF:

$$\langle x = a, \qquad y = bd + cd + b, \qquad z = bd + c, \qquad t = d\rangle.$$

Compared to $\mathcal{Q}_4^4$, here we need to examine combinations, where two probes are placed on the circuit associated with different coordinate functions. We start with the third output bit z which is similar to the former cases. Since the second coordinate function (generating y) has cd in its terms, all possible quadratic monomials $0 \leq i, j \leq 2, c_i d_j$ appear in its component functions. Therefore, when we are searching for a solution for the third coordinate function $f(b, c, d) = bd + c$, we can already add more checks in Algorithm 1 when we construct the set of component functions $\mathcal{F}_{0,1,2}$, $\mathcal{F}_{3,4,5}$, and $\mathcal{F}_{6,7,8}$. More precisely, in addition to 3 checks in lines 6 to 8, 9×3 other conditions are added to reflect the cases where one probe is placed on a component function of y and one probe on an output of the compression layer of z, i.e., $0 \leq \forall i, j \leq 2$,

$$\begin{aligned}
\exists\alpha; \forall b, c, d; P\big(c_i, d_j, \quad f_0, f_1, f_2\big) &= \alpha,\\
\exists\beta; \forall b, c, d; P\big(c_i, d_j, \quad f_3, f_4, f_5\big) &= \beta,\\
\exists\gamma; \forall b, c, d; P\big(c_i, d_j, \quad f_6, f_7, f_8\big) &= \gamma.
\end{aligned}$$

This way, we can strongly reduce the solutions for the third coordinate function. In fact, these extra checks result in finding solutions only for one configuration of component functions (see Equation (4)). No solution exists for the other 279 configurations. Our programs found 73 728 solutions for the third coordinate function.

By adopting the algorithms and the programs to the second coordinate function $f(b, c, d) = bd + cd + b$, we also found 3 072 solutions only for one configuration. As the last step, we need to search for a tuple of solutions for each coordinate function which i) jointly have uniform output sharing, and ii) are second-order probing secure. We have to examine all possible 2-probe combinations placed on different coordinate functions. In general, if we have n coordinate functions, we need to examine the cases where

- both probes are placed on the output of compression layers, i.e., $3 \times \binom{n}{2}$ cases, and
- one probe is placed on a component function and the other one on a compression layer's output, i.e., $9 \times 3 \times 2 \times \binom{n}{2}$ cases.

As stated, due to the second-order non-completeness of component functions, we do not need to consider cases where both probes are placed on the component functions. In total, we need to examine $63 \times \binom{n}{2}$ probe combinations, in this case 63. Among the aforementioned solutions for the second and the third coordinate functions, we found several cases[2] which pass all 63 probe-combination checks and fulfill joint uniformity of the 4-bit output sharing. One of such solutions is given in Appendix B.

[2] We did not let the program terminate as it is very lengthy, but the program found 64 such solutions after passing 2% of the entire search space.

$\mathcal{Q}^4_{293}$: `0123457689CDEFBA`'s coordinate functions are as follows.

$$\langle x = bc + a, \qquad y = bd + cd + b, \qquad z = bd + c, \qquad t = d \rangle$$

We realized that when we consider only three input variables in each component function (as each coordinate function is such), we cannot find any solution satisfying the second-order probing security when all quadratic monomials of three variables (here bc, bd, and cd) are involved. Therefore, we adjusted our programs to consider one more input variable thereby achieving second-order probing security. More precisely, considering the second coordinate function $g(b, c, d) = bd + cd + b$, we can use input shares a_0, a_1, and a_2 to fulfill the requirements for second-order probing security. However, it does mean that the found solution can easily make a joint uniform sharing with other coordinate function, particularly with the first one $f(a, b, c) = bc + a$, which depends on a as well. Since this leads to a huge number of solutions, we allowed a_0 and a_1 to be added to certain component functions to limit the valid solutions. By this, we found 14 592, 1 024 and 41 920 solutions for the first three coordinate functions, respectively. Our programs finally found several joint solutions for $\mathcal{Q}^4_{293}$ satisfying all requirements, one of which is given in Appendix C.

$\mathcal{Q}^4_{294}$: `0123456789BAEFDC` has the following coordinate functions.

$$\langle x = bd + a, \qquad y = cd + b, \qquad z = c, \qquad t = d \rangle$$

Since both first two coordinate functions are a form of AND-XOR, we easily achieved their corresponding solutions. The only condition, which should be additionally considered, is the existence of monomial bd (in the first coordinate function) when finding solutions for the second coordinate function $cd + b$. Similar to what explained for the third coordinate function of $\mathcal{Q}^4_{12}$, this extra condition forces the solutions to belong to only one configuration of component functions. As a result, we found 73 728 solutions for each coordinate function. Among them, we found thousands of joint solutions satisfying the second-order probing security (explained for $\mathcal{Q}^4_{12}$) and joint uniformity of the output sharing. One of the solutions is given in Appendix D.

$\mathcal{Q}^4_{299}$: `012345678ACEB9FD` with the following ANF has a coordinate function with four input variables.

$$\langle x = ad + cd + a, \qquad y = ad + bd + cd + b, \qquad z = bd + cd + c, \qquad t = d \rangle$$

After adjusting the programs to handle such cases as well, we found 3 072, 144 384, and 3 072 valid solutions for its coordinate functions. Note that the reason behind such a difference is that we considered only three corresponding input variables when looking for solutions for the first and third coordinate functions. If we include the missing input variable in the component functions as well, the number of found solutions would have significantly increased. Nevertheless, with the current solutions, we were able to identify one for each coordinate function which jointly fulfills all conditions. One of the found solutions is shown in Appendix E.

$\mathcal{Q}^4_{300}$: `0123458967CDEFAB` has all three possible quadratic monomials between b, c, and d in its forth coordinate function, as given below.

$$\langle x = a, \qquad y = bc + b + d, \qquad z = bc + cd + c + d, \qquad t = bc + bd + cd \rangle$$

This avoids our algorithms to find any solutions for its three-share second-order probing secure realization without fresh randomness.

3.2.4 Composition

As we have shown above, we are able to implement the identifier of all 4-bit quadratic bijective classes except $\mathcal{Q}^4_{300}$. However, it can be decomposed into two quadratic bijections

from the other classes. More concretely, $\mathcal{Q}^4_{300}$ can be written by a composition of two bijections belonging to $\mathcal{Q}^4_4 \times \mathcal{Q}^4_{12}$, $\mathcal{Q}^4_{12} \times \mathcal{Q}^4_4$, $\mathcal{Q}^4_{12} \times \mathcal{Q}^4_{294}$, and $\mathcal{Q}^4_{294} \times \mathcal{Q}^4_{12}$. As an example, we can write $\mathcal{Q}^4_{300} = \mathsf{A}_3 \circ \mathcal{Q}^4_4 \circ \mathsf{A}_2 \circ \mathcal{Q}^4_{12} \circ \mathsf{A}_1$ with affine functions
A_1:`014589CD2367ABEF`: $\langle x = a, \quad y = d, \quad z = b, \quad t = c\rangle$,
A_2:`02DF469B8A57CE13`: $\langle x = b, \quad y = a, \quad z = b + c, \quad t = b + d\rangle$, and
A_3:`08192A3B4C5D6E7F`: $\langle x = b, \quad y = c, \quad z = d, \quad t = a\rangle$.

Hence, we are able to compose the descriptions given in Appendix A and Appendix B. However, we should emphasize that the composition of such designs does not necessarily lead to a second-order probing-secure implementation. As stated in [Rep15], when probes are placed of composed functions, there is no guarantee to maintain the higher-order security, while each function is individually higher-order secure. Hence, we have to refresh the signals traversing between the functions.

By integrating both A_1 and A_2 in $\mathcal{Q}^4_{12}$, and A_3 in $\mathcal{Q}^4_4$, we can write $\mathcal{Q}^4_{300} = \mathsf{G} \circ \mathsf{F}$ with F:`02468A13DF9BCE57` as

$$\langle x = bc + cd + d, \qquad y = a, \qquad z = bc + b + d, \qquad t = bc + cd + c + d\rangle,$$

and G:`08192A3B4C5DE6F7` which is $\mathcal{Q}^4_4$ with permuted outputs. When giving the shared output of F as the input to G, the following rules should be carefully followed.

- Every output share $\langle x_0, x_1, x_2\rangle$ should be refreshed by two individual fresh mask bits r_0 and r_1 as $\langle x_0 + r_0, x_1 + r_1, x_2 + r_0 + r_1\rangle$. This also holds for those outputs which directly come from the input shares if it participates in a coordinate function. For example, the second output bit of F, i.e., $y = a$, does not need to be refreshed since a is not involved in any coordinate function of F. However, if $\mathcal{Q}^4_{299}$ is composed, its forth output share $t = d$ should also be refreshed since d is involved in its other coordinate functions.

- The refreshing should be performed together with the compression layer, i.e., where the output of the component functions (stored in register) are XORed to make the output shares.

- The output of the compression layer should be stored in a register before being given to the next function. Otherwise, a probe placed on the component function of the next function propagates backward to the compression layer and hence to the output of several component functions, which may make the implementation vulnerable even to first-order attacks. This has been discussed in detail in [SM20].

In short, we need 6 fresh mask bits and 3 register stages to realize a second-order probing-secure implementation of $\mathcal{Q}^4_{300}$. Full description of the component functions and how they are connected together is given in Appendix F.

We should mention that we also unsuccessfully tried to extend our algorithms to cover cubic monomials. Since the number of component functions increases from 3 to 9 for each output share, the search space explodes and the chance of finding a second-order probing-secure construction becomes low while each probe placed on a compression layer propagates to the output of 9 component function. Nevertheless, following the comprehensive study conducted in [BNN$^+$15], the 4-bit cubic S-box of all lightweight ciphers can be decomposed to quadratic bijections (in 2 or 3 stages), each of which is affine equivalent to one of the above-explained classes. Hence, we are able to construct their second-order probing-secure implementation, while fresh randomness is required only between their connection. We give more detail when dealing with some of such S-boxes in the next session.

4 Case Studies

In this section, we express some case studies to highlight the benefits and difficulties of the application of our technique on different symmetric cryptographic primitives.

4.1 Keccak

We first focus on Keccak [BDPA13], where a 5-bit S-box is used, called χ function. Each of its coordinate functions is a quadratic Boolean function with three input bits. More precisely, all of them are the AND-XOR that we have discussed in Section 3.2.2, where one of the AND operands is complemented. The ANF of the coordinate functions is given below.

$$\langle x = de + a + d, \quad y = ae + b + e, \quad z = ab + a + c, \quad t = bc + b + d, \quad w = cd + c + e \rangle,$$

where $\langle a, b, c, d, e \rangle$ and $\langle x, y, z, t, w \rangle$ are the 5-bit input and output, respectively.

Looking at the state of the art, a first-order $td + 1$ masked implementation of Keccak with three shares is first given in [BDPA10], whose output sharing is not uniform; one approach to achieve uniformity is to use fresh randomness. A uniform first-order $td + 1$ solution with four shares was then introduced in [BDN+14], which fulfills all requirements without fresh randomness. The trick known as "changing of the guards" was afterwards introduced in [Dae17], that overcomes the non-uniformity of the design in [BDPA10] by re-using the shares of the i-th S-box instance as the fresh mask for the $i + 1$-th S-box, hence not requiring fresh randomness in each clock cycle.

The first $d + 1$ masked Keccak with two shares is given in [GSM17a] with DOM as the underlying technique. Although each DOM AND operation needs a fresh mask bit (see Equation (2)), due to the AND-XOR nature of χ's coordinate functions, the additive variable is used to blind the AND, and finally a two-share Keccak without fresh randomness is presented in [GSM17a]. A security flaw in its implementation (with respect to the location of registers), and two-round first-order secure implementations with five (and six) shares are reported in [ABP+18], which do not make use of any fresh randomness. We also have observed that the implementation given in [GSM17a] does not maintain the uniformity of the χ's output sharing. Although the authors claim that the security loss is negligible [Dae16], for the sake of completeness, we give a solution with uniform output sharing in Appendix G by applying the technique presented in [SM20]. We indeed found 274 924 such solutions.

To the best of our knowledge, the only second-order secure Keccak is given in [GSM17a], where each AND-XOR operation is masked following the second-order DOM multiplier, i.e., 3 fresh mask bits per coordinate function, hence 15-bit fresh randomness per 5-bit S-box of the χ function. As stated in Section 3.2.2, for an AND-XOR, we found 73 728 solutions without fresh randomness. However, considering all 5 coordinate functions, they are not necessarily jointly uniform or second-order probing secure. To reduce the search space, we employed the same technique explained in Section 3.2.3. For instance, the first coordinate function (generating x) receives $\langle a, d, e \rangle$ as the input, where de is the only quadratic monomial. The term ae exists in the ANF of the second coordinate function (generating y), and the monomial ad does not show up in the ANF of the other coordinate functions. Hence, when we search for a solution for the first coordinate function, we added extra checks to consider a probe on each possible quadratic monomial $0 \leq \forall i, j \leq 2, a_i e_j$. Due to χ's rhythmic ANF pattern, i.e., XORing each bit with an AND result of two other adjacent bits in its row, the same technique can be applied to the other coordinate functions. Namely, we add extra conditions to put a probe on each component function of the coordinate function $i + 1 \mod 5$ when searching for a solution for the i-th coordinate function. In this way, we reduced the number of solutions to 24 for each coordinate function. Note that by such extra conditions, the solutions can be found for only one configuration

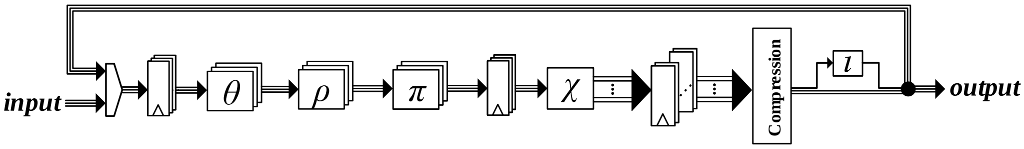


Figure 1: Design architecture of our round-based second-order Keccak.

of the component functions (see Section 3.2.1). Finally, by searching through the found solutions, we identified 659 cases which jointly fulfill all requirements for second-order probing security and uniform output sharing. One of such cases is given in Appendix H.

Based on our S-box constructions, we have designed a two-share and a three-share round-based implementation of Keccak [1088,512] permutation without any fresh masks. It is one of the SHA3 standards and allows to provide a fair comparison to the state of the art. The underlying design architecture is shown in Figure 1. As shown, we placed a register at the input of the θ transformation. Alternatively, it can be placed at the output of the compression layer. This is essential since θ combines (XOR) every two neighboring output bits of each 5-bit S-box of the χ function. Without such a register, a probe placed on the XORs of θ propagates backwards to two outputs of the compression layer. Then, a second probe can be easily found to show a second-order leakage. Further, a register at the input of the χ function is essential to not violate the requirements for the security under the glitch-extended probing model [ABP+18]. As a comparison to the state of the art, we refer to Table 1 where our first-order secure design is the only one which i) uses two shares, ii) does not require any fresh masks, iii) and has uniform output sharing. Note that the designs presented in [GSM17a] suffer from non-completeness issue as addressed in [ABP+18]. Afterwards, the implementations were modified and the results were updated in [GSM17b]. Hence, we compare only with the corrected implementations. Regarding the second order, our construction outperforms the only-previously-published one in terms of randomness complexity and area overhead, as shown in Table 1. Note that the given performance results are excluding PRNGs required to generate the fresh masks, but still our design, which does not use any fresh masks, needs less area footprint. In order to be compatible with the state of the art, we synthesized our designs using UMC 130 standard cell library.

