IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 1, pp. 309–335. DOI:10.46586/tches.v2024.i1.309-335

Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency

Barbara Gigerl[1], Franz Klug[2], Stefan Mangard[1],
Florian Mendel[2] and Robert Primas[3]

[1] Graz University of Technology, Graz, Austria first.last@iaik.tugraz.at
[2] Infineon Technologies AG, Munich, Germany first.last@infineon.com
[3] Intel Labs, Hillsboro, USA first.last@intel.com

Abstract. Cryptographic devices in hostile environments can be vulnerable to physical attacks such as power analysis. Masking is a popular countermeasure against such attacks, which works by splitting every sensitive variable into $d+1$ randomized shares. The implementation cost of the masking countermeasure in hardware increases significantly with the masking order d, and protecting designs often results in a large overhead. One of the main drivers of the cost is the required amount of fresh randomness for masking the non-linear parts of a cipher. In the case of AES, first-order designs have been built without the need for any fresh randomness, but state-of-the-art higher-order designs still require a significant number of random bits per encryption. Attempts to reduce the randomness however often result in a considerable latency overhead, which is not favorable in practice. This raises the need for AES designs offering a decent performance tradeoff, which are efficient both in terms of required randomness and latency.

In this work, we present a second-order AES design with the minimal number of three shares, requiring only 3 200 random bits per encryption at a latency of 5 cycles per round. Our design represents a significant improvement compared to state-of-the-art designs that require more randomness and/or have a higher latency. The core of the design is an optimized 5-cycle AES S-box which needs 78 bits of fresh randomness. We use this S-box to construct a round-based AES design, for which we present a concept for sharing randomness across the S-boxes based on the *changing of the guards* (COTG) technique. We assess the security of our design in the probing model using a formal verification tool. Furthermore, we evaluate the practical side-channel resistance on an FPGA.

Keywords: Masking · AES · OpenTitan · Verification · Hardware

1 Introduction

Embedded devices running cryptographic hardware implementations need to be protected against physical attacks, such as differential power analysis [KJJ99], in which an attacker observes the power consumption of the device and uses the information to learn about secret values, e.g., the cryptographic key. Masking is a popular approach to protect against these attacks on implementation level, aiming at making the power consumption independent of the processed sensitive value [CJRR99]. To protect against a d-th order DPA attack, masking splits each sensitive value into $d+1$ shares such that an attacker probing up to d shares cannot recover the sensitive value.

Applying the masking countermeasure to a cryptographic hardware implementation comes with a considerable area overhead, which increases significantly with the masking

Received: 2023-07-15 Accepted: 2023-09-15 Published: 2023-12-04

order d [GIB18, NGPM22, SP06, ISW03, MRB18]. This overhead is not only caused by an increased area for the handling of the shares but also by the increased demand for fresh randomness that needs to be generated and distributed for masking the non-linear parts of the cipher. While the linear parts can be computed by evaluating them for each share individually, the non-linear parts, such as S-boxes, need to operate on several shares at once and, therefore, require randomness for refreshing to prevent unmasking of intermediate computation results, especially in the presence of glitches [GMK16, BDF⁺17, BBP⁺17, ISW03, RBN⁺15]. The need for fresh randomness usually goes hand-in-hand with an increased design area caused by the required random number generator (RNG) instances.

Methods to reduce randomness for a masked design have been studied extensively, especially focusing on AES. Since its selection by NIST in 2000, the AES [Nat01] has become an essential component for many cryptographic applications in industry. While the first proposed first-order sharings of the AES required about 3 000 to 5 000 random bits per encryption, there by now exist several works suggesting how to perform the computation without any fresh randomness [WM18, Sug19, SM21]. Compared to that, higher-order masked AES designs still require a significant amount of fresh randomness and area. While first works on second-order masking of the AES in hardware require more than three shares [CBR⁺15, BDRS21], in 2016, De Cnudde et al. [CRB⁺16] propose an S-box design with three shares which needs 162 fresh random bits and has a latency of five cycles, resulting in 19 440 random bits and 276 cycles per encryption. Gross et al. [GMK16] improve this situation by proposing a 5-cycle S-box protected by DOM (Domain-Oriented Masking) with only 84 random bits, resulting in 16 800 random bits and 200 cycles per encryption. Reducing the amount of randomness for a design however comes at the price of latency. Naturally, less randomness implies fewer capabilities to control the effect of glitches in a circuit, which in turn needs to be compensated for with more register stages, leading to a higher latency. For example, Dhooghe et al. [DSM22] recently show how to construct a second-order masking of the AES with only 1 012 fresh random bits per encryption, which however result in an S-box latency of 9 cycles per round. In recent years, low-latency has been generally identified as an important design goal for masked designs. Several works construct masked designs optimized for extremely low cycle counts [GIB18, SBHM20, NGPM22, SBB⁺22]. For example, Gross et al. [GIB18] propose a second-order masked low-latency DOM-AES S-box, which only needs two cycles per round but requires almost 900 000 random bits per encryption.

On architectural level, the performance of AES designs can be improved by employing a *parallel* or *round-based* design, in which the S-box is instantiated once per key/state byte, and all instances operate in parallel. By contrast, *serial* designs instantiate the S-box once, which is fed with a new key/state byte in every clock cycle. In parallel designs, the number of pipeline stages in the S-box determines the latency of an encryption round, and therefore, an S-box with a low latency is preferable. While most works in literature focus on serial designs, parallel designs have only been marginally addressed despite their clear practical relevance. For example, Google's OpenTitan project [low19], which aims at building an open root of trust (ROT) chip, includes a parallel AES architecture protected by DOM. They use a first-order version of the 5-cycle DOM AES S-box, which leads to an encryption latency of about 50 cycles per 128-bit block. One of the main challenges when constructing such designs is the high amount of randomness required per cycle, and in practice, it is not trivial to come up with RNGs allowing for such high demands of bandwidth yet keeping the required amount of randomness somewhat balanced per cycle.

Given that first-order protection often does not provide the required security level in practice, and serial designs are often not suitable for the desired performance, the goal is to build second-order designs targeting both low-randomness and low-latency.

1.1 Contributions

In practice, second-order masked AES designs should be efficient and provide a suitable tradeoff between area and latency, which clearly presumes a three-share design. However, state-of-the-art three-share designs are either optimized for low-latency or for low-randomness. Additionally, given the need for parallel designs, the demands of fresh randomness per cycle of these designs are unevenly distributed and often simply too high. We improve the situation by providing the following contributions:

- We present a second-order masked AES S-box based on DOM, which works with the minimum number of three shares, has a latency of only five cycles, and requires 78 bits of fresh randomness. In order to construct this S-box, we take the original DOM design as a starting point and demonstrate that fixing the flaw in higher-order DOM-*dep* multipliers, as identified by [MMSS19], is possible using more randomness. However, we also show that all DOM-*dep* multipliers can be replaced by more area-efficient adapted DOM-*indep* multipliers, which allows to perform one S-Box computation with 78 bits of fresh randomness. *(Section 3)*
- We propose an efficient parallel AES architecture similar to the one used in OpenTitan with an encryption latency of 51 cycles. We show how one encryption can be computed with only 3 200 bits by applying a special COTG-based concept for reusing randomness across all S-box instances for the key and plaintext. The 3 200 bits can smoothly be delivered by an RNG with a bandwidth of 64 fresh random bits per cycle. Given the 5-cycle latency per round, our design currently requires the least amount of fresh randomness in literature. *(Section 4)*
- We evaluate our AES design in terms of area and randomness and compare it to other state-of-the-art designs. *(Section 5)*
- Using a formal verification tool, we show the second-order security of our S-box design and investigate the security of our COTG-based sharing concept for key and plaintext for one round. We deploy our design on an FPGA and show that no leakage can be detected with up to 100 million traces. *(Section 6)*
- We provide access to the complete HDL code on GitHub[1].

2 Preliminaries

2.1 Notation

We denote the sharing of a sensitive variable X with $X = (X_0, X_1, X_2)$, i.e., the subscript index denotes a specific share. Every state byte in the AES is described as $s^{(i,j)}$, where i refers to the row index and j refers to the column index, according to the convention introduced in the AES specification [Nat01]. For example, $s_0^{(0,2)}$ refers to the first share (share domain 0) of the state byte in row 0, column 2. Every key byte in the AES is described as $k^{(i,j)}$ accordingly with the sharing $k^{(i,j)} = (k_0^{(i,j)}, k_1^{(i,j)}, k_2^{(i,j)})$.

2.2 Masking

Masking [CJRR99, GP99, ISW03] aims at defeating side-channel attacks that work by randomizing sensitive values by splitting them into $d + 1$ uniformly random shares. An adversary observing (probing) up to d shares cannot deduce any information about the sensitive value. In classical Boolean masking, the sharing of a sensitive variable s given by $(s_0, s_1, ...s_d)$ must satisfy $s = s_0 \oplus s_1 ... \oplus s_d$. The shares $s_0, s_1, ...s_{d-1}$ are randomly

[1] https://github.com/barbara-gigerl/aes-secondorder-guards

sampled from a uniform distribution, while $s_d = s \oplus s_0 \oplus s_1 ... \oplus s_{d-1}$. For example, in a second-order masking scheme ($d = 2$), s is represented by the sharing (s_0, s_1, s_2) such that $s = s_0 \oplus s_1 \oplus s_2$. s_0 and s_1 are chosen uniformly at random and $s_2 = s_0 \oplus s_1$.