One more important fact to discuss is multivariate leakages between two consecutive rounds. We are not refreshing the χ output, which goes through the diffusion layer (θ, ρ, and π) and is given to the next χ function. A question is whether anything can be gained by placing two probes on two χ operations in consecutive rounds. As a general rule, when two second-order secure functions are composed, fresh masks are required at their conjunction, as illustrated in Section 3.2.4. Our observation is that if there is

Table 1: Performance figures of round-based Keccak [1088,512] permutation. (using Synopsis Design Compiler, and UMC 130 standard cell library, excluding PRNGs)

Design	Security Order	No. of Shares	Fresh Masks/ S-box [bit]	Area [kGE]	Delay [ns]	Latency [cycles]
[BDN+14]	1	3	2	135.2	1.34	25
[BDN+14]	1	4	0	157.6	1.36	24
[GSM17b]	1	2	0	111.8	1.19	72
this work	1	2	0	129.3	1.29	72
[GSM17b]	2	3	15	238.4	1.19	72
this work	2	3	0	231.5	1.50	72

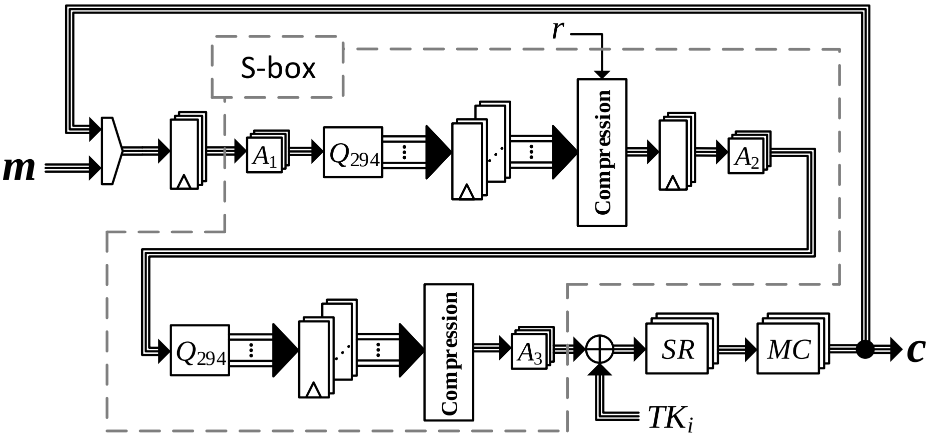


Figure 2: Design architecture of our round-based second-order SKINNY-64 encryption function.

a strong diffusion layer between such compositions, no mask refreshing is required. In case of Keccak, independent of the bit permutations ρ and π, each output bit of θ is the XOR result of 11 bits; 9 of them are taken from the output of 9 different 5-bit S-box instances. We expect that every probe placed on the second χ function observes a distribution independent of any other probe placed on the first χ function. Note that it is just our observation confirmed by practical experiments expressed in Section 5. We further should highlight that no verification tool is yet able to evaluate full cipher implementations; hence we cannot provide any proof for this observation.

4.2 SKINNY

The 4-bit S-box of SKINNY [BJK+16]: `C6901A2B385D4E7F` belongs to the cubic class $\mathcal{C}^4_{223}$ which can be decomposed as $A_3 \circ \mathcal{Q}^4_{294} \circ A_2 \circ \mathcal{Q}^4_{294} \circ A_1$. Among 262 144 ways for the decomposition, we identified a case with the simplest affine functions as
A_1:`FEBA7632DC985410`: $\langle x = a + 1, \quad y = d + 1, \quad z = b + 1, \quad t = c + 1\rangle$,
A_2:`084C2A6E195D3B7F`: $\langle x = d, \quad y = c, \quad z = b, \quad t = a\rangle$, and
A_3:`FDECB9A875643120`: $\langle x = b + 1, \quad y = a + 1, \quad z = c + 1, \quad t = d + 1\rangle$,
which are just bit permutations and negation of input/output variables. Therefore, the solution given for $\mathcal{Q}^4_{294}$ in Appendix D can be directly used here. Note that two output bits of $\mathcal{Q}^4_{294}$ are directly connected to its inputs (see its ANF in page 720), but since they both are involved in the coordinate function of the other output bits, all outputs of the first masked $\mathcal{Q}^4_{294}$ should be refreshed and stored in registers before being given to the second masked $\mathcal{Q}^4_{294}$. Therefore, we made the second-order probing-secure and uniform sharing of the SKINNY 4-bit S-box in 3 register stages using 8 fresh mask bits.

In order to construct a round-based secure implementation of the SKINNY-64, placing a state register at the output of the S-box is not necessary (see Figure 2). That is because the diffusion layer of SKINNY, including `AddRoundTweakey (ART)`, `ShiftRows (SR)`, and `MixColumns (MC)`, does not mix different output bits of any S-box. More precisely, each output bit of `MC` is the XOR of at most three bits that belong to three different S-boxes. However, as explained in the case of Keccak, a register stage at the input of the S-box is essential, which is used in our design as the state register. Note that here we do not need to place any register because of the affine functions A_1, A_2, and A_3, since – as stated – they are just bit permutation and negation. The diffusion layer of the SKINNY round function is not as strong as that of the Keccak. For example, one row of the cipher state passes through `MC` unchanged. Therefore, our argument with respect to multivariate leakages across consecutive cipher rounds, given for Keccak, is not valid here. However, ART can be beneficial here. Let us denote the output of SubCells (SC) by (A, B, C, D), where each element corresponds to a row, here 4 nibbles. Ignoring SR, which permutes the nibbles of each row, the output of MC can be written as

$$(A + D + C + K, \qquad A + K, \qquad B + C + K', \qquad C + A + K),$$

where (K, K') denote the 32-bit round tweakey represented in two rows. It can be seen that – except the second row – each input bit of any S-box in the next round is the XOR result of at least two output bits belonging to different S-boxes. If the round tweakkey is presented in a second-order masked form, i.e., with three shares (as in our design in Figure 2, it plays the role of the fresh mask and blinds the second row $A + K$. Therefore, it is essential to apply key masking, i.e., the key schedule should also be masked with three shares.

In summary, our fully-pipeline round-based implementation has four register stages and requires 8 × 16 fresh mask bits per clock cycle. Table 2 shows the corresponding performance figures. We constructed SKINNY-64-64 encryption function, i.e., with a 64-bit key; the other variants with larger keys can be easily constructed since the SKINNY key schedule is a linear function. Further, due to the lack of higher-order implementation of SKINNY in the open literature, we could not find any other design for comparison.

4.3 Midori

Midori's 4-bit S-box S:CAD3EBF789150246 [BBI+15] is affine equivalent to the identifier of the class $\mathcal{C}^4_{266}$. Among several ways to decompose it to quadratic bijections, we selected the case as $S = A_3 \circ \mathcal{Q}^4_{12} \circ A_2 \circ \mathcal{Q}^4_{12} \circ A_1$ with
A_1:93821B0AF5E47D6C: $\langle x = b + 1, \quad y = a + d, \quad z = d, \quad t = a + c + 1\rangle$,
A_2:08C43BF719D52AE6: $\langle x = c + d, \quad y = c, \quad z = b \quad t = a + b\rangle$, and
A_3:FD75A820EC64B931: $\langle x = c + d + 1, \quad y = a + 1, \quad z = c + 1, \quad t = b + 1\rangle$.
Integrating A_1 into the first $\mathcal{Q}^4_{12}$ would lead to having all quadratic monomials of three input variables in a coordinate function, and we would face similar difficulty observed for $\mathcal{Q}^4_{300}$. In general, we prefer the decompositions with a simple input affine A_1, and would

Table 2: Performance figures of different implementations, including key masking. (using Synopsis Design Compiler, and UMC 90 standard cell library, excluding PRNGs)

Design	Security Order	No. of Shares	Fresh Masks/ S-box [bit]	Area [kGE]	Delay [ns]	Latency [cycles]	Throughput [MB/s]
SKINNY-64-64							
this work	2	3	8	10.6	1.22	128	204.9
Midori-64							
this work	2	3	8	15.5	2.86	64	174.8
PRESENT-80							
[CBRN14][a]	2	5	520	8.3	-	149	-
this work	2	3	8	3.8	2.04	666	5.8
PRINCE							
this work	2	3	8	19.4	3.11	84	214.3
[BKN19][b]	2	3	18	13.4	4.00	72	27.7
[BKN19][b]	2	5	10	18.7	4.10	72	27.1
[BKN19][b,c]	2	3	108	32.4	3.42	24	194.9
[BKN19][b,c]	2	8	88	177.6	3.54	24	188.3

[a] just an S-box and using NanGate 45

[b] using TSMC 90

[c] without S-box decomposition

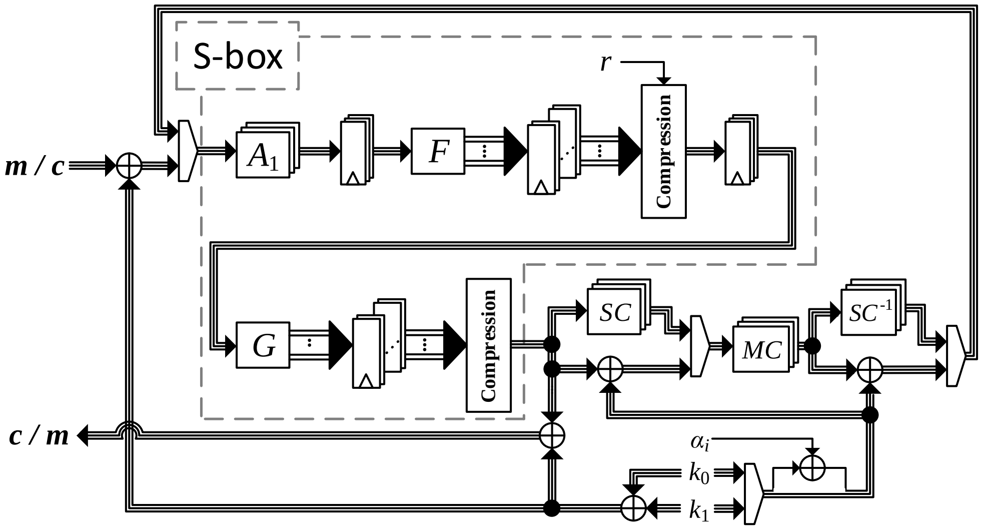


Figure 3: Design architecture of our round-based second-order Midori-64 encryption/decryption function.

integrate the middle and output affine functions A_2 and A_3 at the output of the quadratic functions. More precisely, we write $S = G \circ F \circ A_1$ with $F = A_2 \circ \mathcal{Q}^4_{12}$:08C43BF7192AE6D5 as

$$\langle x = bd + c + d, \qquad y = bd + c, \qquad z = bd + cd + b, \qquad t = bd + cd + a + b \rangle,$$

and $G = A_3 \circ \mathcal{Q}^4_{12}$:FD75A820ECB93164 as

$$\langle x = bd + c + d + 1, \qquad y = a + 1, \qquad z = bd + c + 1, \qquad t = bd + cd + b + 1 \rangle.$$

An important point to mention is that the output of the A_1 should be stored in a register before it is connected to the input of the shared F. Otherwise, the non-completeness would be violated. Further, due to the existence of the input affine, we have to consider more checks when we search for solutions for each coordinate function of F. More precisely, a probe can be placed on the XOR of the input affine, and another probe on component functions or compression layer of F. In order to cover any affine function placed at the input of F, we consider three cases, where the probe is propagated to all input variables of the same share index, i.e.,

$$(a_0, b_0, c_0, d_0), \qquad (a_1, b_1, c_1, d_1), \qquad (a_2, b_2, c_2, d_2).$$

In other words, similar to what explained as extra conditions for $\mathcal{Q}^4_{12}$ in page 719, the combination of each of these propagated probes and a probe placed on every output share should be considered in lines 6 to 8 of Algorithm 1, i.e., 3 × 3 extra conditions for each coordinate function. This way, we make sure that any affine function, placed at the input of the target function, would not violate the second-order probing-security of the implementation. By adjusting our programs and considering these extra checks for F, we found several solutions for each F and G, while one of them is given in Appendix I.

The design architecture of our fully-pipeline round-based second-order Midori-64 supporting both encryption and decryption is depicted in Figure 3, which is similar to the one presented in [MS16a]. As stated, 8-bit fresh masks should be used for the composition of F and G, which is integrated into the compression layer of F whose results should be stored in registers. As a result, the second-order secure Midori's S-box needs 4 register layers and 8-bit fresh masks. Note that we do not need any further registers to implement the cipher as one of the register stages can be seen as the state register, and Midori's `MixColumns (MC)`

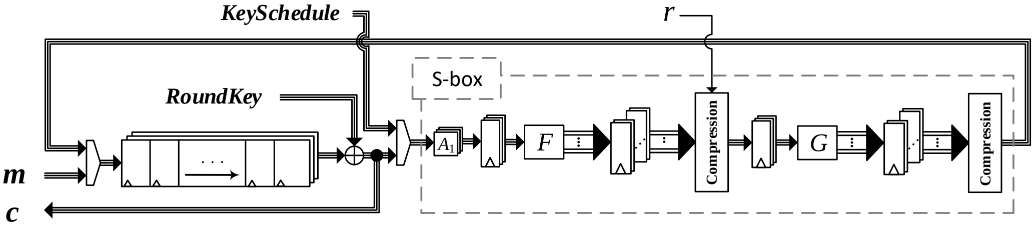


Figure 4: Design architecture of our nibble-serial second-order PRESENT encryption function (permutation layer and key schedule not shown).

only mixes the output bits of different S-boxes, similar to Skinny. Namely, when a probe is placed on MCs' output bit, it propagates to output bits of different S-boxes that are statistically independent. Regarding multivariate leakages between two consecutive rounds, the argument is similar to the one given for Keccak and SKINNY. Each MC's output bit is the XOR of three different bits of different S-boxes, and together with key masking, this blinds the S-box outputs that are given to the next round function avoiding second-order leakages. Note that, these arguments are valid since each S-box itself (including several stages) is second-order probing secure. The synthesis result of our design is also involved in Table 2. Notably, our design seems to be the only second-order implementation of Midori-64 in the literature.

4.4 PRESENT

Similar to Midori, the PRESENT S-box [BKL+07] S:C56B90AD3EF84712 belongs to the cubic class $\mathcal{C}^4_{266}$. Therefore, we follow the same principle and express it as $S = G \circ F \circ A_1$ with
A_1:894501CDAB6723EF: $\langle x = a, \quad y = d, \quad z = b, \quad t = b + c + 1\rangle$,
F:08C43BF7192AE6D5 as

$$\langle x = bd + c + d, \qquad y = bd + c, \qquad z = bd + cd + b, \qquad t = bd + cd + a + b\rangle,$$

and G:9C3672D805EB41AF as

$$\langle x = a + d + 1, \qquad y = cd + b + c, \qquad z = bd + a + c, \qquad t = cd + b + c + d + 1\rangle,$$

where both F and G are affine equivalent to the quadratic class $\mathcal{Q}^4_{12}$. As a matter of chance, here F is the same as that of Midori. Therefore, we only give the sharing of G in Appendix J. Note that all given statements with respect to the input affine A_1 and fresh randomness given for the S-box of Midori hold valid here as well. In summary, we provide a three-share second-order probing-secure realization of the PRESENT S-box with uniform output sharing in 4 register stages making use of 8-bit fresh randomness.

Most of the implementations of PRESENT reported in the literature follow a serialized architecture, where a single S-box is instantiated and shared with the key schedule as well. Staying with the same fashion, we took the design of PRESENT-80 presented in [PMK+11] and plugged our S-box as shown in Figure 4. At each clock cycle, the state- and the key-registers are shifted nibble-wise and feed the S-box (16 clock cycles by the state and one clock cycle by the key schedule). After 20 clock cycles, when sBoxLayer is accomplished, in one clock cycle the permutation pLayer performed and the key schedule is finalized.

Regarding multivariate leakages across two consecutive rounds, we should have a closer look at AddRoundKey and the pLayer. Since pLayer is just a bit permutation, the 4-bit input of any S-box of the next round in a concatenation of 4 output bits of 4 different S-boxes in the previous round. Supposing that each S-box is individually shared, the 4 input bits of any S-box in the next round are independently shared. Moreover, by applying key masking the shares of the input of every S-box is again refreshed.

As a comparison to the state of the art, a second-order secure PRESENT S-box is presented in [CBRN14] in which polynomial masking is employed as the underlying masking scheme. Looking at the performance results in Table 2, the randomness complexity and latency of their masked S-box are extremely higher than our design.

4.5 PRINCE

Both S-box S:`BF32AC916780E5D4` and its inverse S^{-1}:`B732FD89A6405EC1` are used in encryption as well as decryption of PRINCE [BCG+12]. Based on the study published in [MS16a], the S-box belongs to the cubic class $\mathcal{C}^4_{223}$ which cannot be decomposed to two quadratic bijections of those classes that we cover [BNN+15]. Following the decomposition given in [MS16a] for the S-box inverse, we write $S^{-1} = H \circ G \circ F \circ A_1$ with
A_1:`8293C6D70A1B4E5F`: $\langle x = b, \quad y = a \quad z = c, \quad t = a + d + 1\rangle$,
F:`C480E6A2D519B37F` as

$$\langle x = d, \qquad y = c, \qquad z = cd + b + 1, \qquad t = bd + a + 1\rangle,$$

G:`08C43BF72A6ED591` as

$$\langle x = c, \qquad y = c + d, \qquad z = cd + b, \qquad t = bd + cd + a + b\rangle,$$

and H:`21748BDE65039AFC` as

$$\langle x = bd + cd + a + b, \qquad y = bd + a + c + 1, \qquad z = cd + b + d, \qquad t = c\rangle.$$

Using our programs adjusted to these coordinate functions, we found several solutions for each F, G, and H satisfying all requirements to be second-order probing-secure with uniform output sharing. One of such solutions is given in detail in Appendix K. Note that the S-box and its inverse are affine equivalent as $S = A \circ S^{-1} \circ A$ with
A:`B8A93021EDFC6574`: $\langle x = a + b + d + 1, \quad y = a + 1 \quad z = d, \quad t = c + 1\rangle$.
As stated in Section 4.3, we considered those extra checks with respect to the input affine when constructing the component functions. Therefore, placing A at the start of S^{-1} would not violate its second-order security. Since we split the S-box inverse into three quadratic parts, and we should refresh when the functions are composed, our construction is in 6 register stages and needs 16 fresh mask bits for each S-box/inverse calculation. It is important to recall that if a pipeline design is made, the 8-bit fresh mask required for $G \circ F$ and those required for $H \circ G$ can be connected to the same source, i.e., 8-bit fresh randomness per clock cycle. We give more details about the security of this optimization in Appendix L.

In order to construct a secure implementation of the cipher, an extra register layer should be placed at the output of the S-box inverse due to the affine function A at the end of S^{-1}. Figure 5 depicts the design architecture of our fully-pipeline round-based second-order PRINCE supporting both encryption and decryption. Similar to Midori's `MC`, each bit of the output of the M'-layer is the XOR result of three output bits of different S-boxes with independent sharing. combined with key masking, the same arguments, given with respect to avoiding multivariate leakages across two consecutive rounds, hold here as well. In summary, our fully-pipeline design has 7 register layers per cipher round and needs 8 × 16 fresh mask bits per clock cycle.

We are aware of one work dealing with second-order masked hardware implementation of PRINCE presented in [BKN19]. The authors decomposed the S-box inverse into quadratic bijections, where all of them belong to $\mathcal{Q}_{294}$ class. They provided 5-share and 3-share second-order masked implementation of $\mathcal{Q}_{294}$ using 10 and 18 bits fresh masks (per clock cycle) and made a loop over it with different affine functions to realize either the S-box or its inverse. As one can see in page 725, our construction has less randomness complexity

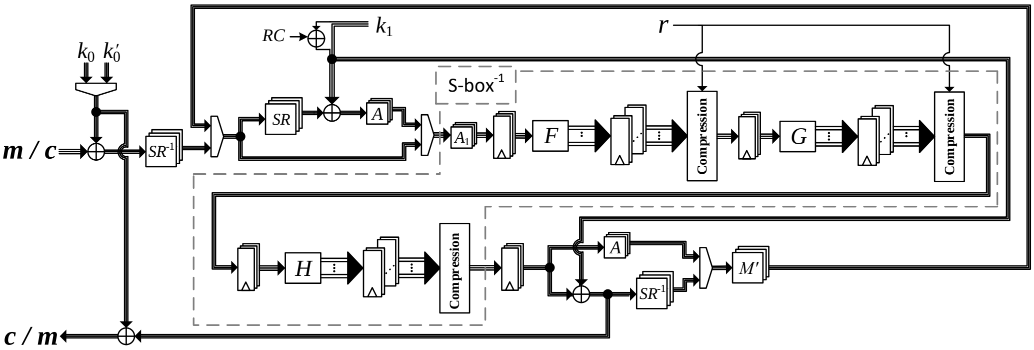


Figure 5: Design architecture of our round-based second-order PRINCE encryption/decryption function.

but higher area overhead due to its fully-pipeline architecture leading to higher throughput. The authors also presented two second-order secure designs without S-box decomposition. Looking at Table 2, with even lower throughput, the randomness complexity and area overhead of their designs are way higher than ours. Note that in order to be more fair in the shown comparison, we synthesized our designs by UCM 90 standard cell library. The performance figures in [BKN19] are based on TSMC 90 which is out of our access.

5 Analysis

As the first analysis step, we employed SILVER [KSM20] to examine our S-box constructions under glitch-extended probing model, dedicated to masked hardware designs. SILVER is a formal verification tool, developed to check the design based on the proofs to avoid writing the proofs for every design. It receives the gate-level netlist of a hardware design and reports the result of evaluations based on the security notions defined in different articles like [MBR19]. Since SILVER does not simplify anything, its analysis results are reliable (without false positive or false negative). To this end, we synthesized the HDL code of our S-box designs (also given in the GitHub) and supplied SILVER with the resulting netlist. For our entire designs, SILVER reported robust-probing security up to second order as well as the uniformity of output sharing. Since no verification tool (including SILVER) is yet able to analyze full cipher implementations, similar to the state of the art, we additionally conducted FPGA-based experimental analyses, as given in detail as follows.

5.1 Setup

We implemented our designs expressed in Section 4 on the target Spartan-6 FPGA of the SAKURA-G board [SAK]. We collected power consumption traces by monitoring the voltage drop over a 1 Ω resistor placed in the Vdd path of the target FPGA using a digital oscilloscope at sampling rate of 500 MS/s. During the measurements, the target FPGA were supplied by a stable clock source at the frequency of 6 MHz. The target FPGA receives masked input (plaintext) and issues output (ciphertext) also in the same sharing form. The fresh mask bits (if needed) are generated on the fly inside the target FPGA by means of 31-bit LFSRs optimized for Xilinx FPGAs [DMW18]. For each mask bit, we instantiated one LFSR seeded at random right after the power-up of the FPGA.

For each design, we collected 100 million traces following the strategy explained in [GJJR11] to conduct reliable fixed-versus-random t-test. We further followed the techniques presented in [SM15] to efficiently perform t-tests at higher orders.

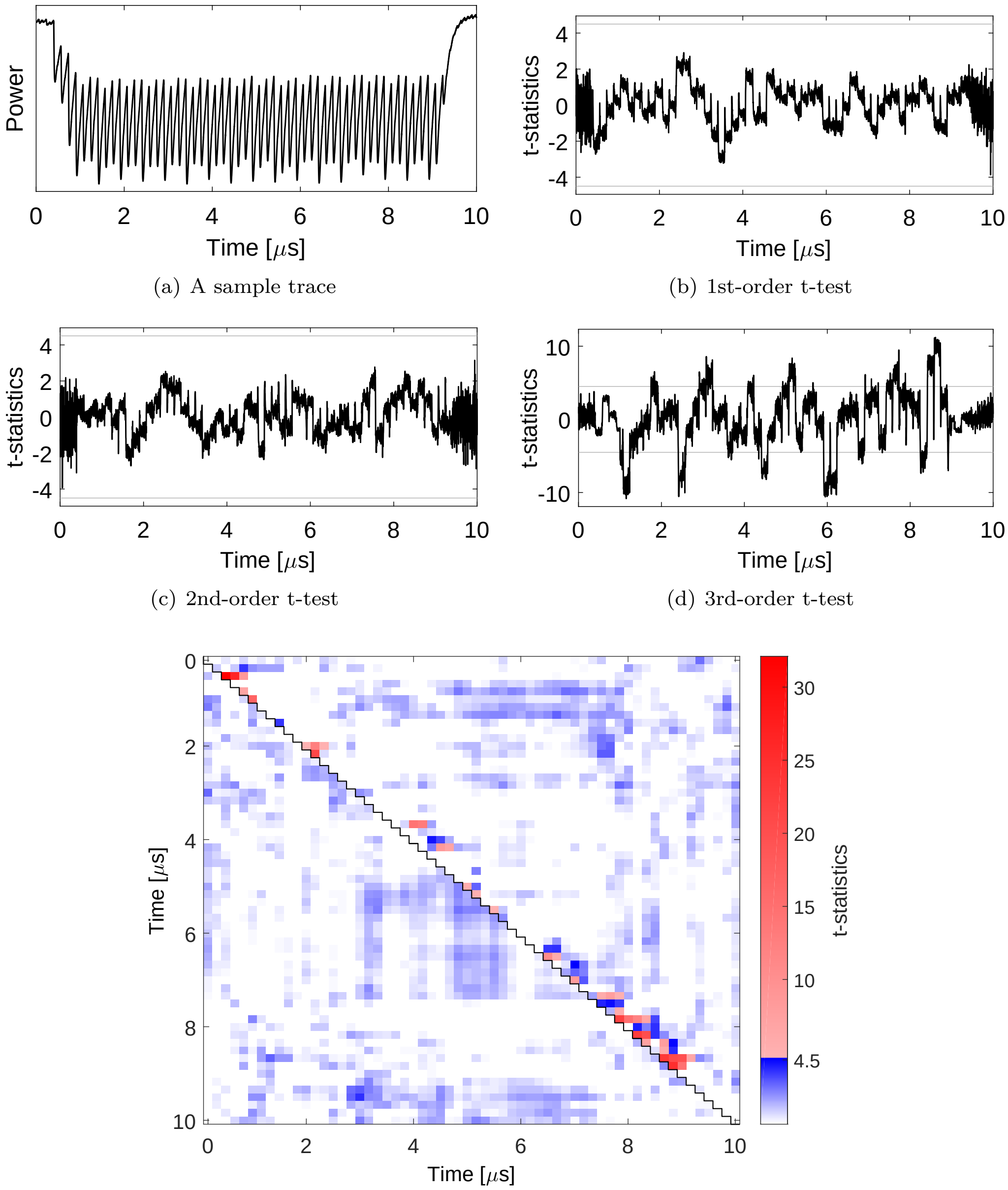
5.2 Results

We start with our Keccak design, given in Section 4.1. Due to the large size of the Keccak instance benchmarked in Table 1, which hardly fits into our target FPGA, similar to [ABP+18] we practically evaluated a smaller variant, i.e., Keccak-f[200]. Note that χ function, whose protection is the difference between the state of the art, is the same in all Keccak variants. It instantiates the 5-bit S-box a different number of times. Figure 6 shows a sample power trace and the result of up to third-order univariate t-tests indicating no detected leakage up to second order. Since we also aim at evaluating our design with respect to multivariate leakages (i.e., the combined sample points are taken from different clock cycles), we performed bivariate t-test as formulated in [SM15]. Here, each power trace contains 5 000 sample points translating to $5\,000 \times (5\,000 + 1)/2 = 12\,502\,500$ individual t-tests, which take around 30 days to accomplish using all cores of a 24-CPU machine running (at most) at 2.93 GHz. Due to the same difficulty, such analyses are usually done on a small part of the traces downsampled, e.g., by covering only one S-box calculation and dividing the sampling rate (e.g., by 4 as done in [CRB+16]). Power consumption traces are inherently low-pass filtered by the Printed Circuit Board (PCB), shunt resistor, the chip package, and the measurement equipments [MOP07] Hence, several sample points in each clock cycle of power traces (close to the power peak, i.e., clock edge) contain the same information about the leakage at that clock cycle (see [MM13] for relevant information). Therefore, considering the power peak at each clock cycle should be adequate for such a bivariate analysis. Therefore, instead of decreasing the sampling frequency (which should be synchronous with the device clock at very low sampling rates [OC15]), we extracted one sample (power peak) per clock cycle for the bivariate analyses, but covered the entire clock cycles involved in the power traces. The results shown at the left side of Figure 6(e) are inline with our expectations, i.e., no detected bivariate second-order leakage. However, in order to verify our bivariate setup, we also implemented a first-order version of the same Keccak variant. To this end, we took the two-share description of the χ function given in Appendix G, and performed the same bivariate analysis. The corresponding results depicted at the right side of Figure 6(e) confirm the correctness and ability of our setup to detect such bivariate leakages.

We conducted exactly the same analyses on all our other designs, all of which actually lead to the more of less similar results (given in Appendix M), and with the same conclusion, i.e., no first- and second-order univariate and bivariate leakage detected. We should highlight that except in the case of our PRESENT implementation, which follows a serialized architecture, in all our measurements we cover the entire calculation of the algorithm (can also be recognized from the shown sample power traces). For the PRESENT design, we cover only the first half of the encryption, which is already 300 clock cycles.

6 Discussions and Conclusions

In this work, we have introduced a methodology to achieve three-share second-order secure implementation of a group of quadratic functions without any fresh randomness. Naturally, by composing such designs we can realize larger constructions. However, refreshing the sharing of the interconnections is inevitable for higher-order security. We showed that having a quadratic round function with a strong diffusion layer, e.g., in Keccak, allows us to realize the second-order secure implementation of the cryptographic primitive without any fresh randomness. Although it is not the case for even lightweight ciphers with a 4-bit S-box, their second-order secure implementations require fresh masks only for composition, i.e., 8 bits per S-box and per clock cycle. To the best of our knowledge, our constructions outperform state-of-the-art implementations with respect to area, throughput, and demand for fresh randomness. More importantly, evaluations based on SILVER [KSM20] confirm



(a) A sample trace

(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

(e) 2nd-order bivariate t-test, (left) second-order and (right) first-order design

Figure 6: Experimental analysis of our second-order secure Keccak-f[200] design, 100 million traces.

the second-order security of our designs under glitch-extended probing model.

Naturally, the most interesting and useful case would be the application of our technique on the AES S-box. It is for sure among our future works, but it seems challenging as the entire operations of the S-box should be represented by quadratic functions. This would lead to a high number of compositions and consequently a high number of registers as well as fresh masks. Therefore, it needs intensive research to cope with such difficulties to outperform the state of the art.

Reduction of the number of required fresh masks per cipher round (due to the composition) is of interesting topics as well. We should refer to the relevant study [BDZ20], which addresses some interesting facts with respect to reusing the masks to refresh the shares between consecutive cipher rounds in higher-order $td + 1$ masked designs. Further, it has been stated in [BKN19], that when each S-box requires n-bit fresh masks, having 16 S-boxes in a second-order implementation of PRINCE, $4n$ fresh mask bits are adequate to apply instead of $16n$ bits, i.e., reusing the fresh masks. We should refer to “changing of the guards” [Dae17] which also tries to reuse the shares of irrelevant cipher states as fresh masks in first-order $td + 1$ masked implementations. Clearly, this topic needs more research and investigations particularly for higher orders.

In all these research activities, the goal is to avoid or reduce the required fresh masks. As given in the performance figures (Table 1 and Table 2), similar to the state of the art, we exclude the PRNGs necessary to generate the fresh masks. A fundamental question, which is not yet answered and needs proper attention, is how expensive it is to generate a certain number of fresh masks per clock cycle. As the cost function, area, energy, power, and latency are certainly the possible choices.

Acknowledgments

The work described in this paper has been supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972 and through the project 406956718 SuCCESS.

References

[ABP$^+$18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic Keccak: SCA Security and Low Latency in HW. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):269–290, 2018.

[BBC$^+$19] 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 2019*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBI$^+$15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In *ASIACRYPT 2015*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BCG$^+$12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[BDN$^+$14] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2014.

[BDPA10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of Keccak. In *Second SHA-3 candidate conference*, 2010.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BDZ20] Tim Beyne, Siemen Dhooghe, and Zhenda Zhang. Cryptanalysis of Masked Ciphers: A Not So Random Idea. In *ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 817–850. Springer, 2020.

[BFG$^+$17] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, Clara Paglialonga, and François-Xavier Standaert. Consolidating inner product masking. In *ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 724–754. Springer, 2017.

[BGN$^+$14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-Order Threshold Implementations. In *ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2014.

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight Private Circuits: Achieving Probing Security with the Least Refreshing. In *ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 343–372. Springer, 2018.