Implementing the masking countermeasure for non-linear functions such as the AES S-box, which computes the inversion in $GF(2^8)$, is especially challenging because they require combining all shares of a sensitive value in a secure and correct way. Hardware-related side-effects such as glitches and transitions need to be considered, which could reveal secret information in an otherwise secure masked implementation [MPG05, MPO05, ISW03]. Masking schemes for the AES S-box have been addressed frequently in literature [OMPR05, GMK16, CRB+16, RP10, MPL+11, SP06, DSM22, BDRS21]. Canright [Can05] presents a decomposition into $GF(2^4)$ and $GF(2^2)$ field elements to perform the inversion more efficiently, which has since then been the basis for many works on masking the AES, including DOM by Gross et al. [GMK16].

2.3 Security Verification of Masking

Empirical measurements are generally an important indicator for the practical security of a masked implementation. However, collecting power traces is usually cumbersome and error-prone, and the results heavily depend on the platform and measurement setup. Formal verification tools represent a complementary approach that allows the analysis of a masked implementation within a specific attacker model, such as the classic probing model [ISW03].

Rebecca [BGI+18] is a formal verification tool to prove the security of masked hardware implementations at any order. It examines the leakage of a given circuit by investigating each gate and determining whether the gate output correlates directly with the unshared sensitive value. Rebecca approximates this correlation using Fourier expansions of Boolean functions [O'D14] and checks for leaks using a SAT solver, making it feasible to verify larger constructions at the cost of accuracy. However, it has been shown that the rate of false positives (tool falsely reports leak) is very low, and false negatives (tool falsely reports no leak) are not possible at all [GPM23]. Other tools like SILVER [KSM20] determine this correlation by exhaustively computing the probability distribution of each gate, which allows a very accurate analysis, but it hardly applies to more complex circuits such as higher-order AES S-boxes [DSM22]. In this work, we will use Coco [GHP+21], a tool based on Rebecca. Coco applies the time-constrained probing model, allowing an adversary to place d probes on an arbitrary wire in the circuit. Each probe allows observing the value of the wire for one specific clock cycle, including transitions and glitches. A masked hardware implementation is considered dth-order secure if the adversary cannot learn any information about the sensitive value by combining the values of these probes.

2.4 Changing of the Guards (COTG)

Masked designs based on TI (Threshold Implementation) require non-completeness and uniformity to be first-order secure [NRR06], but obtaining a uniform output sharing of a masked S-box often requires explicit remasking with fresh randomness. The changing of the guards (COTG) concept was introduced by Daemen [Dae17] to achieve uniformity more efficiently by replacing this fresh randomness with unrelated parts of the cipher state. For example, considering a TI S-box function S and the respective component functions S_0, S_1, S_2 arranged in an S-box layer that maps the shared inputs a, b, c to the shared outputs A, B, C as follows (for $0 \leq i \leq 2$):

$$A_i = S_0(b_i, c_i) \qquad B_i = S_1(a_i, c_i) \qquad C_i = S_2(a_i, b_i)$$

If the sharings of A, B, C are not uniform one needs to perform resharing, which can either be done with fresh randomness or, as suggested by COTG, with another unrelated input share such as the one of the neighbor S-box (for $0 \leq i \leq 2$):

$$A_i = S_0(b_i, c_i) \oplus b_{i-1} \oplus c_{i-1} \qquad B_i = S_1(a_i, c_i) \oplus c_{i-1} \qquad C_i = S_2(a_i, b_i) \oplus b_{i-1}$$

The values of b_{-1} and c_{-1} need to be instantiated with fresh random values. COTG has been applied to several TI implementations including AES [DSM22, Sug19, SBM21, WM18, ADN+22, BDRS21], KETJE [ANR19], Ascon and Keyak [SD17], ARX ciphers [JPS18], and PRINCE [MMM21]. The original idea of COTG is to use the input bytes of the right neighbor S-box as guards and use fresh randomness for the last S-box that does not have a right neighbor. In our work, we propose a more complex selection of guards by precisely analyzing which other state bytes are unrelated and which are not, eliminating the explicit need for fresh randomness for the last S-box.

3 Efficiently Masking the AES S-box

In this section, we present a 5-stage pipelined AES S-box with three shares requiring only 78 bits of fresh randomness, which is currently the lowest amount of randomness required for 5-cycle latency. The second-order S-box design DOM [GMK16], which serves as the basis for our design, requires 104 random bits, while the 5-cycle TI-design of De Cnudde et al. [CRB+16] needs 162 random bits.

In Section 3.1, we describe DOM and the basic structure of their proposed S-box, which uses the Canright decomposition and performs the multiplications in $GF(2^2)$ and $GF(2^4)$ with DOM-*dep* and DOM-*indep* multipliers. In 2019, [MMSS19] pointed out a flaw in higher-order DOM-*dep* multipliers, which we revisit Section 3.2, and discuss a possible fix for this. Unfortunately, including this fix into the second-order S-box requires an additional 20 bits of fresh randomness, resulting in 104 bits in total. Therefore, in Section 3.3, we show how one can optimize the S-box design such that the DOM-*dep* multipliers are not needed anymore at all and can be replaced by three types of adapted versions of DOM-*indep* multipliers, resulting in a randomness-optimized S-box design. We check the second-order security of our S-box design with Coco and give details on the verification in Section 6.

3.1 DOM-based Masking of the AES S-box

In 2016, Gross et al. [GMK16] introduce DOM as a low-cost method to protect circuits against SCA at arbitrary protection orders. DOM is based on the idea of separating shares into independent domains and adding fresh randomness whenever terms from different domains are combined. They introduce a five-cycle variant of the AES S-box intended for high-speed encryption, which serves as the basis of our work and is also used in the OpenTitan project. The S-box design follows Canright's propositions [Can05].

For both the subfield multiplications, Gross et al. propose two masked multiplication gadgets. The second-order DOM-*indep* multiplier, which we will refer to DOM-*indep* multiplier (Type A), is used to multiply two independently shared field elements A with sharing (A_0, A_1, A_2), and B with sharing (B_0, B_1, B_2) using the random variables z_0, z_1, z_2. The resulting output sharing (C_0, C_1, C_2), with registers indicated by parenthesis, is:

$$C_0 = (A_0 \times B_0) \oplus (A_0 \times B_1 \oplus z_0) \oplus (A_0 \times B_2 \oplus z_1) \tag{1}$$
$$C_1 = (A_1 \times B_0 \oplus z_0) \oplus (A_1 \times B_1) \oplus (A_1 \times B_2 \oplus z_2) \tag{2}$$
$$C_2 = (A_2 \times B_0 \oplus z_1) \oplus (A_2 \times B_1 \oplus z_2) \oplus (A_2 \times B_2) \tag{3}$$

The multiplication works in three phases. First, in the *calculation* phase, shares of different domains (cross-domain multiplication) and shares of the same domain (inner-domain multiplication) are multiplied in the respective field. Cross-domain multiplication terms are then refreshed with three fresh random values in the *resharing* phase and stored into a register, while inner-domain terms do not need to be refreshed. In the *integration* phase, the multiplication terms of each component function are accumulated.

In case the multiplier inputs are not shared independently, e.g., when multiplying $A \times A$, one could simply use a DOM-*indep* multiplier and reshare one of its inputs, which however comes at the cost of additional randomness and a register stage. Therefore, Gross et al. propose the DOM-*dep* multiplier that uses a random blinding variable p with the sharing (p_0, p_1, p_2) to compute $A \times B = A \times (B + p) + (A \times p)$. A DOM-*indep* multiplier is used to compute $(A \times p)$, and therefore, the complete second-order DOM-*dep* multiplier requires six fresh random values.

Given these two multiplication gadgets, the 5-cycle S-box first converts the 8-bit input shares from the polynomial basis to the normal basis, inverts them in $GF(2^8)$ by decomposition into $GF(2^4)$ and $GF(2^2)$ field elements, and converts them back. More precisely, in Stage 1, the 8-bit input shares are converted using a linear mapping, which linearly combines the bits of a share within one domain each. Due to glitches, the output of the linear mapping might temporarily result in a related sharing, and therefore, a $GF(2^4)$ DOM-*dep* multiplier is used. In Stage 2, the resulting $GF(2^4)$ field elements are combined with the outputs of the square scalers, and glitches could temporarily produce a related input sharing, therefore requiring the use of a $GF(2^2)$ DOM-*dep* multiplier. In Stage 3, a similar situation occurs, and consequently, both $GF(2^2)$ multipliers must be DOM-*dep* multipliers. The last multipliers in Stage 4 take as an input the pipelined S-box inputs and the output of Stage 3, which are clearly independent of each other, and therefore, $GF(2^4)$ DOM-*indep* multipliers can be used. In Stage 5, the output shares are converted back to the polynomial basis using the inverse linear mapping.

3.2 Fixing the second-order DOM-*dep* multiplier

In a follow-up work, Moos et al. [MMSS19] point out a flaw in the DOM-*dep* multiplier for $d \geq 2$. Recall from the previous section that a DOM-*dep* multiplier computes $A \times B = A \times (B + p) + (A \times p)$. They show that DOM-*dep* multipliers are not secure in the presence of glitches by combining information about the individual shares of $A \times (B + p)$, and multiplication terms in the DOM-*indep* multiplier (Type A) computing $(A \times p)$. A second-order adversary possesses two probes. One probe is used to access the individual shares of $A \times (B + p)$, which includes $A_2 \times (B_0 \oplus p_0)$. The other probe is placed in the DOM-*indep* multiplier to access the shared subproducts of $(A \times p)$, which includes the cross-domain term $A_1 \times p_0$. By combining these two probed values and considering that the sharings of A and B are related, the adversary can derive information about the sensitive value A.

We propose a way to fix this issue by preventing the adversary from accessing $B_0 \oplus p_0$ directly by adding more randomness to it. More concretely, we refresh the term $B + p$ with a sharing of the zero-bit vector $(q_0, q_1, q_0 \oplus q_1)$ and store that value to a register. The computation performed is now $A \times B = A \times (B + p + 0) + (A \times p)$ with 0 being a shared into q_0 and q_1 such that $0 = q_0 \oplus q_1$. Hence, the first probe will only allow access to $A_2 \times (B_0 \oplus p_0 \oplus q_0)$, and no information about A can be inferred due to the random value q_0. The advantage of this solution compared to refreshing B and using a DOM-*indep* multiplier afterward is that no additional register stage is required. Nevertheless, for the fixed second-order DOM-*dep* $GF(2^2)$ multiplier, 16 instead of 12 random bits are needed, or 32 instead of 24 in the case of $GF(2^4)$.

We successfully verify with Coco that our proposed solution indeed solves the issue and is second-order probing-secure in the presence of glitches. Furthermore, we apply the formal verification tool SILVER [KSM20] to prove that our construction is secure under the 2nd-order PINI (Probe Isolating Non-Interference) [CS20] notion and can, therefore, trivially be composed.

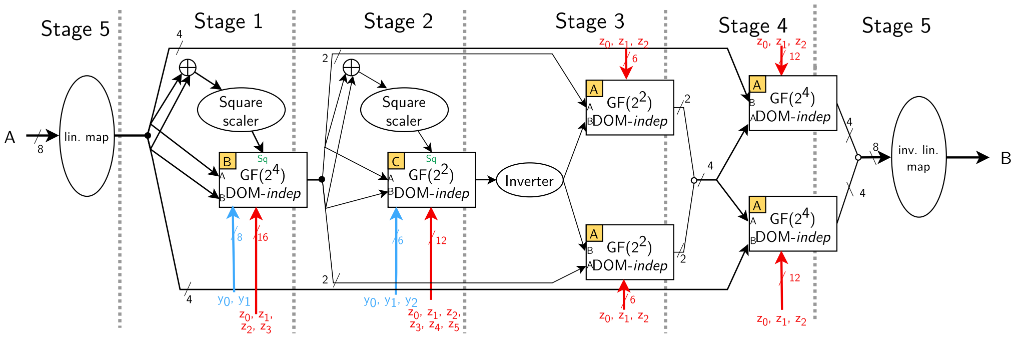


Figure 1: Our second-order AES DOM S-box with three shares and five register stages, requiring 78 bits of randomness. For simplicity, we draw a single line for all three shares. The ● indicates that a signal is split into a lower and upper part. The ○ indicates that the lower and upper parts of a signal are concatenated. Register stages are sketched by gray dotted lines. The respective type of each DOM-*indep* multiplier is indicated by a letter in the yellow box in the upper left corner, that is either Type A (Equations (1-3)), Type B (Equations (4-6)) or Type C (Equations (7-9)).

3.3 Optimized second-order DOM S-box

Integrating the proposed fix directly into the S-box design requires 104 bits of fresh randomness instead of the originally proposed 84 bits. For a complete AES encryption, this results in 20 800 required random bits instead of 16 800. Therefore, we propose a way to optimize this construction by replacing all DOM-*dep* multipliers with three types (Type A, B, C) of adapted DOM-*indep* multipliers, which are more efficient in both area and randomness. The resulting 78 bits of required fresh randomness are even less than in the originally proposed design. While the Type A multiplier refers to the original DOM-*indep* multiplier, the Type B and C multipliers work by additionally refreshing inner-domain multiplication terms besides cross-domain multiplication terms, which leads to an independent output sharing of a multiplier, and therefore allows the use of a DOM-*indep* multiplier in the next pipeline stage. Figure 1 gives an overview of the complete S-box design. Using Coco, we successfully verify the second-order security of our S-box. Now we describe the design considerations made in each stage in detail.

Linear mapping of input. Our goal is to replace the DOM-*dep* multiplier in Stage 1 with a DOM-*indep* multiplier. DOM-*indep* multipliers require that their inputs (the outputs of the linear map in our case) are shared independently. In general, glitches may temporarily cause a related sharing at the output of the linear map, and therefore, we need to store the output of the linear map in a register. Since our goal is a considerably low latency, we do not add an additional pipeline stage but move the computation of the linear map to the pipeline stage before. Considering the entire AES design, the complete linear layer (including the inverse linear map, ShiftRows, MixColumns, and AddRoundKey) is already computed in Stage 5 (c.f. Section 4), where we now also move the linear map of the SubBytes computation of the next round. Hence, the state registers in the design will not store the field elements in the polynomial base but the field elements in the normal base. From a security perspective, it is valid to do so because in Stage 5, only linear functions are computed, and adding the linear map to the end will not cause any additional leakage.

Multiplier in Stage 1 (Type B). We want to replace the DOM-*dep* multiplier in Stage 2 with a DOM-*indep* multiplier. The DOM-*indep* multiplier in Stage 2 only supports independent inputs, so the DOM-*indep* multiplier in Stage 1 needs to be modified such that it generates an independent output sharing. To do so, we need to perform the addition of the square scaler already in Stage 1, protect the inner-domain multiplication terms and use additional randomness on the cross-domain multiplication terms. The modified DOM-*indep* multiplier, which will be referred to as the Type B multiplier, used in Stage 1 with parenthesis again indicating registers, is given by:

$$C_0 = (A_0 \times B_0 \oplus Sq_0 \oplus y_0 \oplus y_1) \oplus (A_0 \times B_1 \oplus z_0 \oplus z_3) \oplus (A_0 \times B_2 \oplus z_1) \tag{4}$$
$$C_1 = (A_1 \times B_0 \oplus z_0) \oplus (A_1 \times B_1 \oplus Sq_1 \oplus y_1) \oplus (A_1 \times B_2 \oplus z_2) \tag{5}$$
$$C_2 = (A_2 \times B_0 \oplus z_1 \oplus z_3) \oplus (A_2 \times B_1 \oplus z_2) \oplus (A_2 \times B_2 \oplus Sq_2 \oplus y_0) \tag{6}$$

Note that this multiplier does not support dependent inputs, but independent inputs are obtained by storing the output of the linear map in a register. In the original design, the square scaler terms (Sq_0, Sq_1, Sq_2) were added to the output of the Stage 1 DOM-*dep* multiplier in the second pipeline stage. This can potentially cause a related input sharing to the multiplier in Stage 2 due to glitches. Therefore, we perform the addition of these terms already in Stage 1 by adding them to the inner-domain multiplication terms before the register layer. As a nice benefit, this saves registers to store the square scaler output in the original design.

Another issue is that the Stage 1 multiplier might temporarily only output the same-domain terms due to glitches if, e.g., the wire length of cross-domain terms is significantly longer. In that case, the Stage 2 multiplier, which multiplies the lower and higher two bits of the Stage 1 multiplier, might temporarily operate on related inputs. Therefore, we use 2 × 4 random bits y_0 and y_1 to also refresh the inner-domain terms. In order to maintain second-order probing security, the cross-domain terms need to be refreshed with an additional z_3 in this case. Otherwise, an attacker can place a probe in the calculation phase of the Stage 2 multiplier to get a combination of masks, which is used to protect the integration phase of the Stage 1 multiplier.

Multiplier in Stage 2 (Type C). We want to replace the DOM-*dep* multipliers in Stage 3 by a DOM-*indep* multiplier. The DOM-*indep* multiplier in Stage 3 only supports independent inputs, so the DOM-*indep* multiplier in this stage needs to be modified such that it generates an independent output sharing. To do so, we need to perform changes similar to Stage 1, including shifting the addition of square scaler terms and additional protection for inner-domain and cross-domain terms. In summary, the modified DOM-*indep* multiplier, which will be referred to as the Type C multiplier, used in Stage 2, with parenthesis indicating registers, is given by:

$$C_0 = (A_0 \times B_0 \oplus Sq_0 \oplus y_0 \oplus y_1) \oplus (A_0 \times B_1 \oplus z_0 \oplus z_3) \oplus (A_0 \times B_2 \oplus z_1 \oplus z_5) \tag{7}$$
$$C_1 = (A_1 \times B_0 \oplus z_0 \oplus z_4) \oplus (A_1 \times B_1 \oplus Sq_1 \oplus y_1 \oplus y_2) \oplus (A_1 \times B_2 \oplus z_2 \oplus z_5) \tag{8}$$
$$C_2 = (A_2 \times B_0 \oplus z_1 \oplus z_3) \oplus (A_2 \times B_1 \oplus z_2 \oplus z_4) \oplus (A_2 \times B_2 \oplus Sq_2 \oplus y_0 \oplus y_2) \tag{9}$$

Note that, also this multiplier does not support dependent inputs, but independent inputs are obtained by appropriate refreshing in the stage before. Compared to the multiplier in Stage 1 (Type B), we need more randomness for refreshing the multiplication terms. In total, 3 × 2 bits are needed for inner-domain terms (y_0, y_1, y_2), and 6 × 2 bits are needed for cross-domain terms $(z_0, z_1, z_2, z_3, z_4, z_5)$.

Multipliers in stages 3 and 4 (Type A). After performing these changes, the DOM-*dep* multiplier in Stage 3 can simply be replaced by the original DOM-*indep* multiplier (Type A) because independent inputs are obtained by refreshing in Stage 2. The multiplier in Stage 4 has originally been a DOM-*indep* multiplier and therefore, no further modifications are required there.

Table 1: Comparison of the amount of fresh randomness required for the insecure and fixed second-order DOM-*dep* multipliers, and the resulting insecure, fixed and optimized second-order DOM AES S-boxes. For the S-box constructions we give in brackets the amount of required random bits per stage.