[BJK$^+$16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *CRYPTO 2016*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

[BKL$^+$07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BKN19] Dusan Bozilov, Miroslav Knezevic, and Ventzislav Nikov. Optimized Threshold Implementations: Minimizing the Latency of Secure Cryptographic Accelerators. In *CARDIS 2019*, volume 11833 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2019.

[BNN$^+$15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small S-boxes. *Cryptogr. Commun.*, 7(1):3–33, 2015.

[CBR$^+$15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES s-box. In *CARDIS 2015*, volume 9514 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2015.

[CBRN14] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, and Svetla Nikova. Higher-Order Glitch Resistant Implementation of the PRESENT S-Box. In *BalkanCryptSec 2014*, volume 9024 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2014.

[CRB$^+$16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 Shares in Hardware. In *CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[Dae16] Joan Daemen. Spectral Characterization of Iterating Lossy Mappings. In *SPACE 2016*, volume 10076 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2016.

[Dae17] Joan Daemen. Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing. In *CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2017.

[DC07] Christophe De Cannière. *Analysis and Design of Symmetric Encrytption Algorithms*. PhD thesis, KULeuven, 2007.

[DMW18] Lauren De Meyer, Amir Moradi, and Felix Wegener. Spin Me Right Round Rotational Symmetry for FPGA-Specific AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):596–626, 2018.

[FGP$^+$18] 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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In *CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic Low-Latency Masking in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation. In *NIST non-invasive attack testing workshop*, 2011. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[GM18] Hannes Groß and Stefan Mangard. A unified masking approach. *J. Cryptogr. Eng.*, 8(2):109–124, 2018.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Theory of Implementation Security - TIS@CCS 2016*, page 3. ACM, 2016.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order. In *CT-RSA 2017*, volume 10159 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2017.

[GSM17a] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-Order Side-Channel Protected Implementations of Keccak. In *DSD 2017*, pages 205–212. IEEE Computer Society, 2017.

[GSM17b] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of keccak. *IACR Cryptol. ePrint Arch.*, 2017:395, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In *ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[MBR19] Lauren De Meyer, Begül Bilgin, and Oscar Reparaz. Consolidating Security Notions in Hardware Masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):119–147, 2019.

[MM13] Amir Moradi and Oliver Mischke. On the Simplicity of Converting Leakages from Multivariate to Univariate - (Case Study of a Glitch-Resistant Masking Scheme). In *CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

[MME10] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-Enhanced Power Analysis Collision Attack. In *CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2010.

[MMSS19] 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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007.

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully Attacking Masked AES Hardware Implementations. In *CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.

[MRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. Multiplicative Masking for AES in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):431–468, 2018.

[MS16a] Amir Moradi and Tobias Schneider. Side-Channel Analysis Protection and Low-Latency in Action - - Case Study of PRINCE and Midori -. In *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 517–547, 2016.

[MS16b] Amir Moradi and François-Xavier Standaert. Moments-Correlating DPA. In *Theory of Implementation Security - TIS@CCS 2016*, pages 5–15. ACM, 2016.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS 2006*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.

[OC15] Colin O'Flynn and Zhizhang Chen. Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection. *J. Cryptogr. Eng.*, 5(1):53–69, 2015.

[PMK+11] Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-Channel Resistant Crypto for Less than 2, 300 GE. *J. Cryptology*, 24(2):322–345, 2011.

[RAD20] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. SCAUL: Power Side-Channel Analysis With Unsupervised Learning. *IEEE Trans. Computers*, 69(11):1626–1638, 2020.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[Rep15] Oscar Reparaz. A note on the security of Higher-Order Threshold Implementations. *IACR Cryptol. ePrint Arch.*, 2015:1, 2015.

[SAK] SAKURA. Side-channel Attack User Reference Architecture. http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

[SM20] Aein Rezaei Shahmirzadi and Amir Moradi. Re-Consolidating First-Order Masking Schemes - Nullifying Fresh Randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):305–342, 2020.

[Tim19] Benjamin Timon. Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):107–131, 2019.

[Tri03] Elena Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. *IACR Cryptol. ePrint Arch.*, 2003:236, 2003.

A 3-share Masked $\mathcal{Q}_4^4$ without Fresh Randomness

$\mathsf{F}(a,b,c,d)$: `0123456789ABDCFE`

$$x = \mathsf{f}(a,b,c,d) = cd + a$$
$$y = \mathsf{g}(a,b,c,d) = b$$
$$z = \mathsf{h}(a,b,c,d) = c$$
$$t = \mathsf{k}(a,b,c,d) = d$$

$$\begin{array}{llll}
\mathsf{f}_0(c_0,d_0) & = c_0d_0 + d_0 & \to x'_0 & \\
\mathsf{f}_1(c_0,d_1) & = c_0d_1 & \to x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
\mathsf{f}_2(c_0,d_2,a_0) & = c_0d_2 + a_0 & \to x'_2 & \\
\hline
\mathsf{f}_3(c_1,d_0) & = c_1d_0 + d_0 & \to x'_3 & \\
\mathsf{f}_4(c_1,d_1) & = c_1d_1 & \to x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
\mathsf{f}_5(c_1,d_2,a_1) & = c_1d_2 + d_2 + a_1 & \to x'_5 & \\
\hline
\mathsf{f}_6(c_2,d_0,a_2) & = c_2d_0 + c_2 + a_2 & \to x'_6 & \\
\mathsf{f}_7(c_2,d_1) & = c_2d_1 & \to x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
\mathsf{f}_8(c_2,d_2) & = c_2d_2 + c_2 + d_2 & \to x'_8 &
\end{array}$$

$$\begin{array}{c}
b_0 \to y_0 \\
b_1 \to y_1 \\
b_2 \to y_2 \\
\\
c_0 \to z_0 \\
c_1 \to z_1 \\
c_2 \to z_2 \\
\\
d_0 \to t_0 \\
d_1 \to t_1 \\
d_2 \to t_2
\end{array}$$

B 3-share Masked $\mathcal{Q}^4_{12}$ without Fresh Randomness

$\mathsf{F}(a,b,c,d)$: 0123456789CDEFAB

$x = \mathsf{f}(a,b,c,d) = a$

$y = \mathsf{g}(a,b,c,d) = bd + cd + b$

$z = \mathsf{h}(a,b,c,d) = bd + c$

$t = \mathsf{k}(a,b,c,d) = d$

$$\begin{array}{l} x_0 = a_0 \\ x_1 = a_1 \\ x_2 = a_2 \end{array}$$

$$\begin{array}{lllr}
\mathsf{g}_0(d_0,c_0,b_0) & = d_0c_0 + d_0b_0 & \rightarrow y'_0 & \\
\mathsf{g}_1(d_0,c_1,b_1) & = d_0c_1 + d_0b_1 + b_1 & \rightarrow y'_1 & y'_0 + y'_1 + y'_2 = y_0 \\
\mathsf{g}_2(d_0,c_2,b_2) & = d_0c_2 + d_0b_2 + c_2 + b_2 & \rightarrow y'_2 & \\
\hline
\mathsf{g}_3(d_1,c_0,b_0) & = d_1c_0 + d_1b_0 + c_0 & \rightarrow y'_3 & \\
\mathsf{g}_4(d_1,c_1,b_1) & = d_1c_1 + d_1b_1 & \rightarrow y'_4 & y'_3 + y'_4 + y'_5 = y_1 \\
\mathsf{g}_5(d_1,c_2,b_2) & = d_1c_2 + d_1b_2 + c_2 + b_2 & \rightarrow y'_5 & \\
\hline
\mathsf{g}_6(d_2,c_0,b_0) & = d_2c_0 + d_2b_0 + d_2 + c_0 + b_0 & \rightarrow x'_6 & \\
\mathsf{g}_7(d_2,c_1,b_1) & = d_2c_1 + d_2b_1 & \rightarrow y'_7 & y'_6 + y'_7 + y'_8 = y_2 \\
\mathsf{g}_8(d_2,c_2,b_2) & = d_2c_2 + d_2b_2 + d_2 + b_2 & \rightarrow y'_8 &
\end{array}$$

$$\begin{array}{lllr}
\mathsf{h}_0(d_0,b_0,c_0) & = d_0b_0 + b_0 + c_0 & \rightarrow z'_0 & \\
\mathsf{h}_1(d_0,b_1) & = d_0b_1 + d_0 + b_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 = z_0 \\
\mathsf{h}_2(d_0,b_2) & = d_0b_2 + d_0 & \rightarrow z'_2 & \\
\hline
\mathsf{h}_3(d_1,b_0) & = d_1b_0 + d_1 + b_0 & \rightarrow z'_3 & \\
\mathsf{h}_4(d_1,b_1) & = d_1b_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 = z_1 \\
\mathsf{h}_5(d_1,b_2,c_2) & = d_1b_2 + d_1 + b_2 + c_2 & \rightarrow z'_5 & \\
\hline
\mathsf{h}_6(d_2,b_0) & = d_2b_0 + d_2 & \rightarrow z'_6 & \\
\mathsf{h}_7(d_2,b_1,c_1) & = d_2b_1 + b_1 + c_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 = z_2 \\
\mathsf{h}_8(d_2,b_2) & = d_2b_2 + d_2 + b_2 & \rightarrow z'_8 &
\end{array}$$

$$\begin{array}{l} d_0 \rightarrow t_0 \\ d_1 \rightarrow t_1 \\ d_2 \rightarrow t_2 \end{array}$$

C 3-share Masked $\mathcal{Q}^4_{293}$ without Fresh Randomness

$\mathsf{F}(a,b,c,d)$: `0123457689CDEFBA`

$x = \mathsf{f}(a,b,c,d) = bc + a$

$y = \mathsf{g}(a,b,c,d) = bd + cd + b$

$z = \mathsf{h}(a,b,c,d) = bd + c$

$t = \mathsf{k}(a,b,c,d) = d$

$$\begin{array}{llll}
\mathsf{f}_0(b_0,c_0,a_0) & = b_0c_0 + a_0 + c_0 & \rightarrow x'_0 & \\
\mathsf{f}_1(b_0,c_1) & = b_0c_1 + b_0 + c_1 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
\mathsf{f}_2(b_0,c_2) & = b_0c_2 + b_0 & \rightarrow x'_2 & \\
\hline
\mathsf{f}_3(b_1,c_0) & = b_1c_0 + c_0 & \rightarrow x'_3 & \\
\mathsf{f}_4(b_1,c_1) & = b_1c_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
\mathsf{f}_5(b_1,c_2,a_1) & = b_1c_2 + a_1 & \rightarrow x'_5 & \\
\hline
\mathsf{f}_6(b_2,c_0) & = b_2c_0 + b_2 & \rightarrow x'_6 & \\
\mathsf{f}_7(b_2,c_1) & = b_2c_1 + c_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
\mathsf{f}_8(b_2,c_2,a_2) & = b_2c_2 + a_2 + b_2 & \rightarrow x'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{g}_0(b_0,c_0,d_0,a_0) & = b_0d_0 + c_0d_0 + a_0 + b_0 + d_0 & \rightarrow y'_0 & \\
\mathsf{g}_1(b_1,c_2,d_0) & = b_1d_0 + c_2d_0 & \rightarrow y'_1 & y'_0 + y'_1 + y'_2 \ = y_0 \\
\mathsf{g}_2(b_2,c_1,d_0) & = b_2d_0 + c_1d_0 + b_2 + c_1 + d_0 & \rightarrow y'_2 & \\
\hline
\mathsf{g}_3(b_0,c_1,d_1,a_0) & = b_0d_1 + c_1d_1 + a_0 + c_1 & \rightarrow y'_3 & \\
\mathsf{g}_4(b_1,c_2,d_1,a_1) & = b_1d_1 + c_2d_1 + a_1 + b_1 & \rightarrow y'_4 & y'_3 + y'_4 + y'_5 \ = y_1 \\
\mathsf{g}_5(b_2,c_0,d_1) & = b_2d_1 + c_0d_1 + b_2 + c_0 & \rightarrow y'_5 & \\
\hline
\mathsf{g}_6(b_0,c_2,d_2) & = b_0d_2 + c_2d_2 & \rightarrow y'_6 & \\
\mathsf{g}_7(b_1,c_1,d_2,a_1) & = b_1d_2 + c_1d_2 + a_1 & \rightarrow y'_7 & y'_6 + y'_7 + y'_8 \ = y_2 \\
\mathsf{g}_8(b_2,c_0,d_2) & = b_2d_2 + c_0d_2 + b_2 + c_0 & \rightarrow y'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{h}_0(b_0,d_0,a_0) & = b_0d_0 + a_0 + d_0 & \rightarrow z'_0 & \\
\mathsf{h}_1(b_1,d_0) & = b_1d_0 + b_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 \ = z_0 \\
\mathsf{h}_2(b_2,d_0,c_0) & = b_2d_0 + c_0 + d_0 & \rightarrow z'_2 & \\
\hline
\mathsf{h}_3(b_0,d_1,c_1,a_0) & = b_0d_1 + a_0 + c_1 & \rightarrow z'_3 & \\
\mathsf{h}_4(b_1,d_1,a_1) & = b_1d_1 + a_1 + b_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 \ = z_1 \\
\mathsf{h}_5(b_2,d_1,b_1) & = b_2d_1 + b_2 & \rightarrow z'_5 & \\
\hline
\mathsf{h}_6(b_0,d_2,c_2) & = b_0d_2 + c_2 + d_2 & \rightarrow z'_6 & \\
\mathsf{h}_7(b_1,d_2,a_1) & = b_1d_2 + a_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 \ = z_2 \\
\mathsf{h}_8(b_2,d_2) & = b_2d_2 + b_2 + d_2 & \rightarrow z'_8 &
\end{array}$$

$$\begin{array}{l}
d_0 \rightarrow t_0 \\
d_1 \rightarrow t_1 \\
d_2 \rightarrow t_2
\end{array}$$

D 3-share Masked $\mathcal{Q}^4_{294}$ without Fresh Randomness

$F(a,b,c,d)$: `0123456789BAEFDC`

$$
\begin{aligned}
x &= f(a,b,c,d) = bd + a\\
y &= g(a,b,c,d) = cd + b\\
z &= h(a,b,c,d) = c\\
t &= k(a,b,c,d) = d
\end{aligned}
$$

$$
\begin{array}{llll}
f_0(b_0,d_0,a_0) & = b_0d_0 + b_0 + d_0 + a_0 & \rightarrow x'_0 & \\
f_1(b_0,d_1) & = b_0d_1 + d_1 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0\\
f_2(b_0,d_2) & = b_0d_2 + b_0 & \rightarrow x'_2 & \\
\hline
f_3(b_1,d_0) & = b_1d_0 + d_0 & \rightarrow x'_3 & \\
f_4(b_1,d_1) & = b_1d_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1\\
f_5(b_1,d_2,a_1) & = b_1d_2 + a_1 & \rightarrow x'_5 & \\
\hline
f_6(b_2,d_0,a_2) & = b_2d_0 + a_2 & \rightarrow x'_6 & \\
f_7(b_2,d_1) & = b_2d_1 + d_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2\\
f_8(b_2,d_2) & = b_2d_2 & \rightarrow x'_8 &
\end{array}
$$

$$
\begin{array}{llll}
g_0(d_0,c_0,b_2) & = d_0c_0 + b_2 & \rightarrow y'_0 & \\
g_1(d_0,c_1) & = d_0c_1 + d_0 + c_1 & \rightarrow y'_1 & y'_0 + y'_1 + y'_2 = y_0\\
g_2(d_0,c_2) & = d_0c_2 + d_0 & \rightarrow y'_2 & \\
\hline
g_3(d_1,c_0) & = d_1c_0 + d_1 + c_0 & \rightarrow y'_3 & \\
g_4(d_1,c_1) & = d_1c_1 + d_1 & \rightarrow y'_4 & y'_3 + y'_4 + y'_5 = y_1\\
g_5(d_1,c_2,b_1) & = d_1c_2 + b_1 & \rightarrow y'_5 & \\
\hline
g_6(d_2,c_0) & = d_2c_0 + d_2 + c_0 & \rightarrow y'_6 & \\
g_7(d_2,c_1,b_0) & = d_2c_1 + d_2 + c_1 + b_0 & \rightarrow y'_7 & y'_6 + y'_7 + y'_8 = y_2\\
g_8(d_2,c_2) & = d_2c_2 & \rightarrow y'_8 &
\end{array}
$$

$$
\begin{aligned}
c_0 &\rightarrow z_0\\
c_1 &\rightarrow z_1\\
c_2 &\rightarrow z_2
\end{aligned}
$$

$$
\begin{aligned}
d_0 &\rightarrow t_0\\
d_1 &\rightarrow t_1\\
d_2 &\rightarrow t_2
\end{aligned}
$$