Construction		Fresh randomness	Area
Insecure second-order DOM-*dep* [GMK16]	$GF(2^2)$	12 bit	N/A
	$GF(2^4)$	24 bit	
Fixed second-order DOM-*dep*	$GF(2^2)$	16 bit	
	$GF(2^4)$	32 bit	
Insecure second-order DOM AES S-box [GMK16]		84 bit (24/12/24/24)	N/A
Fixed second-order DOM AES S-box		104 bit (32/16/32/24)	4.37 kGE
Optimized second-order DOM AES S-box		78 bit (24/18/12/24)	4.29 kGE

3.4 Discussion

In Table 1 we compare the randomness properties of the different constructions. As stated by [GMK16], it requires 6/12 bits of fresh randomness for a $GF(2^2)/GF(2^4)$ DOM-*indep* multiplier. The insecure DOM-*dep* multiplier requires 12/24 bits of fresh randomness for $GF(2^2)/GF(2^4)$. The fixed version of the DOM-*dep* multiplier, which works with our fix, requires 16/32 bits of fresh randomness. The amount of 84 bits for the whole insecure S-box denotes to 24 bits in Stage 1, 12 bits in Stage 2, $2 \times 12 = 24$ bits in Stage 3, and 2×12 bits in Stage 4. When exchanging the DOM-*dep* multipliers in that design with our fixed multipliers, the final construction leads to a randomness consumption of 104 bits, implying an increase of 24%. More precisely, 32 bits of fresh randomness are now needed in Stage 1, 16 bits in Stage 2, 32 bits in Stage 3 and 24 bits in Stage 4. Our optimized second-order S-box design, which does not use any DOM-*dep* multipliers, has a lower randomness consumption of 78 bits and also a slightly lower area (4.29 kGE) compared to the originally proposed version.

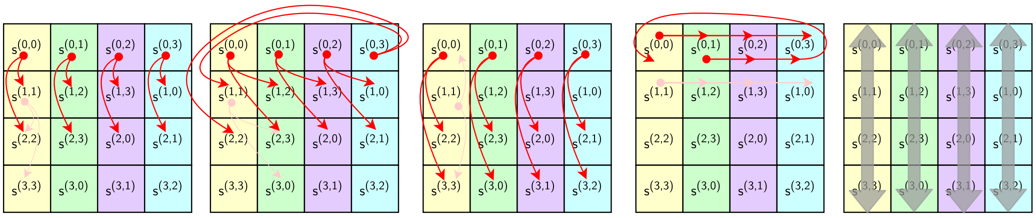
4 COTG-based Design of AES

Using the S-box design described in Section 3 directly in a masked AES implementation requires 15 600 bits of fresh randomness per encryption. In this section, we show how a COTG-based concept inspired by [Dae17] can be used to reduce this number to only 3 200. In general, each S-box requires 78 bits for refreshing the multiplication terms in the multipliers. Our main goal is to replace as many of these 78 bits by *guards*, i.e., shares of state bytes of another unrelated S-box, and use fresh randomness produced by an RNG where necessary, such that in total, connecting an RNG producing 64 bits of fresh randomness per cycle to the design is sufficient.

We give a general overview of our concept in Section 4.1. In Section 4.2, we give more details on the exact COTG-based SubBytes operation for the shared plaintext. In Section 4.3, we show how a similar concept applies to the key schedule. We verify the basic assumptions made for our concept with Coco, as described in more detail in Section 6.

4.1 Overview

The AES round function can be divided into four smaller *super boxes*, mapping a 32-bit input to a 32-bit output by applying SubBytes, MixColumns, AddRoundKey, and the second SubBytes function. The four input bytes of a super box are the columns of the state when viewed after ShiftRows. From a masking point of view, the non-linear SubBytes operation processes each state byte individually but combines the share domains. In contrast, the linear MixColumns operation combines the four state bytes of a super box but does this for each share domain individually.



Stage 1 Stage 2 Stage 3 Stage 4 Stage 5 (MixColumns)

Figure 2: Overview of the proposed COTG concept. The squares represent the 4x4 AES state grouped in four super boxes (=the state after ShiftRows). For a specific state byte (indicated by ●), the red arrow illustrates the other state bytes used as guards. In the last stage, we sketch the MixColumns operation combining all bytes of a super box.

These considerations suggest some general constraints regarding a COTG-based AES design. First, without COTG, the state bytes are kept isolated from each other until MixColumns, while with COTG, other state bytes are mixed in during the SubBytes operation in terms of randomness required by the multipliers. As noted by [BDZ20], this could change the diffusion properties of the masked cipher in an unfavorable way, for which we account with super box-wise resharing using fresh randomness before MixColumns similar to [DSM22]. Second, on the level of a single S-box, we need to choose guards for refreshing the multipliers such that they are always independent of the multipliers' inputs. This becomes even more complex considering that a multiplier input is usually just the output of another multiplier from the previous stage, which again directly relates to the guards used there.

Therefore, from the view of a single S-box (located in super box i) in our design we make the following decisions regarding which other state bytes can be used as guards for refreshing (we sketch this in Figure 2):

- MixColumns combines all state bytes of a super box, i.e., all guards used in all Stage 4 multipliers of the super box bytes are combined. Therefore, the guards need to be chosen from the three foreign super boxes $i+1, i+2, i+3$. To avoid changing diffusion properties, we refresh the inner-domain terms with fresh randomness.
- Taking the guards for Stage 4 from the three foreign super boxes leaves us with no choice but to ensure that the multiplier inputs are related to the domestic super box. The inputs are (a) the plain shares after the linear map (by default related to domestic super box i) and (b) the output of the Stage 3 multiplier. By choosing guards from the domestic super box i in combination with fresh randomness, we get independence here as well. In order to obtain the independence even in the presence of glitches, the inner-domain terms in Stage 3 are again refreshed with fresh randomness.
- The inputs of the Stage 3 multiplier are the outputs of stages 1 and 2. However, the guards of the Stage 3 multiplier are independent of any unmasked state byte because they are combined with fresh randomness. Hence, we can simply choose guards from the domestic super box for Stage 1 and guards from the neighbor super box for stage 2.

4.2 COTG for SubBytes of Plaintext

Choice of guards for Stage 4. Stage 5 of our design computes the complete linear layer, i.e., the inverse linear map, ShiftRows, MixColumns, AddRoundKey, and the linear map of SubBytes of the next round. Each operation is applied exactly once per share and does

Table 2: Assignment of guards and fresh randomness to refresh the inner- and cross-domain terms of the DOM-*indep* multipliers in our design. The operator $X[a : b]$ extracts the bits in range from b to (including) a from a given binary word X. The 64 bits of fresh randomness R given to the design in every cycle is arranged in rows $R0, R1, R2, R3$ of 16 bits each.

	DOM-*indep* multiplier					
	1	**2**	**3/1**	**3/2**	**4/1**	**4/2**
z_0	$s_0^{(i+1,j+1)}$	Ri[7:0]	$s_2^{(i+2,j+2)}$[5:0] $\oplus$ Ri[5:0]	$s_0^{(i+3,j+3)}$[5:0] $\oplus$ Ri[13:8]	$s_0^{(i,j+1)}$[3:0]	$s_1^{(i,j+2)}$[7:4]
z_1					$s_0^{(i,j+1)}$[7:4]	$s_2^{(i,j+3)}$[3:0]
z_2	Ri[7:0]				$s_1^{(i,j+2)}$[3:0]	$s_2^{(i,j+3)}$[7:4]
z_3			-	-	-	-
z_4	$s_1^{(i+2,j+2)}$ $\oplus$ Ri[15:8]	$s_0^{(i+1,j+2)}$ $\oplus$ Ri[15:8]	-	-	-	-
z_5			-	-	-	-
y_0	-		Ri[7:6]	Ri[7:6]	Ri[3:0]	Ri[11:8]
y_1	-		Ri[15:14]	Ri[15:14]	Ri[7:4]	Ri[15:12]
y_2	-	$s_1^{(i+2,j+3)}$[1:0]	-	-	-	-

not combine shares of different domains. The linear mappings of the S-box mix the bits of a share byte, and AddRoundKey combines the state bytes bitwise with unrelated key material. MixColumns however combines the bytes of each super box in the design, or, when viewed from a masking perspective, combines the refreshed multiplication terms of the Stage 4 multipliers of the four super box bytes. Due to glitches, every masked multiplication term can be observed individually, and thus, all their combinations. In order to refresh these multiplication terms, which is done in the two DOM-*indep* multipliers using z_0, z_1, and z_2, we instantiate 24 bits of guards. As shown in Table 2, we use *guards of three different foreign super boxes with rotating share domains* for this purpose. For example, the Stage 4 multipliers of the first two super boxes use the following state bytes as guards:

$s^{(0,0)}$	$: s_0^{(0,1)}, s_1^{(0,2)}, s_2^{(0,3)}$	$s^{(0,1)}$	$: s_0^{(0,2)}, s_1^{(0,3)}, s_2^{(0,0)}$
$s^{(1,1)}$	$: s_0^{(1,2)}, s_1^{(1,3)}, s_2^{(1,0)}$	$s^{(1,2)}$	$: s_0^{(1,3)}, s_1^{(1,0)}, s_2^{(1,1)}$
$s^{(2,2)}$	$: s_0^{(2,3)}, s_1^{(2,0)}, s_2^{(2,1)}$	$s^{(2,3)}$	$: s_0^{(2,0)}, s_1^{(2,1)}, s_2^{(2,2)}$
$s^{(3,3)}$	$: s_0^{(3,0)}, s_1^{(3,1)}, s_2^{(3,2)}$	$s^{(3,0)}$	$: s_0^{(3,1)}, s_1^{(3,2)}, s_2^{(3,3)}$

Rotating share domains means that we use share domain 0 for the first guard, share domain 1 for the second, and share domain 2 for the third. We cannot use the same guard domain, e.g., domain 0, for all guards because that would lead to many Stage 4 multiplication terms being refreshed with the same guard. By rotating the domains, every state byte share is used exactly once in the Stage 4 multipliers. Assume that the guards for an S-box are not distributed across super boxes, but that for super box i, we use state bytes of the same domestic super box i. That implies that $s^{(0,0)}$ uses $s_2^{(3,3)}$, $s^{(1,1)}$ uses $s_1^{(3,3)}$ and $s^{(2,2)}$ uses $s_0^{(3,3)}$ as a guard, and hence, in MixColumns, state byte $s^{(3,3)}$ is unmasked. The same holds when super box i uses state bytes of the same foreign super box. Therefore, the guards need to originate from *three different foreign superboxes*. At the same time, it is important to note that every MixColumns operation combines shares of exactly one share domain of each super box. For example, super box 0 uses guards from super box 1, but all of share domain 0. That is important to prevent an attacker from placing two probes in the MixColumns operations of different super boxes. Additionally, we use the 64 bits of fresh randomness produced by the RNG to refresh the inner-domain terms with y_0 and y_1.

Similar to [DSM22], instead of refreshing the complete state (which would require 256 bits of fresh randomness), we align the 64 bits of fresh randomness into four rows $R0, R1, R2, R3$ of 16 bits each such that the randomness is reused in every super box.

Choice of guards for Stage 3. The Stage 4 DOM-*indep* multipliers multiply (a) the plain input shares of the S-box after the linear map, with (b) the output of the Stage 3 multipliers. The guards used in Stage 4 must be independent of both (a) and (b). In the case of (a), independence between the plain input shares of a specific S-box and state bytes of other super boxes is naturally given. In the case of (b), the independence is determined by the output of the multipliers in Stage 3 and, therefore, by the guards used in Stage 3. In Stage 3, $2 \times 6 = 12$ bits are required for refreshing cross-domain multiplication terms (z_0, z_1, z_2 in multipliers 3/1 and 3/2), and additionally, 2×4 bit are required for refreshing inner-domain multiplication terms (y_0, y_1) to achieve that even in the presence of glitches, the inputs to Stage 4 are independent. In total, this makes 20 bits, which can however be reduced to 16 bits because, in the multiplier 3/1 and the multiplier 3/2, the same values for y_0 and y_1 can be used.

In summary, we therefore need to come up with 16 bits of randomness per S-box. Similar to Stage 4, we again arrange the 64 bits of fresh randomness generated in this cycle by the RNG in four rows of 16 bits and re-use this randomness in every super box. Verification with Coco reveals that while this is valid for inner-domain terms (y_0, y_1), the cross-domain terms must be refreshed with unique randomness (z_0, z_1, z_2).As shown in Table 2, we use a trick to generate unique terms by combining the fresh randomness from the RNG with guards taken from the domestic super box. For example, the multiplier 3/1 of the first two super boxes uses the following values for z_0, z_1, z_2:

$$
\begin{array}{llll}
s^{(0,0)} & : s_2^{(2,2)}[5{:}0] \oplus R0[5{:}0] & s^{(0,1)} & : s_2^{(2,3)}[5{:}0] \oplus R0[5{:}0] \\
s^{(1,1)} & : s_2^{(0,0)}[5{:}0] \oplus R1[5{:}0] & s^{(1,2)} & : s_2^{(3,0)}[5{:}0] \oplus R1[5{:}0] \\
s^{(2,2)} & : s_2^{(1,1)}[5{:}0] \oplus R2[5{:}0] & s^{(2,3)} & : s_2^{(0,1)}[5{:}0] \oplus R2[5{:}0] \\
s^{(3,3)} & : s_2^{(2,2)}[5{:}0] \oplus R3[5{:}0] & s^{(3,0)} & : s_2^{(1,2)}[5{:}0] \oplus R3[5{:}0]
\end{array}
$$

By doing so, the uniqueness of the term is given by Ri within the super box, and by the guards across super boxes, and every Stage 3 multiplier in all S-boxes uses unique values to refresh the multiplication terms. Similar to Stage 4, we perform share domain rotation by using share 2 for the 3/1 multipliers and share 0 for the 3/2 multipliers in order to achieve that within a super box, two different shares of a state byte are used as guards.

Choice of guards for Stage 2. The Stage 3 DOM-*indep* multipliers multiply the output of the Stage 2 multiplier with the output of the Stage 1 multiplier. The guards used in Stage 3 are inherently independent of these because the randomness generated by the RNG in Stage 3, which is used to *mask* the guards, is only used in that cycle. Therefore, the choice of guards for stages 1 and 2 is relatively unconstrained as long as they are independent of each other (otherwise, a DOM-*dep* multiplier would need to be used). In Stage 2, 18 bits are required for refreshing cross-domain multiplication terms ($z_0, z_1, z_2, z_3, z_4, z_5$) and inner-domain multiplication terms (y_0, y_1, y_2). Using an analysis with Coco, we find out that for second-order probing security, z_0, z_1, z_2, z_3 can be re-used across super boxes, while the rest of the values need to be unique. As shown in Table 2, we apply a similar trick as in Stage 3 to generate this uniqueness: We use the fresh randomness generated by the RNG, distribute it over the columns of the state, and re-mask it with guards as necessary to obtain a unique random value. For example, the values used for refreshing in the Stage

2 multipliers are:

$s^{(0,0)}$: $R0[7{:}0], s_0^{(1,2)} \oplus R0[15{:}8], s_1^{(2,3)}[1{:}0]$	$s^{(0,1)}$: $R0[7{:}0], s_0^{(1,3)} \oplus R0[15{:}8], s_1^{(2,0)}[1{:}0]$
$s^{(1,1)}$: $R1[7{:}0], s_0^{(2,3)} \oplus R1[15{:}8], s_1^{(3,0)}[1{:}0]$	$s^{(1,2)}$: $R1[7{:}0], s_0^{(2,0)} \oplus R1[15{:}8], s_1^{(3,1)}[1{:}0]$
$s^{(2,2)}$: $R2[7{:}0], s_0^{(3,0)} \oplus R2[15{:}8], s_1^{(0,1)}[1{:}0]$	$s^{(2,3)}$: $R2[7{:}0], s_0^{(3,1)} \oplus R2[15{:}8], s_1^{(0,2)}[1{:}0]$
$s^{(3,3)}$: $R3[7{:}0], s_0^{(0,1)} \oplus R3[15{:}8], s_1^{(1,2)}[1{:}0]$	$s^{(3,0)}$: $R3[7{:}0], s_0^{(0,2)} \oplus R3[15{:}8], s_1^{(1,3)}[1{:}0]$

Note that we again perform share domain rotation, i.e., every byte in a superbox uses guards from two different domains.

Choice of guards for Stage 1. The Stage 2 DOM-*indep* multiplier multiplies the four most significant bits of the Stage 1 multiplier output with the four least significant bits. The 24 bits required for refreshing in the Stage 1 multiplier hence need to be chosen independently of the guards in Stage 2. Analysis with Coco reveals that in this situation, the values for z_0, z_1, z_4 and z_5 need to be unique, while z_2 and z_3 can again be re-used across super boxes. As shown in Table 2, for z_2 and z_3 we use a byte of fresh randomness from the RNG, which is re-used once per super box. For z_4 and z_5 we use another byte of fresh randomness from the RNG, which is also re-used once per super box, but made unique by re-masking with a guard from the domestic super box. For z_0 and z_1 we need a unique value as well, however, the 16 bits of randomness available are already used up, and therefore, we directly use as a guard a state byte from the same domestic super box. For example, the values used for refreshing in the Stage 1 multipliers are:

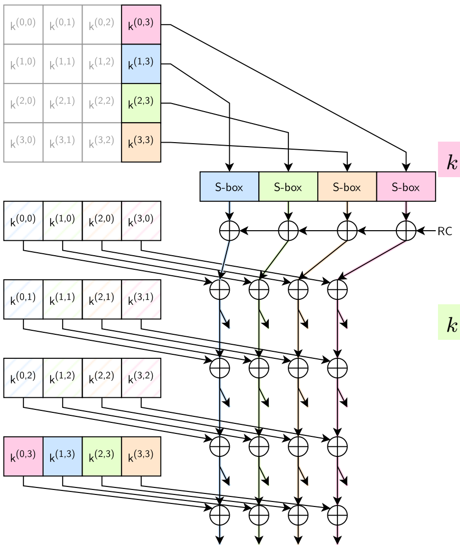
$s^{(0,0)}$: $s_0^{(1,1)}, R0[7{:}0], s_1^{(2,2)} \oplus R0[15{:}8]$	$s^{(0,1)}$: $s_0^{(1,2)}, R0[7{:}0], s_1^{(2,3)} \oplus R0[15{:}8]$
$s^{(1,1)}$: $s_0^{(2,2)}, R1[7{:}0], s_1^{(3,3)} \oplus R1[15{:}8]$	$s^{(1,2)}$: $s_0^{(2,3)}, R1[7{:}0], s_1^{(3,0)} \oplus R1[15{:}8]$
$s^{(2,2)}$: $s_0^{(3,3)}, R2[7{:}0], s_1^{(0,0)} \oplus R2[15{:}8]$	$s^{(2,3)}$: $s_0^{(3,0)}, R2[7{:}0], s_1^{(0,1)} \oplus R2[15{:}8]$
$s^{(3,3)}$: $s_0^{(0,0)}, R3[7{:}0], s_1^{(1,1)} \oplus R3[15{:}8]$	$s^{(3,0)}$: $s_0^{(0,1)}, R3[7{:}0], s_1^{(1,2)} \oplus R3[15{:}8]$

4.3 COTG for SubWord of Key Schedule

In our AES design, we use the same shared S-box design for the key as for the plaintext. Masking the key schedule using COTG is however much simpler than for the plaintext because only four key bytes are transformed using SubWord, which is comprised of four S-boxes, and no MixColumns operation is performed during the key schedule (c.f. Figure 3). Therefore, we first identify key state bytes that cannot be used in a straightforward way as guards in the SubWord operation of the key schedule, that are, the set of key bytes combined with each SubWord input byte. This set includes the SubWord input bytes $k^{(0,3)}, k^{(1,3)}, k^{(2,3)}, k^{3,3}$ themselves, and then for each byte, the three other key bytes added to the S-box output later in the key schedule. For example, for $k^{(0,3)}$ we do not use $k^{(3,0)}, k^{(3,1)}, k^{(3,2)}$ as guards. In Figure 3, we mark the key bytes not used as guards for a specific S-box with stripes of the respective color.