E 3-share Masked $\mathcal{Q}^4_{299}$ without Fresh Randomness

$F(a,b,c,d)$: `012345678ACEB9FD`

$$x = f(a,b,c,d) = ad + cd + a$$

$$y = g(a,b,c,d) = ad + bd + cd + b$$

$$z = h(a,b,c,d) = bd + cd + c$$

$$t = k(a,b,c,d) = d$$

$$\begin{array}{lllll}
f_0(a_0,d_0,c_0) & = a_0d_0 + c_0d_0 + a_0 + c_0 & \to x'_0 & & \\
f_1(a_1,d_0,c_1) & = a_1d_0 + c_1d_0 + a_1 & \to x'_1 & x'_0 + x'_1 + x'_2 & = x_0 \\
f_2(a_2,d_0,c_2) & = a_2d_0 + c_2d_0 & \to x'_2 & & \\
\hline
f_3(a_0,d_1,c_0) & = a_0d_1 + c_0d_1 + a_0 & \to x'_3 & & \\
f_4(a_1,d_1,c_1) & = a_1d_1 + c_1d_1 & \to x'_4 & x'_3 + x'_4 + x'_5 & = x_1 \\
f_5(a_2,d_1,c_2) & = a_2d_1 + c_2d_1 + a_2 + c_2 & \to x'_5 & & \\
\hline
f_6(a_0,d_2,c_0) & = a_0d_2 + c_0d_2 + a_0 + c_0 & \to x'_6 & & \\
f_7(a_1,d_2,c_1) & = a_1d_2 + c_1d_2 & \to x'_7 & x'_6 + x'_7 + x'_8 & = x_2 \\
f_8(a_2,d_2,c_2) & = a_2d_2 + c_2d_2 + c_2 & \to x'_8 & &
\end{array}$$

$$\begin{array}{lllll}
g_0(a_0,d_0,b_0,c_0) & = a_0d_0 + b_0d_0 + c_0d_0 + b_0 + d_0 & \to y'_0 & & \\
g_1(a_1,d_0,b_1,c_1) & = a_1d_0 + b_1d_0 + c_1d_0 + d_0 & \to y'_1 & y'_0 + y'_1 + y'_2 & = y_0 \\
g_2(a_2,d_0,b_2,c_2) & = a_2d_0 + b_2d_0 + c_2d_0 + a_2 & \to y'_2 & & \\
\hline
g_3(a_0,d_1,b_0,c_0) & = a_0d_1 + b_0d_1 + c_0d_1 & \to y'_3 & & \\
g_4(a_1,d_1,b_1,c_1) & = a_1d_1 + b_1d_1 + c_1d_1 + & & & \\
 & \quad a_1 + c_1 + d_1 & \to y'_4 & y'_3 + y'_4 + y'_5 & = y_1 \\
g_5(a_2,d_1,b_2,c_2) & = a_2d_1 + b_2d_1 + c_2d_1 + & & & \\
 & \quad a_2 + b_2 + c_2 + d_1 & \to y'_5 & & \\
\hline
g_6(a_0,d_2,b_0,c_0) & = a_0d_2 + b_0d_2 + c_0d_2 & \to y'_6 & & \\
g_7(a_1,d_2,b_1,c_1) & = a_1d_2 + b_1d_2 + c_1d_2 + a_1 + b_1 + c_1 & \to y'_7 & y'_6 + y'_7 + y'_8 & = y_2 \\
g_8(a_2,d_2,b_2,c_2) & = a_2d_2 + b_2d_2 + c_2d_2 + c_2 & \to y'_8 & &
\end{array}$$

$$\begin{array}{lllll}
h_0(b_0,d_0,c_0) & = b_0d_0 + c_0d_0 & \to z'_0 & & \\
h_1(b_1,d_0,c_1) & = b_1d_0 + c_1d_0 + c_1 & \to z'_1 & z'_0 + z'_1 + z'_2 & = z_0 \\
h_2(b_2,d_0,c_2) & = b_2d_0 + c_2d_0 + b_2 + c_2 & \to z'_2 & & \\
\hline
h_3(b_0,d_1,c_0) & = b_0d_1 + c_0d_1 + b_0 & \to z'_3 & & \\
h_4(b_1,d_1,c_1) & = b_1d_1 + c_1d_1 & \to z'_4 & z'_3 + z'_4 + z'_5 & = z_1 \\
h_5(b_2,d_1,c_2) & = b_2d_1 + c_2d_1 + b_2 + c_2 & \to z'_5 & & \\
\hline
h_6(b_0,d_2,c_0) & = b_0d_2 + c_0d_2 + b_0 + c_0 & \to z'_6 & & \\
h_7(b_1,d_2,c_1) & = b_1d_2 + c_1d_2 & \to z'_7 & z'_6 + z'_7 + z'_8 & = z_2 \\
h_8(b_2,d_2,c_2) & = b_2d_2 + c_2d_2 + c_2 & \to z'_8 & &
\end{array}$$

$$\begin{array}{l}
d_0 \to t_0 \\
d_1 \to t_1 \\
d_2 \to t_2
\end{array}$$

F 3-share Masked $\mathcal{Q}^4_{300}$ with 6-bit Fresh Randomness

$Q^4_{300}(a,b,c,d)$: `0123458967CDEFAB` $= G \circ F$
$F(a,b,c,d)$: `02468A13DF9BCE57`

$$x = f(a,b,c,d) = bc + cd + d \qquad y = g(a,b,c,d) = a$$
$$z = h(a,b,c,d) = bcd + b + d \qquad t = k(a,b,c,d) = bc + cd + c + d$$

$G(x,y,z,t)$: `08192A3B4C5DE6F7` is $\mathcal{Q}^4_4$ with permuted outputs given in Appendix A

$$\begin{array}{lll|ll}
f_0(c_0,b_0,d_0) & = c_0b_0 + c_0d_0 + c_0 + b_0 + d_0 & \rightarrow x'_0 & & \\
f_1(c_0,b_1,d_1) & = c_0b_1 + c_0d_1 + c_0 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 + r_0 & \rightarrow x_0 \\
f_2(c_0,b_2,d_2) & = c_0b_2 + c_0d_2 + b_2 & \rightarrow x'_2 & & \\
\hline
f_3(c_1,b_0,d_0) & = c_1b_0 + c_1d_0 + c_1 & \rightarrow x'_3 & & \\
f_4(c_1,b_1,d_1) & = c_1b_1 + c_1d_1 + b_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 + r_1 & \rightarrow x_1 \\
f_5(c_1,b_2,d_2) & = c_1b_2 + c_1d_2 + c_1 + b_2 + d_2 & \rightarrow x'_5 & & \\
\hline
f_6(c_2,b_0,d_0) & = c_2b_0 + c_2d_0 + b_0 & \rightarrow x'_6 & & \\
f_7(c_2,b_1,d_1) & = c_2b_1 + c_2d_1 + b_1 + d_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 + r_0 + r_1 & \rightarrow x_2 \\
f_8(c_2,b_2,d_2) & = c_2b_2 + c_2d_2 & \rightarrow x'_8 & &
\end{array}$$

$$\begin{array}{rl}
a_0 \rightarrow y'_0 & y'_0 \rightarrow y_0 \\
a_1 \rightarrow y'_1 & y'_1 \rightarrow y_1 \\
a_2 \rightarrow y'_2 & y'_2 \rightarrow y_2
\end{array}$$

$$\begin{array}{lll|ll}
h_0(c_0,b_0) & = c_0b_0 + c_0 & \rightarrow z'_0 & & \\
h_1(c_0,b_1,d_1) & = c_0b_1 + b_1 + d_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 + r_2 & \rightarrow z_0 \\
h_2(c_0,b_2) & = c_0b_2 + c_0 + b_2 & \rightarrow z'_2 & & \\
\hline
h_3(c_1,b_0,d_0) & = c_1b_0 + d_0 & \rightarrow z'_3 & & \\
h_4(c_1,b_1) & = c_1b_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 + r_3 & \rightarrow z_1 \\
h_5(c_1,b_2) & = c_1b_2 + b_2 & \rightarrow z'_5 & & \\
\hline
h_6(c_2,b_0) & = c_2b_0 + b_0 & \rightarrow z'_6 & & \\
h_7(c_2,b_1) & = c_2b_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 + r_2 + r_3 & \rightarrow z_2 \\
h_8(c_2,b_2,d_2) & = c_2b_2 + b_2 + d_2 & \rightarrow z'_8 & &
\end{array}$$

$$\begin{array}{lll|ll}
k_0(c_0,b_0,d_0) & = c_0b_0 + c_0d_0 + c_0 + b_0 + d_0 & \rightarrow t'_0 & & \\
k_1(c_0,b_1,d_1) & = c_0b_1 + c_0d_1 + c_0 & \rightarrow t'_1 & t'_0 + t'_1 + t'_2 + r_4 & \rightarrow t_0 \\
k_2(c_0,b_2,d_2) & = c_0b_2 + c_0d_2 + c_0 + b_2 & \rightarrow t'_2 & & \\
\hline
k_3(c_1,b_0,d_0) & = c_1b_0 + c_1d_0 & \rightarrow t'_3 & & \\
k_4(c_1,b_1,d_1) & = c_1b_1 + c_1d_1 + b_1 & \rightarrow t'_4 & t'_3 + t'_4 + t'_5 + r_5 & \rightarrow t_1 \\
k_5(c_1,b_2,d_2) & = c_1b_2 + c_1d_2 + c_1 + b_2 + d_2 & \rightarrow t'_5 & & \\
\hline
k_6(c_2,b_0,d_0) & = c_2b_0 + c_2d_0 + c_2 + b_0 & \rightarrow t'_6 & & \\
k_7(c_2,b_1,d_1) & = c_2b_1 + c_2d_1 + c_2 + b_1 + d_1 & \rightarrow t'_7 & t'_6 + t'_7 + t'_8 + r_4 + r_5 & \rightarrow t_2 \\
k_8(c_2,b_2,d_2) & = c_2b_2 + c_2d_2 + c_2 & \rightarrow t'_8 & &
\end{array}$$

G 2-share Masked χ function without Fresh Randomness

$$\begin{aligned}
x &= f(a,e,d) = de + a + d\\
y &= g(a,b,e) = ae + b + e\\
z &= h(a,b,c) = ab + a + c\\
t &= k(b,c,d) = bc + b + d\\
w &= m(c,d,e) = cd + c + e
\end{aligned}$$

$$\begin{array}{llll}
f_0(d_0,e_0,a_0) & = d_0e_0 + a_0 + d_0 & \rightarrow x'_0 & \\
f_1(d_0,e_1) & = d_0e_1 & \rightarrow x'_1 & x'_0 + x'_1 \; = x_0 \\
\hline
f_2(d_1,e_0,a_1) & = d_1e_0 + a_1 + d_1 & \rightarrow x'_2 & x'_2 + x'_3 \; = x_1 \\
f_3(d_1,e_1) & = d_1e_1 & \rightarrow x'_3 & \\
\\
g_0(a_0,e_0) & = a_0e_0 & \rightarrow y'_0 & \\
g_1(a_0,e_1,b_0) & = a_0e_1 + b_0 & \rightarrow y'_1 & y'_0 + y'_1 \; = y_0 \\
\hline
g_2(a_1,e_0) & = a_1e_0 + e_0 & \rightarrow y'_2 & y'_2 + y'_3 \; = y_1 \\
g_3(a_1,e_1,b_1) & = a_1e_1 + b_1 + e_1 & \rightarrow y'_3 & \\
\\
h_0(a_0,b_0) & = a_0b_0 & \rightarrow z'_0 & \\
h_1(a_0,b_1,c_0) & = a_0b_1 + a_0 + c_0 & \rightarrow z'_1 & z'_0 + z'_1 \; = z_0 \\
\hline
h_2(a_1,b_0,c_1) & = a_1b_0 + a_1 + c_1 & \rightarrow z'_2 & z'_2 + z'_3 \; = z_1 \\
h_3(a_1,b_1) & = a_1b_1 & \rightarrow z'_3 & \\
\\
k_0(b_0,c_0,d_0,a_0) & = b_0c_0 + a_0 + d_0 & \rightarrow t'_0 & \\
k_1(b_0,c_1,a_0) & = b_0c_1 + a_0 + b_0 & \rightarrow t'_1 & t'_0 + t'_1 \; = t_0 \\
\hline
k_2(b_1,c_0) & = b_1c_0 & \rightarrow t'_2 & t'_2 + t'_3 \; = t_1 \\
k_3(b_1,c_1,d_1) & = b_1c_1 + b_1 + d_1 & \rightarrow t'_3 & \\
\\
m_0(c_0,d_0,e_0) & = c_0d_0 + e_0 & \rightarrow w'_0 & \\
m_1(c_0,d_1) & = c_0d_1 + c_0 & \rightarrow w'_1 & w'_0 + w'_1 = w_0 \\
\hline
m_2(c_1,d_0,a_1,b_1,e_1) & = c_1d_0 + a_1 + b_1 + e_1 & \rightarrow w'_2 & w'_2 + w'_3 = w_1 \\
m_3(c_1,d_1,a_1,b_1) & = c_1d_1 + a_1 + b_1 + c_1 & \rightarrow w'_3 &
\end{array}$$

H 3-share Masked χ function without Fresh Randomness

$$x = \mathsf{f}(a,e,d) = de + a + d \qquad y = \mathsf{g}(a,b,e) = ae + b + e \qquad z = \mathsf{h}(a,b,c) = ab + a + c$$
$$t = \mathsf{k}(b,c,d) = bc + b + d \qquad w = \mathsf{m}(c,d,e) = cd + c + e$$

$$\begin{array}{llll}
\mathsf{f}_0(d_0,e_0,a_0) & = d_0 + a_0 + d_0e_0 & \to x'_0 & \\
\mathsf{f}_1(d_1,e_0) & = d_1 + d_1e_0 & \to x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
\mathsf{f}_2(d_2,e_0) & = d_2e_0 & \to x'_2 & \\ \hline
\mathsf{f}_3(d_0,e_1) & = d_0 + d_0e_1 & \to x'_3 & \\
\mathsf{f}_4(d_1,e_1) & = d_1e_1 & \to x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
\mathsf{f}_5(d_2,e_1,a_2) & = a_2 + d_2e_1 & \to x'_5 & \\ \hline
\mathsf{f}_6(d_0,e_2) & = d_0 + d_0e_2 & \to x'_6 & \\
\mathsf{f}_7(d_1,e_2) & = d_1e_2 & \to x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
\mathsf{f}_8(d_2,e_2,a_1) & = d_2 + a_1 + d_2e_2 & \to x'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{g}_0(a_0,e_0,b_0) & = e_0 + a_0 + b_0 + e_0a_0 & \to y'_0 & \\
\mathsf{g}_1(a_0,e_1) & = e_1 + e_1a_0 & \to y'_1 & y'_0 + y'_1 + y'_2 = y_0 \\
\mathsf{g}_2(a_0,e_2) & = a_0 + e_2a_0 & \to y'_2 & \\ \hline
\mathsf{g}_3(a_1,e_0) & = e_0 + e_0a_1 & \to y'_3 & \\
\mathsf{g}_4(a_1,e_1) & = e_1a_1 & \to y'_4 & y'_3 + y'_4 + y'_5 = y_1 \\
\mathsf{g}_5(a_1,e_2,b_2) & = e_2 + b_2 + e_2a_1 & \to y'_5 & \\ \hline
\mathsf{g}_6(a_2,e_0) & = e_0 + e_0a_2 & \to y'_6 & \\
\mathsf{g}_7(a_2,e_1) & = e_1a_2 & \to y'_7 & y'_6 + y'_7 + y'_8 = y_2 \\
\mathsf{g}_8(a_2,e_2,b_1) & = b_1 + e_2a_2 & \to y'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{h}_0(a_0,b_0,c_2) & = a_0 + b_0 + c_2 + a_0b_0 & \to z'_0 & \\
\mathsf{h}_1(a_1,b_0) & = b_0 + a_1b_0 & \to z'_1 & z'_0 + z'_1 + z'_2 = z_0 \\
\mathsf{h}_2(a_2,b_0) & = a_2 + a_2b_0 & \to z'_2 & \\ \hline
\mathsf{h}_3(a_0,b_1) & = a_0 + a_0b_1 & \to z'_3 & \\
\mathsf{h}_4(a_1,b_1) & = a_1b_1 & \to z'_4 & z'_3 + z'_4 + z'_5 = z_1 \\
\mathsf{h}_5(a_2,b_1,c_0) & = c_0 + a_2b_1 & \to z'_5 & \\ \hline
\mathsf{h}_6(a_0,b_2,c_1) & = a_0 + c_1 + a_0b_2 & \to z'_6 & \\
\mathsf{h}_7(a_1,b_2) & = a_1 + a_1b_2 & \to z'_7 & z'_6 + z'_7 + z'_8 = z_2 \\
\mathsf{h}_8(a_2,b_2) & = a_2b_2 & \to z'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{k}_0(b_0,c_0) & = b_0 + c_0 + b_0c_0 & \to t'_0 & \\
\mathsf{k}_1(b_1,c_0) & = b_1c_0 & \to t'_1 & t'_0 + t'_1 + t'_2 = t_0 \\
\mathsf{k}_2(b_2,c_0,d_2) & = c_0 + d_2 + b_2c_0 & \to t'_2 & \\ \hline
\mathsf{k}_3(b_0,c_1) & = b_0 + b_0c_1 & \to t'_3 & \\
\mathsf{k}_4(b_1,c_1) & = b_1c_1 & \to t'_4 & t'_3 + t'_4 + t'_5 = t_1 \\
\mathsf{k}_5(b_2,c_1,d_0) & = b_2 + d_0 + b_2c_1 & \to t'_5 & \\ \hline
\mathsf{k}_6(b_0,c_2,d_1) & = b_0 + c_2 + d_1 + b_0c_2 & \to t'_6 & \\
\mathsf{k}_7(b_1,c_2) & = b_1 + b_1c_2 & \to t'_7 & t'_6 + t'_7 + t'_8 = t_2 \\
\mathsf{k}_8(b_2,c_2) & = c_2 + b_2c_2 & \to t'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{m}_0(c_0,d_0) & = c_0d_0 & \to w'_0 & \\
\mathsf{m}_1(c_1,d_0) & = c_1 + d_0 + c_1d_0 & \to w'_1 & w'_0 + w'_1 + w'_2 = w_0 \\
\mathsf{m}_2(c_2,d_0,e_0) & = c_2 + d_0 + e_0 + c_2d_0 & \to w'_2 & \\ \hline
\mathsf{m}_3(c_0,d_1) & = c_0 + c_0d_1 & \to w'_3 & \\
\mathsf{m}_4(c_1,d_1) & = c_1d_1 & \to w'_4 & w'_3 + w'_4 + w'_5 = w_1 \\
\mathsf{m}_5(c_2,d_1,e_1) & = c_2 + e_1 + c_2d_1 & \to w'_5 & \\ \hline
\mathsf{m}_6(c_0,d_2,e_2) & = e_2 + c_0d_2 & \to w'_6 & \\
\mathsf{m}_7(c_1,d_2) & = c_1d_2 & \to w'_7 & w'_6 + w'_7 + w'_8 = w_2 \\
\mathsf{m}_8(c_2,d_2) & = c_2 + c_2d_2 & \to w'_8 &
\end{array}$$

I 3-share Masked Midori S-box with 8-bit Fresh Masks

S : CAD3EBF789150246 $= G \circ F \circ A_1$

$A_1(a,b,c,d)$: 93821B0AF5E47D6C : $x = b+1, \quad y = a+d, \quad z = d, \quad t = a+c+1$

$F(a,b,c,d)$: 08C43BF7192AE6D5

$x = f(a,b,c,d) = bd+c+d \qquad y = g(a,b,c,d) = bd+c$

$z = h(a,b,c,d) = bd+cd+b \qquad t = k(a,b,c,d) = bd+cd+a+b$

$$
\begin{array}{llllll}
f_0(d_0,b_0) & = d_0b_0 & \to x'_0 & & \\
f_1(d_0,b_1,c_1) & = d_0b_1+c_1 & \to x'_1 & x'_0+x'_1+x'_2+r_0 & \to x_0 \\
f_2(d_0,b_2) & = d_0b_2+d_0+b_2 & \to x'_2 & & \\
\hline
f_3(d_1,b_0,c_0) & = d_1b_0+c_0 & \to x'_3 & & \\
f_4(d_1,b_1) & = d_1b_1+b_1 & \to x'_4 & x'_3+x'_4+x'_5+r_1 & \to x_1 \\
f_5(d_1,b_2) & = d_1b_2+d_1 & \to x'_5 & & \\
\hline
f_6(d_2,b_0) & = d_2b_0+d_2 & \to x'_6 & & \\
f_7(d_2,b_1) & = d_2b_1+b_1 & \to x'_7 & x'_6+x'_7+x'_8+r_0+r_1 & \to x_2 \\
f_8(d_2,b_2,c_2) & = d_2b_2+b_2+c_2 & \to x'_8 & &
\end{array}
$$

$$
\begin{array}{llllll}
g_0(d_0,b_0) & = d_0b_0+d_0 & \to y'_0 & & \\
g_1(d_0,b_1,c_1) & = d_0b_1+d_0+c_1 & \to y'_1 & y'_0+y'_1+y'_2+r_2 & \to y_0 \\
g_2(d_0,b_2) & = d_0b_2+b_2 & \to y'_2 & & \\
\hline
g_3(d_1,b_0,c_0) & = d_1b_0+d_1+c_0 & \to y'_3 & & \\
g_4(d_1,b_1) & = d_1b_1+b_1 & \to y'_4 & y'_3+y'_4+y'_5+r_3 & \to y_1 \\
g_5(d_1,b_2) & = d_1b_2+d_1 & \to y'_5 & & \\
\hline
g_6(d_2,b_0) & = d_2b_0+d_2 & \to y'_6 & & \\
g_7(d_2,b_1) & = d_2b_1+b_1 & \to y'_7 & y'_6+y'_7+y'_8+r_2+r_3 & \to y_2 \\
g_8(d_2,b_2,c_2) & = d_2b_2+d_2+b_2+c_2 & \to y'_8 & &
\end{array}
$$

$$
\begin{array}{llllll}
h_0(d_0,c_0,b_0) & = d_0c_0+d_0b_0+c_0+b_0 & \to z'_0 & & \\
h_1(d_0,c_1,b_1) & = d_0c_1+d_0b_1+d_0 & \to z'_1 & z'_0+z'_1+z'_2+r_4 & \to z_0 \\
h_2(d_0,c_2,b_2) & = d_0c_2+d_0b_2+d_0+c_2 & \to z'_2 & & \\
\hline
h_3(d_1,c_0,b_0) & = d_1c_0+d_1b_0+b_0 & \to z'_3 & & \\
h_4(d_1,c_1,b_1) & = d_1c_1+d_1b_1 & \to z'_4 & z'_3+z'_4+z'_5+r_5 & \to z_1 \\
h_5(d_1,c_2,b_2) & = d_1c_2+d_1b_2+c_2+b_2 & \to z'_5 & & \\
\hline
h_6(d_2,c_0,b_0) & = d_2c_0+d_2b_0+c_0+b_0 & \to z'_6 & & \\
h_7(d_2,c_1,b_1) & = d_2c_1+d_2b_1+b_1 & \to z'_7 & z'_6+z'_7+z'_8+r_4+r_5 & \to z_2 \\
h_8(d_2,c_2,b_2) & = d_2c_2+d_2b_2 & \to z'_8 & &
\end{array}
$$

$$
\begin{array}{llllll}
k_0(d_0,c_0,b_0) & = d_0c_0+d_0b_0+c_0+b_0 & \to t'_0 & & \\
k_1(d_0,c_1,b_1,a_0) & = d_0c_1+d_0b_1+d_0+a_0 & \to t'_1 & t'_0+t'_1+t'_2+r_6 & \to t_0 \\
k_2(d_0,c_2,b_2) & = d_0c_2+d_0b_2+d_0+c_2 & \to t'_2 & & \\
\hline
k_3(d_1,c_0,b_0) & = d_1c_0+d_1b_0+b_0 & \to t'_3 & & \\
k_4(d_1,c_1,b_1,a_1) & = d_1c_1+d_1b_1+a_1 & \to t'_4 & t'_3+t'_4+t'_5+r_7 & \to t_1 \\
k_5(d_1,c_2,b_2) & = d_1c_2+d_1b_2+c_2+b_2 & \to t'_5 & & \\
\hline
k_6(d_2,c_0,b_0) & = d_2c_0+d_2b_0+d_2+c_0+b_0 & \to t'_6 & & \\
k_7(d_2,c_1,b_1,a_2) & = d_2c_1+d_2b_1+b_1+a_2 & \to t'_7 & t'_6+t'_7+t'_8+r_6+r_7 & \to t_2 \\
k_8(d_2,c_2,b_2) & = d_2c_2+d_2b_2+d_2 & \to t'_8 & &
\end{array}
$$

$G(a,b,c,d) : \texttt{FD75A820ECB93164}$

$x = f(a,b,c,d) = bd + c + d + 1$

$y = g(a,b,c,d) = a + 1$

$z = h(a,b,c,d) = bd + c + 1$

$t = k(a,b,c,d) = bd + cd + b + 1$

$$\begin{array}{llll}
f_0(d_0,b_0) & = d_0b_0 + d_0 + 1 & \rightarrow x'_0 & \\
f_1(d_0,b_1,c_1) & = d_0b_1 + d_0 + b_1 + c_1 & \rightarrow x'_1 & x'_0 + x'_1 + x'_2 = x_0 \\
f_2(d_0,b_2) & = d_0b_2 + d_0 + b_2 & \rightarrow x'_2 & \\
\hline
f_3(d_1,b_0,c_0) & = d_1b_0 + c_0 & \rightarrow x'_3 & \\
f_4(d_1,b_1) & = d_1b_1 & \rightarrow x'_4 & x'_3 + x'_4 + x'_5 = x_1 \\
f_5(d_1,b_2) & = d_1b_2 + d_1 + b_2 & \rightarrow x'_5 & \\
\hline
f_6(d_2,b_0) & = d_2b_0 + d_2 & \rightarrow x'_6 & \\
f_7(d_2,b_1) & = d_2b_1 + b_1 & \rightarrow x'_7 & x'_6 + x'_7 + x'_8 = x_2 \\
f_8(d_2,b_2,c_2) & = d_2b_2 + c_2 & \rightarrow x'_8 &
\end{array}$$

$$\begin{array}{rl}
1 + a_0 & \rightarrow y_0 \\
a_1 & \rightarrow y_1 \\
a_2 & \rightarrow y_2
\end{array}$$

$$\begin{array}{llll}
h_0(d_0,b_0) & = d_0b_0 + 1 & \rightarrow z'_0 & \\
h_1(d_0,b_1,c_1) & = d_0b_1 + d_0 + b_1 + c_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 \;\; = z_0 \\
h_2(d_0,b_2) & = d_0b_2 + d_0 + b_2 & \rightarrow z'_2 & \\
\hline
h_3(d_1,b_0,c_0) & = d_1b_0 + c_0 & \rightarrow z'_3 & \\
h_4(d_1,b_1) & = d_1b_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 \;\; = z_1 \\
h_5(d_1,b_2) & = d_1b_2 + b_2 & \rightarrow z'_5 & \\
\hline
h_6(d_2,b_0) & = d_2b_0 & \rightarrow z'_6 & \\
h_7(d_2,b_1) & = d_2b_1 + b_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 \;\; = z_2 \\
h_8(d_2,b_2,c_2) & = d_2b_2 + c_2 & \rightarrow z'_8 &
\end{array}$$

$$\begin{array}{llll}
k_0(d_0,c_0,b_0) & = d_0c_0 + d_0b_0 + d_0 + c_0 + 1 & \rightarrow t'_0 & \\
k_1(d_0,c_1,b_1) & = d_0c_1 + d_0b_1 + d_0 & \rightarrow t'_1 & t'_0 + t'_1 + t'_2 \;\; = t_0 \\
k_2(d_0,c_2,b_2) & = d_0c_2 + d_0b_2 + c_2 + b_2 & \rightarrow t'_2 & \\
\hline
k_3(d_1,c_0,b_0) & = d_1c_0 + d_1b_0 + d_1 & \rightarrow t'_3 & \\
k_4(d_1,c_1,b_1) & = d_1c_1 + d_1b_1 + d_1 + b_1 & \rightarrow t'_4 & t'_3 + t'_4 + t'_5 \;\; = t_1 \\
k_5(d_1,c_2,b_2) & = d_1c_2 + d_1b_2 + c_2 + b_2 & \rightarrow t'_5 & \\
\hline
k_6(d_2,c_0,b_0) & = d_2c_0 + d_2b_0 + c_0 + b_0 & \rightarrow t'_6 & \\
k_7(d_2,c_1,b_1) & = d_2c_1 + d_2b_1 + d_2 & \rightarrow t'_7 & t'_6 + t'_7 + t'_8 \;\; = t_2 \\
k_8(d_2,c_2,b_2) & = d_2c_2 + d_2b_2 + d_2 + b_2 & \rightarrow t'_8 &
\end{array}$$

J 3-share Masked PRESENT S-box with 8-bit Fresh Masks

S : C56B90AD3EF84712 $= \mathsf{G} \circ \mathsf{F} \circ \mathsf{A}_1$
$\mathsf{A}_1(a,b,c,d)$: 894501CDAB6723EF : $x = a, \quad y = d, \quad z = b, \quad t = b + c + 1$

$\mathsf{F}(a,b,c,d)$: 08C43BF7192AE6D5 idential to F of the Midori's S-box given in Appendix I

$\mathsf{G}(a,b,c,d)$: 9C3672D805EB41AF

$$\begin{aligned}
x &= \mathsf{f}(a,b,c,d) = a + d + 1 \\
y &= \mathsf{g}(a,b,c,d) = cd + b + c \\
z &= \mathsf{h}(a,b,c,d) = bd + a + c \\
t &= \mathsf{k}(a,b,c,d) = cd + b + c + d + 1
\end{aligned}$$

$$\begin{array}{lll}
\mathsf{f}_0(a_0,d_0) & = a_0 + d_0 + 1 & \to x_0 \\
\mathsf{f}_1(a_1,d_1) & = a_1 + d_1 & \to x_1 \\
\mathsf{f}_2(a_2,d_2) & = a_2 + d_2 & \to x_2
\end{array}$$

$$\begin{array}{llll}
\mathsf{g}_0(d_0,c_0) & = d_0c_0 & \to y'_0 & \\
\mathsf{g}_1(d_0,c_1) & = d_0c_1 + c_1 & \to y'_1 & y'_0 + y'_1 + y'_2 = y_0 \\
\mathsf{g}_2(d_0,c_2,b_0) & = d_0c_2 + c_2 + b_0 & \to y'_2 & \\
\hline
\mathsf{g}_3(d_1,c_0) & = d_1c_0 + c_0 & \to y'_3 & \\
\mathsf{g}_4(d_1,c_1) & = d_1c_1 & \to y'_4 & y'_3 + y'_4 + y'_5 = y_1 \\
\mathsf{g}_5(d_1,c_2,b_1) & = d_1c_2 + c_2 + b_1 & \to y'_5 & \\
\hline
\mathsf{g}_6(d_2,c_0,b_2) & = d_2c_0 + b_2 & \to y'_6 & \\
\mathsf{g}_7(d_2,c_1) & = d_2c_1 & \to y'_7 & y'_6 + y'_7 + y'_8 = y_2 \\
\mathsf{g}_8(d_2,c_2) & = d_2c_2 + c_2 & \to y'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{h}_0(d_0,b_0,a_0) & = d_0b_0 + b_0 + a_0 & \to z'_0 & \\
\mathsf{h}_1(d_0,b_1) & = d_0b_1 & \to z'_1 & z'_0 + z'_1 + z'_2 = z_0 \\
\mathsf{h}_2(d_0,b_2,c_0) & = d_0b_2 + b_2 + c_0 & \to z'_2 & \\
\hline
\mathsf{h}_3(d_1,b_0,a_1) & = d_1b_0 + b_0 + a_1 & \to z'_3 & \\
\mathsf{h}_4(d_1,b_1) & = d_1b_1 & \to z'_4 & z'_3 + z'_4 + z'_5 = z_1 \\
\mathsf{h}_5(d_1,b_2,c_1) & = d_1b_2 + c_1 & \to z'_5 & \\
\hline
\mathsf{h}_6(d_2,b_0,c_0,a_2) & = d_2b_0 + c_2 + a_2 & \to z'_6 & \\
\mathsf{h}_7(d_2,b_1) & = d_2b_1 & \to z'_7 & z'_6 + z'_7 + z'_8 = z_2 \\
\mathsf{h}_8(d_2,b_2) & = d_2b_2 + b_2 & \to z'_8 &
\end{array}$$

$$\begin{array}{llll}
\mathsf{k}_0(d_0,c_0) & = d_0c_0 + d_0 + 1 & \to t'_0 & \\
\mathsf{k}_1(d_0,c_1) & = d_0c_1 + c_1 & \to t'_1 & t'_0 + t'_1 + t'_2 = t_0 \\
\mathsf{k}_2(d_0,c_2,b_0) & = d_0c_2 + c_2 + b_0 & \to t'_2 & \\
\hline
\mathsf{k}_3(d_1,c_0) & = d_1c_0 + d_1 + c_0 & \to t'_3 & \\
\mathsf{k}_4(d_1,c_1) & = d_1c_1 & \to t'_4 & t'_3 + t'_4 + t'_5 = t_1 \\
\mathsf{k}_5(d_1,c_2,b_1) & = d_1c_2 + c_2 + b_1 & \to t'_5 & \\
\hline
\mathsf{k}_6(d_2,c_0,b_2) & = d_2c_0 + b_2 & \to t'_6 & \\
\mathsf{k}_7(d_2,c_1) & = d_2c_1 & \to t'_7 & t'_6 + t'_7 + t'_8 = t_2 \\
\mathsf{k}_8(d_2,c_2) & = d_2c_2 + d_2 + c_2 & \to t'_8 &
\end{array}$$