For each of the four input bytes, we can then simply assign the remaining key state bytes as guards for the respective S-box and perform share-domain rotation on that. Using this technique, we can obtain the second-order probing security of the construction. An adversary placing two probes in the same S-box of the key schedule cannot probe a complete sharing of a guard byte because per S-box, at most one share of a guard is used. With two probes in two different S-boxes, an adversary can therefore at most probe two out of three shares.

The RNG connected to the AES design produces 64 bits of fresh randomness per cycle for encrypting the plaintext. However, in Stage 5 of computing the S-box for the plaintext,



(a)

$k^{(0,3)}$: $k_0^{(0,0)}, k_1^{(0,1)}, k_2^{(0,2)},$
$k_1^{(1,0)}, k_2^{(1,1)}, k_0^{(1,2)},$
$k_2^{(2,0)}, k_0^{(2,1)}, k_1^{(2,2)},$
$R[15{:}0]$

$k^{(1,3)}$: $k_0^{(1,0)}, k_1^{(1,1)}, k_2^{(1,2)},$
$k_1^{(2,0)}, k_2^{(2,1)}, k_0^{(2,2)},$
$k_2^{(3,0)}, k_0^{(3,1)}, k_1^{(3,2)},$
$R[31{:}16]$

$k^{(2,3)}$: $k_0^{(2,0)}, k_1^{(2,1)}, k_2^{(2,2)},$
$k_1^{(3,0)}, k_2^{(3,1)}, k_0^{(3,2)},$
$k_2^{(0,0)}, k_0^{(0,1)}, k_1^{(0,2)},$
$R[47{:}32]$

$k^{(3,3)}$: $k_0^{(3,0)}, k_1^{(3,1)}, k_2^{(3,2)},$
$k_1^{(0,0)}, k_2^{(0,1)}, k_0^{(0,2)},$
$k_2^{(1,0)}, k_0^{(1,1)}, k_1^{(1,2)},$
$R[63{:}48]$

(b)

Figure 3: (a) The AES key schedule. We mark the input bytes of SubWord with colors, and hatch the key state bytes which are later combined with a specific input byte.
(b) The assignment of guards for the S-boxes of the key schedule.

no fresh randomness is required because only linear operations are performed, and we can use the 64 bits of fresh randomness produced in that cycle for refreshing the key schedule. We distribute the 64 bits over the four S-boxes, such that we add 16 distinct bits per S-box. By that, we can keep the refreshing of plaintext and key completely independent of each other, which is also important for probing security across multiple rounds, as discussed in Section 6.

5 Architecture

Masked AES hardware implementations either follow a *serial* or a *parallel* design paradigm. *Serial* AES designs instantiate the S-box once, which is fed with a new state or key byte every clock cycle. Most existing masked AES designs in literature focus on serial designs, including [DSM22, GMK16, Sug19, MPL+11, BGN+14, BGN+15, ADN+22], which is suitable for low-area, low-power purposes, but less for high throughput or low latency [UMHA16]. *Super box-serial* designs instantiate four S-boxes that are fed with a new super box every clock cycle and therefore provide a higher performance at the cost of area. *Parallel* or *round-based* AES designs instantiate the S-box 20 times, 16 inside SubBytes and 4 inside KeyExpand, which enables even higher performance at the cost of area. Our design follows a parallel architecture, as we use the AES implementation of the OpenTitan project as a basis. OpenTitan includes a first-order masked AES with a fully-parallel data path in order to achieve higher performance, but also because parallel architectures increase the noise in a system, which makes SCA harder [low23].

We give a sketch of our design in Figure 4. It takes 50+1 cycles to encrypt a block of 16 plaintext bytes. One cycle in the beginning is needed because the key schedule is started 1 cycle earlier than the processing of the plaintext in our design, such that the round key used in AddRoundKey for a specific round always comes from the key state registers. The linear map of our S-box design is now computed in the fifth stage of a round, which means the state registers of our implementation do not store the plain AES state but the state in the normal basis. We connect a Trivium RNG [Can06] to our design in order to further analyze the area overhead caused by utilizing multiple RNGs. We choose Trivium only as

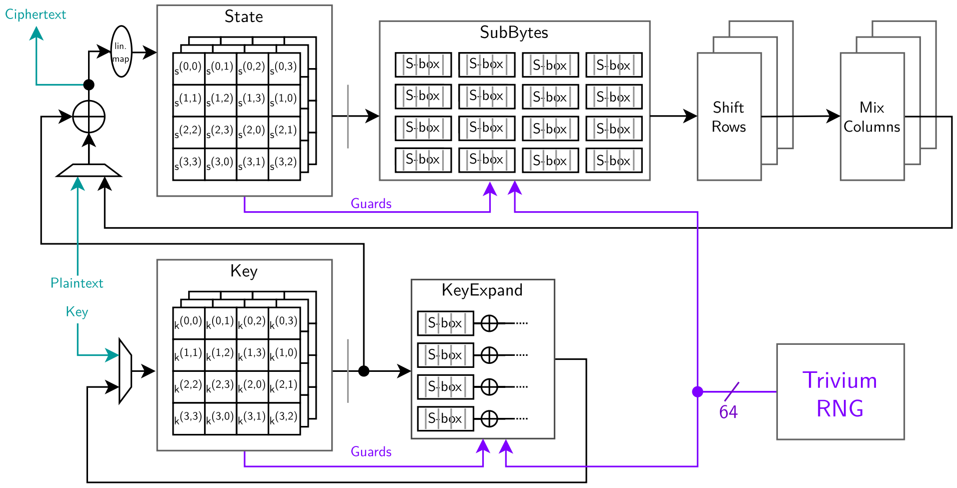

Ciphertext
lin. map
State
s(0,0) s(0,1) s(0,2) s(0,3)
s(1,1) s(1,2) s(1,3) s(1,0)
s(2,2) s(2,3) s(2,0) s(2,1)
s(3,3) s(3,0) s(3,1) s(3,2)
SubBytes
S-box
Shift Rows
Mix Columns
Guards
Plaintext
Key
Key
k(0,0) k(0,1) k(0,2) k(0,3)
k(1,1) k(1,2) k(1,3) k(1,0)
k(2,2) k(2,3) k(2,0) k(2,1)
k(3,3) k(3,0) k(3,1) k(3,2)
KeyExpand
S-box
Guards
64
Trivium RNG

Figure 4: Architecture of our second-order AES implementation. Pipeline stages are sketched with gray lines, inputs and outputs are marked in turquoise, and terms used for refreshing the S-box multipliers (guards and fresh randomness) are printed in purple.

an example that can, in practice, be replaced by any other RNG producing randomness at a sufficient quality. Our Trivium implementation provides 64 bits of fresh randomness per clock cycle. The randomness produced in the first four cycles of a round is consumed by the plaintext encryption (256 bits), and the randomness produced in the fifth cycle is consumed by the key schedule (64 bits). Our design requires 320 bits of fresh randomness per round, or 3 200 bits for 10 rounds.

5.1 Implementation and Comparison

We implement our design and obtain area measures using Cadence Genus Synthesis Solution 19.11-s087_1 for synthesis. All data is collected for a UMC 64 nm process and is expressed in 2-input NAND gate equivalents. The area of one NAND gate is $1.44\,\mu\text{m}^2$. In Table 3a, we give details about the area consumption of our AES design, which is in total 102 kGE. Two-thirds of the total area is attributed to the S-box instances for the plaintext/data, followed by the S-box instances for the key schedule. Since, to the best of our knowledge, our design is currently the only second-order parallel AES design, any direct comparison on cipher-level to related work is not possible. [ADN+22] provide a first-order parallel AES design with a 5-cycle S-box requiring 102.4 kGE, which is about the same as our second-order design. However, the comparison is not fair because the design does not use any online randomness at all, and the gate libraries as well as design compilers do not match.

On S-box level, we compare our design to related work in literature, as shown in Table 3b. However, it must be noted that these implementations use different CMOS libraries and design compilers, and therefore, the comparison only serves as a rough point of reference. Our optimized S-box design requires 4.3 kGE, which is slightly less (-0.1 kGE) than the fixed version of [GMK16], in which we include the fixed DOM-*dep* multipliers. Compared to the original versions of [GMK16], the area consumption of our design has not changed significantly. [SBB+22] and [NGPM22] propose S-box designs with a much lower latency than ours (1 cycle) but also with a higher area consumption. Gross et al. [GIB18] construct another DOM-S-box design focused on low-latency (2 cycles) without dual-rail logic, which however has a higher overhead in area and randomness than our

Table 3: Evaluation and comparison of our design in terms of area (* including control logic for COTG)

Module	Area	
	[%]	*[kGE]*
DOM-AES with COTG		
Data SubBytes	62%	63.7
Key SubWord	15%	15.7
MixColumns	3%	2.8
Control logic, state registers, etc	20%	19.8
Total AES	100%	**102**

(a)

2nd-order AES S-box	Area [kGE]	Latency [cycles]	Rand. [bits]	CMOS library
[CBR+15]	7.8	6	126	NanGate 45nm
[CRB+16]	3.8	5	162	NanGate 45nm
[GIB18]	57.1	2	4446	UMC 90nm
[NGPM22]	14.8	1	51	UMC 65nm
[SBB+22]	11.4	1	108	N/A (40nm)
[GMK16]	5.3	8	54	UMC 180nm
[GMK16] (insecure)	5.7	5	84	UMC 180nm
[GMK16] (fixed)	4.4	5	104	UMC 65nm
This work	**4.3**	**5**	**78**	UMC 65nm

(b)

Module	Area	
	[%]	*[kGE]*
DOM-AES with COTG, 1 Trivium instance		
AES*	87%	102
Trivium instance	5%	5.2
Outer control logic	8%	9.4
Total	100%	**116.6**
DOM-AES without COTG, 7.5 Trivium instances		
AES	68%	96.9
Trivium instances	25%	35.7
Outer control logic	7%	10.4
Total	100%	**142.1**
DOM-AES with fixed DOM-*dep*, 10 Trivium instances		
AES	65%	115.1
Trivium instances	30%	51.7
Outer control logic	5%	9.4
Total	100%	**176.2**

(c)

design. The five-cycle S-box proposed by [CRB+16] has a slightly lower area than our design but requires more than twice as much randomness.