K 3-share Masked PRINCE S-box Inverse with 16-bit Fresh Masks

$S^{-1} : = H \circ G \circ F \circ A_1$

$A_1(a,b,c,d)$: `8293C6D70A1B4E5F` : $x = b, \quad y = a, \quad z = c, \quad t = a + d + 1$

$F(a,b,c,d)$: `C480E6A2D519B37F`

$x = f(a,b,c,d) = d$

$y = g(a,b,c,d) = c$

$z = h(a,b,c,d) = cd + b + 1$

$t = k(a,b,c,d) = bd + a + 1$

$$\begin{array}{llr}
d_0 \rightarrow x'_0 & x'_0 + r_0 & \rightarrow x_0 \\
d_1 \rightarrow x'_1 & x'_1 + r_1 & \rightarrow x_1 \\
d_2 \rightarrow x'_2 & x'_2 + r_0 + r_1 & \rightarrow x_2 \\
 & & \\
c_0 \rightarrow y'_0 & y'_0 + r_2 & \rightarrow y_0 \\
c_1 \rightarrow y'_1 & y'_1 + r_3 & \rightarrow y_1 \\
c_2 \rightarrow y'_2 & y'_2 + r_2 + r_3 & \rightarrow y_2
\end{array}$$

$$\begin{array}{lllll}
h_0(d_0, c_0) & = d_0c_0 + 1 & \rightarrow z'_0 & & \\
h_1(d_0, c_1, b_1) & = d_0c_1 + c_1 + b_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 + r_4 & \rightarrow z_0 \\
h_2(d_0, c_2) & = d_0c_2 + c_2 & \rightarrow z'_2 & & \\
\hline
h_3(d_1, c_0, b_0) & = d_1c_0 + b_0 & \rightarrow z'_3 & & \\
h_4(d_1, c_1) & = d_1c_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 + r_5 & \rightarrow z_1 \\
h_5(d_1, c_2) & = d_1c_2 + c_2 & \rightarrow z'_5 & & \\
\hline
h_6(d_2, c_0) & = d_2c_0 & \rightarrow z'_6 & & \\
h_7(d_2, c_1) & = d_2c_1 + c_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 + r_4 + r_5 & \rightarrow z_2 \\
h_8(d_2, c_2, b_2) & = d_2c_2 + b_2 & \rightarrow z'_8 & &
\end{array}$$

$$\begin{array}{lllll}
k_0(d_0, b_0, a_0) & = d_0b_0 + d_0 + b_0 + a_0 + 1 & \rightarrow t'_0 & & \\
k_1(d_0, b_1) & = d_0b_1 + d_0 + b_1 & \rightarrow t'_1 & t'_0 + t'_1 + t'_2 + r_6 & \rightarrow t_0 \\
k_2(d_0, b_2) & = d_0b_2 & \rightarrow t'_2 & & \\
\hline
k_3(d_1, b_0, a_2) & = d_1b_0 + b_0 + a_2 & \rightarrow t'_3 & & \\
k_4(d_1, b_1) & = d_1b_1 & \rightarrow t'_4 & t'_3 + t'_4 + t'_5 + r_7 & \rightarrow t_1 \\
k_5(d_1, b_2) & = d_1b_2 + b_2 & \rightarrow t'_5 & & \\
\hline
k_6(d_2, b_0) & = d_2b_0 & \rightarrow t'_6 & & \\
k_7(d_2, b_1) & = d_2b_1 + b_1 & \rightarrow t'_7 & t'_6 + t'_7 + t'_8 + r_6 + r_7 & \rightarrow t_2 \\
k_8(d_2, b_2, a_1) & = d_2b_2 + b_2 + a_1 & \rightarrow t'_8 & &
\end{array}$$

$\mathsf{G}(a,b,c,d)$: `08C43BF72A6ED591`

$$
\begin{aligned}
x &= \mathsf{f}(a,b,c,d) = c\\
y &= \mathsf{g}(a,b,c,d) = c + d\\
z &= \mathsf{h}(a,b,c,d) = cd + b\\
t &= \mathsf{k}(a,b,c,d) = bd + cd + a + b
\end{aligned}
$$

$$
\begin{array}{lllll}
 & & c_0 \rightarrow x'_0 & x'_0 + r_8 & \rightarrow x_0\\
 & & c_1 \rightarrow x'_1 & x'_1 + r_9 & \rightarrow x_1\\
 & & c_2 \rightarrow x'_2 & x'_2 + r_8 + r_9 & \rightarrow x_2\\
 & & & & \\
\mathsf{g}_0(c_0,d_0) & = c_0 + d_0 & \rightarrow y'_0 & y'_0 + r_{10} & \rightarrow y_0\\
\mathsf{g}_1(c_1,d_1) & = c_1 + d_1 & \rightarrow y'_1 & y'_1 + r_{11} & \rightarrow y_1\\
\mathsf{g}_2(c_2,d_2) & = c_2 + d_2 & \rightarrow y'_2 & y'_2 + r_{10} + r_{11} & \rightarrow y_2\\
 & & & & \\
\mathsf{h}_0(d_0,c_0) & = d_0c_0 + d_0 + c_0 & \rightarrow z'_0 & & \\
\mathsf{h}_1(d_0,c_1) & = d_0c_1 & \rightarrow z'_1 & z'_0 + z'_1 + z'_2 + r_{12} & \rightarrow z_0\\
\mathsf{h}_2(d_0,c_2,b_2) & = d_0c_2 + d_0 + c_2 + b_2 & \rightarrow z'_2 & & \\
\hline
\mathsf{h}_3(d_1,c_0) & = d_1c_0 + c_0 & \rightarrow z'_3 & & \\
\mathsf{h}_4(d_1,c_1) & = d_1c_1 & \rightarrow z'_4 & z'_3 + z'_4 + z'_5 + r_{13} & \rightarrow z_1\\
\mathsf{h}_5(d_1,c_2,b_1) & = d_1c_2 + b_1 & \rightarrow z'_5 & & \\
\hline
\mathsf{h}_6(d_2,c_0,b_0) & = d_2c_0 + b_0 & \rightarrow z'_6 & & \\
\mathsf{h}_7(d_2,c_1) & = d_2c_1 & \rightarrow z'_7 & z'_6 + z'_7 + z'_8 + r_{12} + r_{13} & \rightarrow z_2\\
\mathsf{h}_8(d_2,c_2) & = d_2c_2 + c_2 & \rightarrow z'_8 & & \\
 & & & & \\
\mathsf{k}_0(d_0,c_0,b_0,a_0) & = d_0c_0 + d_0b_0 + a_0 & \rightarrow t'_0 & & \\
\mathsf{k}_1(d_0,c_1,b_1) & = d_0c_1 + d_0b_1 + c_1 + b_1 & \rightarrow t'_1 & t'_0 + t'_1 + t'_2 + r_{14} & \rightarrow t_0\\
\mathsf{k}_2(d_0,c_2,b_2) & = d_0c_2 + d_0b_2 + b_2 & \rightarrow t'_2 & & \\
\hline
\mathsf{k}_3(d_1,c_0,b_0,a_1) & = d_1c_0 + d_1b_0 + d_1 + a_1 & \rightarrow t'_3 & & \\
\mathsf{k}_4(d_1,c_1,b_1) & = d_1c_1 + d_1b_1 + c_1 & \rightarrow t'_4 & t'_3 + t'_4 + t'_5 + r_{15} & \rightarrow t_1\\
\mathsf{k}_5(d_1,c_2,b_2) & = d_1c_2 + d_1b_2 + d_1 + c_2 + b_2 & \rightarrow t'_5 & & \\
\hline
\mathsf{k}_6(d_2,c_0,b_0,a_2) & = d_2c_0 + d_2b_0 + b_0 + a_2 & \rightarrow t'_6 & & \\
\mathsf{k}_7(d_2,c_1,b_1) & = d_2c_1 + d_2b_1 + d_2 & \rightarrow t'_7 & t'_6 + t'_7 + t'_8 + r_{14} + r_{15} & \rightarrow t_2\\
\mathsf{k}_8(d_2,c_2,b_2) & = d_2c_2 + d_2b_2 + d_2 + c_2 + b_2 & \rightarrow t'_8 & &
\end{array}
$$

$H(a,b,c,d)$: `21748BDE65039AFC`

$$\begin{aligned}
x &= f(a,b,c,d) = bd + cd + a + b\\
y &= g(a,b,c,d) = bd + a + c + 1\\
z &= h(a,b,c,d) = cd + b + d\\
t &= k(a,b,c,d) = c
\end{aligned}$$

$$\begin{array}{lllll}
f_0(d_0,c_0,b_0,a_0) & = d_0c_0 + d_0b_0 + a_0 & \to x'_0 & & \\
f_1(d_1,c_1,b_1) & = d_0c_1 + d_0b_1 + c_1 & \to x'_1 & x'_0 + x'_1 + x'_2 & = x_0\\
f_2(d_2,c_2,b_2) & = d_0c_2 + d_0b_2 + c_2 + b_2 & \to x'_2 & & \\
\hline
f_3(d_0,c_0,b_0,a_1) & = d_1c_0 + d_1b_0 + b_0 + a_1 & \to x'_3 & & \\
f_4(d_1,c_1,b_1) & = d_1c_1 + d_1b_1 & \to x'_4 & x'_3 + x'_4 + x'_5 & = x_1\\
f_5(d_2,c_2,b_2) & = d_1c_2 + d_1b_2 + c_2 + b_2 & \to x'_5 & & \\
\hline
f_6(d_0,c_0,b_0,a_2) & = d_2c_0 + d_2b_0 + d_2 + a_2 & \to x'_6 & & \\
f_7(d_1,c_1,b_1) & = d_2c_1 + d_2b_1 + c_1 + b_1 & \to x'_7 & x'_6 + x'_7 + x'_8 & = x_2\\
f_8(d_2,c_2,b_2) & = d_2c_2 + d_2b_2 + d_2 + b_2 & \to x'_8 & &
\end{array}$$

$$\begin{array}{lllll}
g_0(d_0,b_0,a_0) & = d_0b_0 + d_0 + b_0 + a_0 + 1 & \to y'_0 & & \\
g_1(d_0,b_1) & = d_0b_1 + d_0 + c_1 & \to y'_1 & y'_0 + y'_1 + y'_2 & = y_0\\
g_2(d_0,b_2) & = d_0b_2 & \to y'_2 & & \\
\hline
g_3(d_1,b_0,a_1) & = d_1b_0 + c_0 + a_1 & \to y'_3 & & \\
g_4(d_1,b_1) & = d_1b_1 & \to y'_4 & y'_3 + y'_4 + y'_5 & = y_1\\
g_5(d_1,b_2) & = d_1b_2 + b_2 & \to y'_5 & & \\
\hline
g_6(d_2,b_0,a_2) & = d_2b_0 + b_0 + a_2 & \to y'_6 & & \\
g_7(d_2,b_1) & = d_2b_1 & \to y'_7 & y'_6 + y'_7 + y'_8 & = y_2\\
g_8(d_2,b_2) & = d_2b_2 + b_2 + c_2 & \to y'_8 & &
\end{array}$$

$$\begin{array}{lllll}
h_0(d_0,c_0) & = d_0c_0 + c_0 & \to z'_0 & & \\
h_1(d_0,c_1) & = d_0c_1 & \to z'_1 & z'_0 + z'_1 + z'_2 & = z_0\\
h_2(d_0,c_2,b_2) & = d_0c_2 + d_0 + c_2 + b_2 & \to z'_2 & & \\
\hline
h_3(d_1,c_0) & = d_1c_0 & \to z'_3 & & \\
h_4(d_1,c_1,b_1) & = d_1c_1 + c_1 + b_1 & \to z'_4 & z'_3 + z'_4 + z'_5 & = z_1\\
h_5(d_1,c_2) & = d_1c_2 + d_1 + c_2 & \to z'_5 & & \\
\hline
h_6(d_2,c_0,b_0) & = d_2c_0 + d_2 + c_0 + b_0 & \to z'_6 & & \\
h_7(d_2,c_1) & = d_2c_1 + d_2 + c_1 & \to z'_7 & z'_6 + z'_7 + z'_8 & = z_2\\
h_8(d_2,c_2) & = d_2c_2 + d_2 & \to z'_8 & &
\end{array}$$

$$\begin{array}{r}
c_0 \to t_0\\
c_1 \to t_1\\
c_2 \to t_2
\end{array}$$

L Fresh Mask-reuse in PRINCE S-box

As explained in Section 4.5, the PRINCE S-box (resp. its inverse) needs to be decomposed to three quadratic bijections as $H \circ G \circ F \circ A_1$. Therefore, we need to use 8-bit fresh mask r_1 when we compose G with $F \circ A_1$ and another 8-bit fresh mask r_2 when composing with H. However, since r_1 and r_2 are required in different clock cycles (indeed with 2 clock cycles distance) if a fully-pipeline design is made, we can provide r_1 and r_2 using the same source of randomness which is updated at every clock cycle, i.e., 8-bit fresh randomness per clock cycle. In order to provide evidence for the security of such an optimization, we constructed a test circuit as shown in Figure 7, which emulates the pipeline architecture. More precisely, two S-boxes are performed with 2 clock cycles distance. Hence, 8-bit r_2 used for the second composition of the first S-box is re-used by the first stage of the second S-box. After synthesizing the circuit, which receives 24 fresh mask bits and an 8-bit input and provides an 8-bit output (both shared with three shares), we gave the corresponding netlist to SILVER [KSM20], which confirmed its second-order security under glitch-extended probing model and uniformity of its output sharing. Note that this optimization is possible since each function F, G and H is individually second-order glitch-extended probing secure with uniform output sharing, and the fresh masks are only used to avoid multivariate leakages with respect to probes places on different functions.