In Table 3c, we compare our design with COTG to two versions of the design without COTG, connected to multiple instances of the Trivium RNG. This comparison highlights how important the reduction of randomness in a masked design is to achieve area efficiency. We evaluate our DOM-AES design using COTG, to which we connect a single Trivium instance, providing 64 bits of fresh randomness per clock cycle. The whole design requires 116.6 kGE, and the RNG makes 5% of the total area. We compare this to a version of our design where we do not use COTG but exclusively use fresh randomness for refreshing in the S-boxes, which consequently requires 7.5 Trivium instances. The total design area is 142.1 kGE, thus, represents an overhead of 22%. In a third scenario, we analyze the area consumption of the original DOM-AES design using our fixed DOM-*dep* multipliers. Here, 10 Trivium instances necessary, which consume 30% of the total design area, which is 176.2 kGE and represents an overhead of about 50% compared to our design using COTG. The area of the AES core has an overhead of 13% by using the DOM-*dep* multipliers instead of the smaller DOM-*indep* multipliers. Note that our AES design provides plenty of further possibilities for optimization, which would eventually reduce the area even more, including the elimination of the extensively used control logic for COTG. Additionally, instead of placing multiple Trivium instances, the Trivium state update function can further be unrolled to save area, as described in [CMM+23].

5.2 Application to other use-cases

Despite our decision to follow a parallel (round-based) design concept, the proposed concept for COTG can easily be carried over to serial and super box-serial architectures. The choice of guards stays the same; only the distribution of the randomness supplied by the RNG slightly changes. In a parallel design, all four super boxes are computationally in the same pipeline stage p in a specific cycle, and the 64 bits of fresh randomness are sent to that stage. In a super box-serial design, super box 0 would be in stage p, but super box 1 would be in stage $p-1$. Hence, one can send the 64 bits of fresh randomness to stage p for super box 0 and to stage $p-1$ for super box 1. Similar considerations are possible for a serial design, although an RNG supplying less than 64 bits would be sufficient.

While we focus on the second-order case, the proposed techniques can theoretically also be applied to higher-order ($d > 2$) DOM-protected AES implementations. To do so, one needs first to replace the DOM-*dep* multipliers in the S-box with DOM-*indep* multipliers, which requires adding even more fresh randomness per DOM-*indep* multiplier. Next, independent state bytes need to be identified, which can be used as guards in each S-box stage, similar to what is done in this work. We expect that this analysis, which is not trivial and becomes harder the higher the masking order, needs to be done individually for every order, while some knowledge, e.g., about the general dependency of state bytes, can be re-used from the second-order case.

The applicability of the concept to other ciphers, potentially protected by techniques other than DOM, highly depends on the concrete construction and requires a more in-depth individual analysis. For example, we expect that a similar technique can be applied to Ascon [DEMS21], and obtaining a COTG-based concept might be even less complex since DOM-masked Ascon implementations are available without using DOM-*dep* multipliers [GM17].

6 Security Evaluation

In this section, we elaborate on the security of our second-order DOM-AES implementation using COTG. First, we provide a formal security analysis of the design for which we use the formal verification tool Coco [GHP+21]. Second, we provide a practical security analysis by porting the circuit to an FPGA and showing that no leakage could be detected using TVLA with up to 100 million traces.

6.1 Formal verification setup

In this work, we use Coco [GHP+21] for formally verifying our design in the time-constrained probing model. The original purpose of Coco is to verify masked software implementations directly on the CPU netlist by incorporating control signals originating from the software execution. Given that Coco operates on gate-level netlists, it can also be used directly to verify masked hardware circuits with control logic, as demonstrated in [HB21]. To apply Coco, our design is first synthesized with Yosys [Wol16] to obtain such a gate-level netlist. We simulate the design to obtain values for control signals generated by the state machine in our design for the verification. Additionally, labels are assigned to the circuit inputs in order to indicate their purpose (share of a sensitive variable, fresh randomness, or unimportant/control signal). We further add some small modifications to Coco for our needs. For example, the original version of Coco constructs one SAT equation per sensitive bit in the circuit and then uses the incremental CaDiCaL SAT solver [BFFH20] to solve the equations in a sequential order. More precisely, the solver first checks the equation of the first sensitive bit and then uses the learned clauses for the remaining ones. Incremental SAT solving however comes with a certain overhead, e.g., for storing the learned clauses, and we found out that for our second-order hardware designs, the amount of re-usable learned clauses is so small that incremental solving does not pay off. Therefore, we use a parallel solver that solves all SAT equations individually but at the same time in parallel. We therefore adapt the Coco backend such that it uses the Kissat [BFFH20] solver. All experiments are executed on a machine with 88 CPU cores with 500 GB of RAM, such that approximately one CPU core is available per SAT formula.

6.2 Formal security of the design

In order to evaluate the security of our design, we follow a multi-step approach. First, we formally verify the second-order security of the S-box, treating the 78 input bits for refreshing the multipliers as fresh randomness first. Second, we take a look at the security of the design for one round on super box-level, including the usage of guards for refreshing, and formally verify it for both the key schedule and plaintext using Coco. Finally, we comment on the situation for the later rounds.

Formal verification of the S-box. As a first step, we formally verify with Coco that our proposed fix for the second-order DOM-*dep* multipliers is secure. For that, we create a $GF(2^2)$ and a $GF(2^4)$ DOM-*dep* multiplier implementation in System Verilog and verify the security in the time-constrained probing model for both implementations, which takes a few seconds. We then focus on the S-box construction proposed in Section 3.3, which does however not use the fixed DOM-*dep* multipliers to save randomness, which we verify for six cycles. We mark the three input shares (eight bits each) as sensitive values and the 78 bits of randomness for refreshing, which we all mark as uniformly random. Coco confirms the second-order security of our S-box implementation in the time-constrained probing model after running for approximately 1.5 days.

Formal verification of COTG for SubWord of key schedule. In order to formally verify one round of the key schedule using COTG, we label the three shares of the complete 128-bit key state as sensitive variables. During the computation of SubWord, these will be used as guards for refreshing. Additionally, we mark the 64 bits of fresh randomness required by the key schedule in Stage 4 of the S-boxes. With Coco, we can confirm the probing security of the construction computing four S-boxes in parallel over one round in 2 days and 18 h. This involves solving one SAT formula per unshared key bit, i.e., 128 SAT formulas in parallel. Not every SAT formula needs the same amount of time to solve, for example, the formulas of key bits that are not processed by SubWord are solved very quickly (in 2 s), while it takes up to the indicated 2 days and 18 h to check the security of key bits processed by the S-box.

One of the goals when constructing our design was to keep the refreshing terms used in the key schedule and plaintext isolated from each other to allow for easier security analysis. That is, no randomness or guards for refreshing are used in both the key schedule and the processing of the plaintext, and the only meeting point is AddRoundKey. Processing of the plaintext does not require fresh randomness in Stage 5 where the linear operations are done, but still, the RNG produces 64 bits of fresh randomness in that cycle, which we use for refreshing the key state after SubWords, impeding to probe key bytes in two different rounds.

Formal verification of COTG for SubBytes of plaintext. Compared to verification of the key schedule, verification of the COTG-based concept for the plaintext is much harder due to more complex dependencies between the state bytes. First, 16 S-boxes are computed in parallel instead of only four, and the guards used for these S-boxes are at the same time sent through their own S-box, where other guards are used. Second, we are using a combination of guards and fresh randomness for refreshing the multipliers connected by the $\oplus$ operation. Due to these two aspects, verifying a complete round for the complete 128-bit state becomes computationally infeasible.

Therefore, we constrain the verification to super boxes 0 and the first byte of super box 1 ($s^{(0,1)}$), i.e., we mark the whole 128-bit state of the AES as sensitive but disable the S-box computation for ($s^{(1,2)}, s^{(2,3)}, s^{(3,0)}$) and the bytes of the super boxes 2 and 3. This should not affect the verification of super box 0 since, in the first three stages, every super box uses guards only from the same or neighbor super box. Using this setup, we verify the construction for the first three stages, including the resharing phase of Stage 4. In Section 4.2, we discuss that inputs to Stage 4 are independent of each other, which allows to start the verification after Stage 3, assuming independent input shares. We verify the design beginning with the integration phase in Stage 3 until the end of Stage 5, including MixColumns, which is completed successfully.