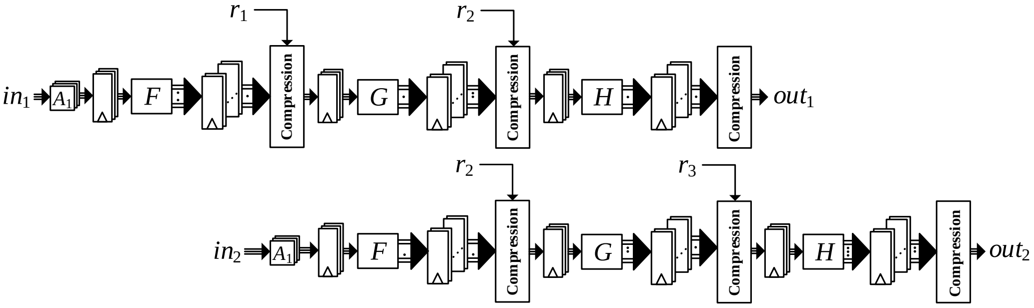
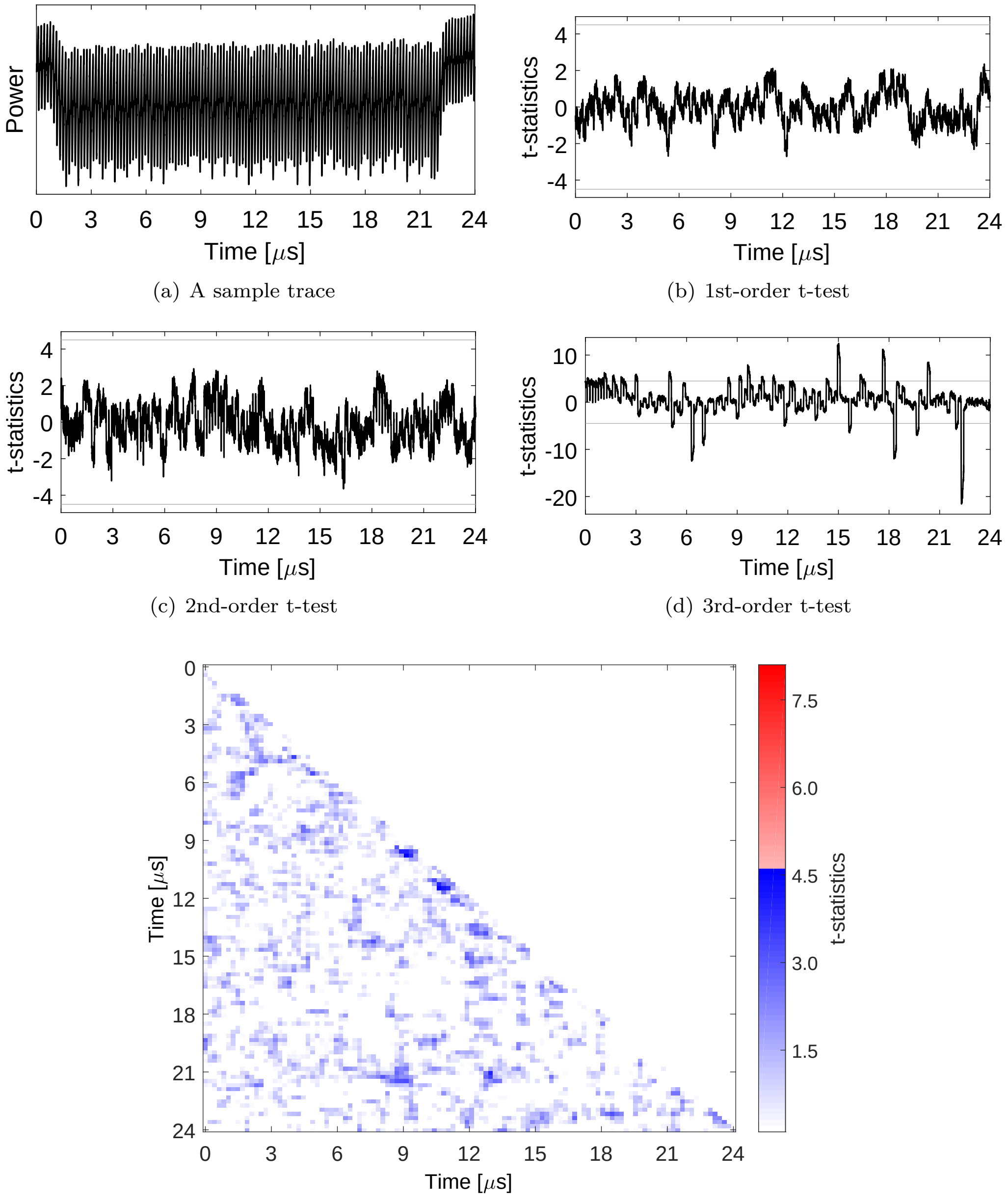


Figure 7: Emulation of fresh mask reuse in PRINCE S-box.

M Result of Experimental Analyses on Our Implementations (SKINNY, Midori, PRESENT, and PRINCE)



(a) A sample trace

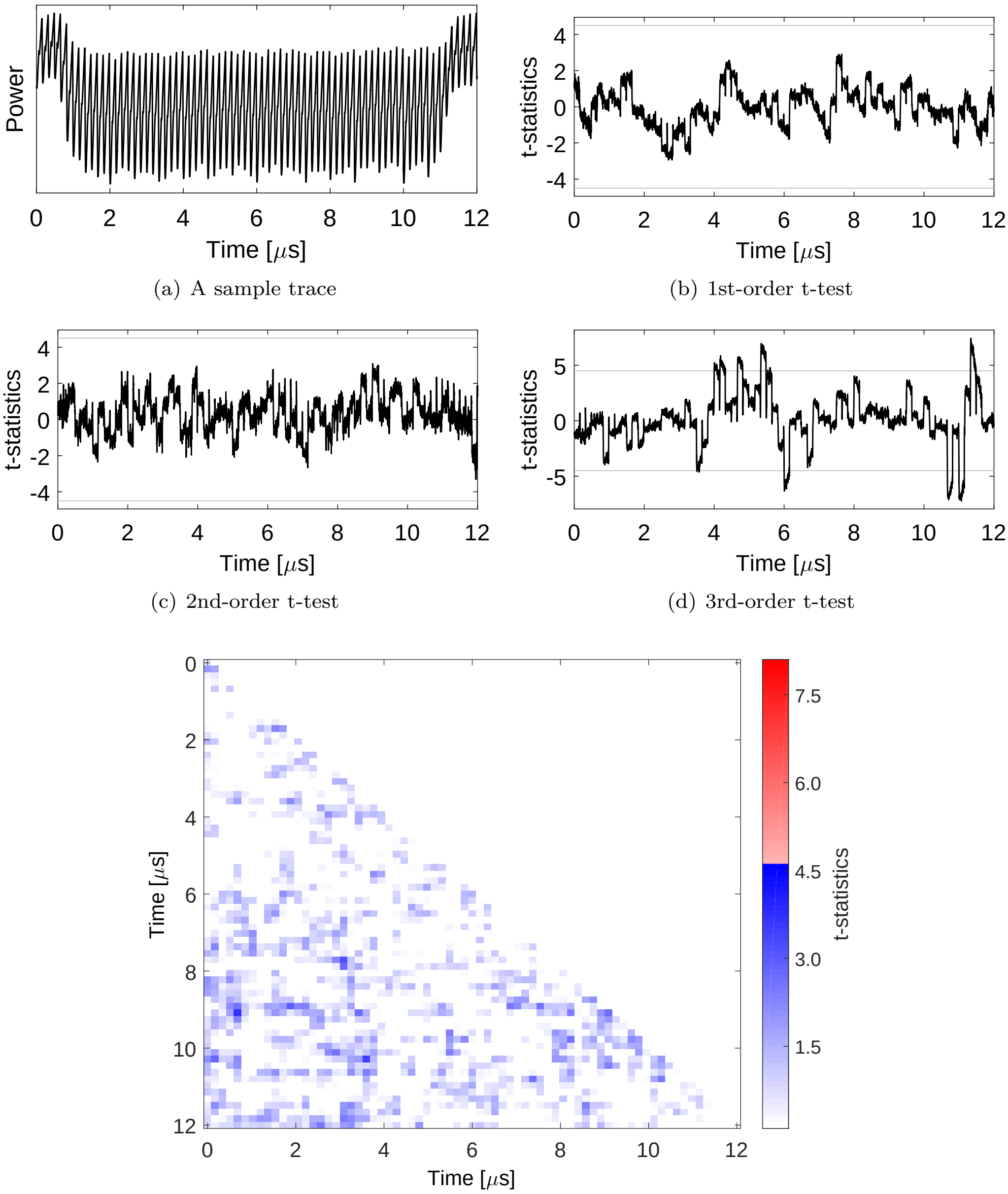
(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

(e) 2nd-order bivariate t-test

Figure 8: Experimental analysis of our second-order secure round-based SKINNY-64-64 encryption design, 100 million traces.



(a) A sample trace

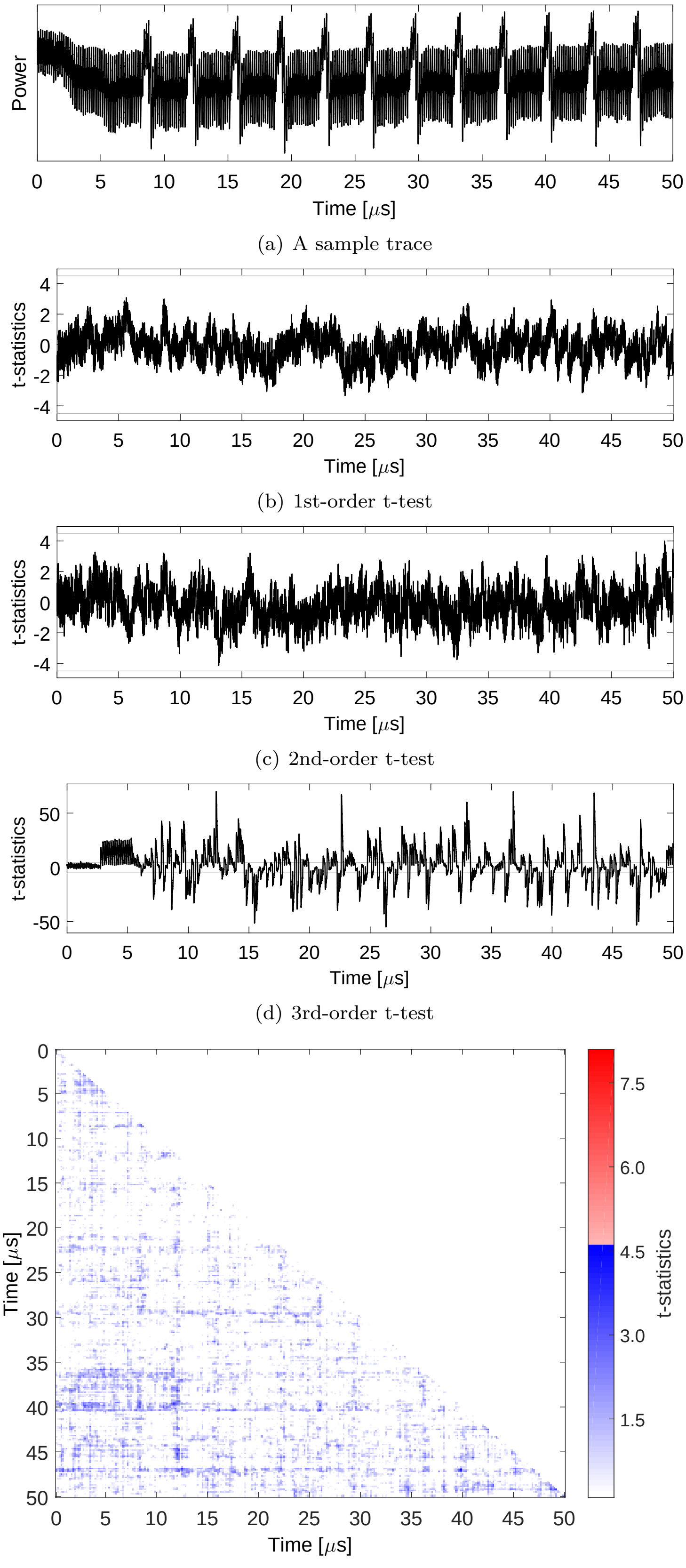
(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

(e) 2nd-order bivariate t-test

Figure 9: Experimental analysis of our second-order secure round-based Midori-64 encryption design, 100 million traces.



(a) A sample trace

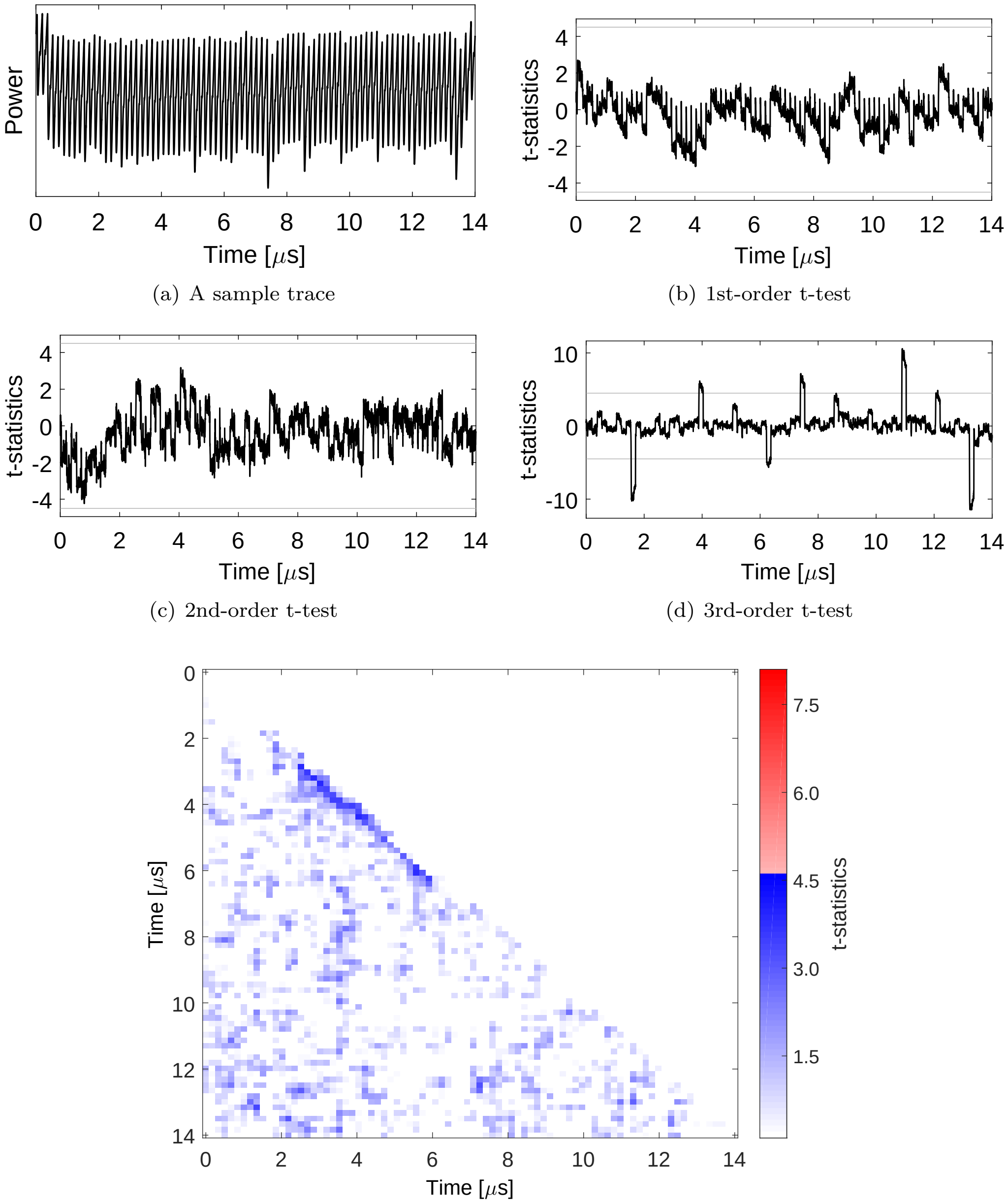
(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

(e) 2nd-order bivariate t-test

Figure 10: Experimental analysis of our second-order secure round-based PRESENT encryption design, 100 million traces.



(a) A sample trace

(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

(e) 2nd-order bivariate t-test

Figure 11: Experimental analysis of our second-order secure round-based PRINCE encryption design, 100 million traces.