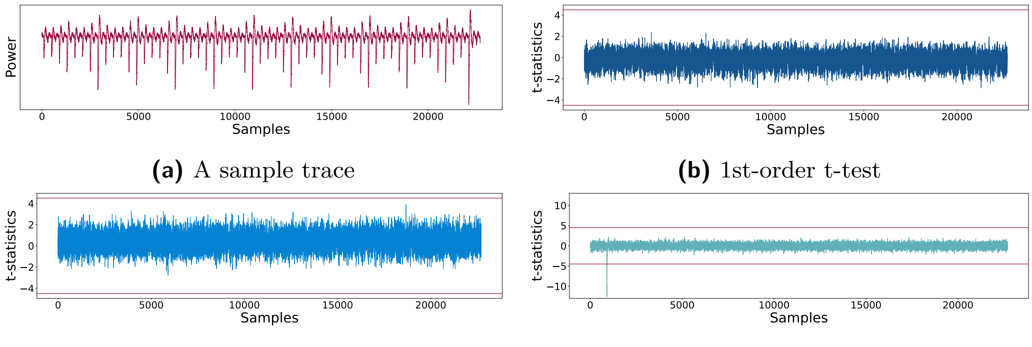
An attempt to verify a complete round at once was not conclusive, as the verification has been running for 55 days, and no leak has been found yet, but the security for all bits could not be confirmed either. The formula for the 88 bits not sent through S-boxes, which are only used as guards, could be solved within seconds, for further five bits we could confirm probing security after 37, 40, 41, 47, and 48 days respectively, but the confirmation for the remaining bits is still open.

Security across several rounds. As described above, our COTG-based design is considered to be probing secure for one round. Although we do not make any security claim beyond one round, our practical evaluations indicate that multiple rounds of our implementation are also secure due to the refreshing performed at two points in the design at the end of every round. First, we add 64 bits of fresh randomness before MixColumns by performing column-wise resharing. Second, AddRoundKey refreshes the complete 128-bit state of the cipher with state-independent key material. The key is completely independent of the state because of the strict separation of guards and fresh randomness for the key schedule and plaintext. However, after two rounds, the key shares and the state cannot be considered completely independent anymore because of the AES key schedule. More concretely, the key bytes are initially completely independent of each other. After executing one round of the key schedule, every key byte will at least depend on one other key byte, the guards used in the S-box, and some randomness. Even though this might lead to a small bias, our practical evaluations using TVLA confirm that this bias is not observable nor exploitable in practice.

6.3 Experimental Verification

In the last section, we discuss the outcome of the formal analysis, which indicates that our design is also second-order secure in the presence of glitches. Since formal verification is limited to less than one round of the design, we show practical evidence for the proposed statements for multiple rounds by porting the design to an FPGA in this section.

Evaluation setup. We perform practical evaluations using a first-, second- and third-order t-test on the NewAE CW305 Artix-7 FPGA evaluation board connected to a PicoScope 6404C at 625 Ms/s sampling rate (1.6 ns sampling interval). The hardware design operates at a clock frequency of 1.5625 MHz, which was chosen as a fraction of the sampling rate. To reduce the noise level, we synchronize the clocks between the FPGA and the oscilloscope and apply a preprocessing step to provide the equal alignment of traces. We implement our complete AES design, including the Trivium RNG as shown in Figure 4, along with some outer control logic used to send and receive data via the USB interface. The implementation receives three shares for the 128-bit plaintext, three shares of the 128-bit key, and a key-IV-pair to initialize the Trivium RNG. The Trivium RNG is initialized once in the beginning and produces 64 bits of fresh randomness per cycle during the encryption. In order to show whether or not a masked implementation exhibits first-order leakage, we follow the standard method and perform Welch's t-test following the guidelines of Goodwill et al. [GJJR11]. The basic idea of the test is to create a random and a fixed set of measurements, one representing the power consumption of the design when processing a random input and one when processing a fixed (constant) input. In order to determine if there are statistically significant differences in the mean power consumption of the two trace sets, one can compute Welch's t-score. The null hypothesis is that both trace sets have equal means, which can be rejected with a confidence greater than 99.999% if the t-score exceeds ±4.5. This implies that the trace sets can be distinguished from each other.



(a) A sample trace

(b) 1st-order t-test

(c) 2nd-order t-test

(d) 3rd-order t-test

Figure 5: Experimental analysis of our masked AES using 100 million traces.

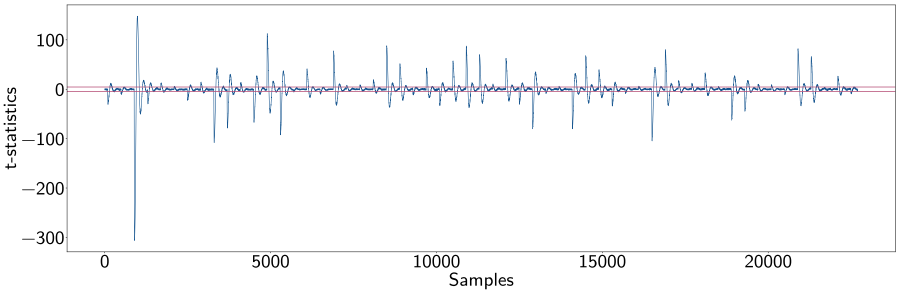


Figure 6: 1st-order t-test with RNG off and no initial sharing (two shares of plaintext and two shares of key are zeros) using 100 000 traces.

A first-order univariate t-test investigates distinguishably on the basis of the mean (first statistical moment) of the trace sets, a second-order univariate t-test uses the variance (second statistical moment) and a third-order univariate t-test uses the third statistical moment.

Discussion. To conduct a first-order, second-order, and third-order t-test, we choose a constant key, for which we generate a new valid sharing for every trace. For the fixed trace set, we set the input plaintext to zero and generate a new valid sharing for every trace of the fixed set. For the random set, we choose all three plaintext shares randomly for every trace. The fixed and random sets are recorded in an interleaved manner, and the RNG is enabled during our measurements. We measured the complete AES encryption, i.e., 10 rounds, as shown in a sample power trace in Figure 5a. The results of the first-order and second-order t-test are given in Figure 5b and Figure 5c. We did not observe evidence for first- or second-order leakage with up to 100 million traces, as the t-score never crosses the ±4.5 threshold. As shown in Figure 5d, we recorded third-order leakage as expected. The t-score exceeded the ±4.5 threshold during the initial AddRoundKey, where the overall noise level is expected to be very low. Since the key schedule starts one cycle before the processing of the plaintext, during the initial AddRoundKey, the processing of the data has not yet started, and the SubWord of the key schedule is only computing the linear mapping. No significant other computations are performed, leading to a low noise level.

To verify the soundness of our setup and to demonstrate that our countermeasure is effective, we show the t-test results of the design without supplying fresh randomness in Figure 6. This means we disable the RNG and the initial sharing of plaintext and key, i.e., two shares of the plaintext and two shares of the key are all zeros. As expected, after 100 000 traces, the design clearly showed first-order leakage.

7 Conclusion

In this work, we presented a second-order masked hardware design of the AES with an improved latency-randomness tradeoff. The resulting round-based (parallel) DOM-masked AES design works with three shares, has a latency of 5 cycles per round, and requires 3 200 random bits per encryption, which can smoothly be delivered by an RNG producing 64 bits of fresh randomness per cycle. The core of our AES design is a masked 5-cycle S-box which requires 78 bits of fresh randomness. We show how randomness can be reused across S-box instances using the COTG technique. We give formal security proofs, conduct an empirical evaluation using TVLA on an FPGA, and compare the implementation cost in terms of area consumption.

Acknowledgments

This work was supported by the FWF SFB project SpyCoDe F8504, and the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We thank our shepherd and anonymous reviewers for their valuable comments in strengthening this work. The authors would like to thank Gaëtan Cassiers for helpful discussions.

References

[ADN+22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. Guarding the first order: The rise of AES maskings. In Ileana Buhan and Tobias Schneider, editors, *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*, volume 13820 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2022.

[ANR19] Victor Arribas, Svetla Nikova, and Vincent Rijmen. Guards in action: First-order SCA secure implementations of KETJE without additional randomness. *Microprocess. Microsystems*, 71, 2019.

[BBP+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 397–426. Springer, 2017.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[BDRS21] Tim Beyne, Siemen Dhooghe, Adrián Ranea, and Danilo Sijacic. A low-randomness second-order masked AES. In Riham AlTawy and Andreas Hülsing, editors, *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*, volume 13203 of *Lecture Notes in Computer Science*, pages 87–110. Springer, 2021.

[BDZ20] Tim Beyne, Siemen Dhooghe, and Zhenda Zhang. Cryptanalysis of masked ciphers: A not so random idea. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 817–850. Springer, 2020.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. A more efficient AES threshold implementation. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, volume 8469 of *Lecture Notes in Computer Science*, pages 267–284. Springer, 2014.

[BGN+15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-offs for threshold implementations illustrated on AES. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(7):1188–1200, 2015.

[Can05] David Canright. A very compact s-box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006.

[CBR+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES s-box. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2015.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CMM$^+$23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking - unrolled trivium to the rescue. *IACR Cryptol. ePrint Arch.*, page 1134, 2023.

[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 Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[Dae17] Joan Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2017.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon. submission as a finalist to the nist lightweight cryptostandardization process, 2ß21. https://csrc.nist.gov/Projects/lightweight-cryptography/finalists. Retrieved on July 12th, 2023.

[DSM22] Siemen Dhooghe, Aein Rezaei Shahmirzadi, and Amir Moradi. Second-order low-randomness d + 1 hardware sharing of the AES. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 815–828. ACM, 2022.

[GHP$^+$21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[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 side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[GM17] Hannes Groß and Stefan Mangard. Reconciling d+1 masking in hardware and software. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017,* *Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 115–136. Springer, 2017.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GPM23] Barbara Gigerl, Robert Primas, and Stefan Mangard. Formal verification of arithmetic masking in hardware and software. *ACNS 2023*, 2023.

[HB21] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 1–10. IEEE, 2021.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[JPS18] Bernhard Jungk, Richard Petri, and Marc Stöttinger. Efficient side-channel protections of ARX ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):627–653, 2018.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 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 Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[low19] lowRISC contributors. Open titan, 2019. https://opentitan.org/. Retrieved on March 23th, 2023.

[low23] lowRISC contributors. Open titan - aes - theory of operation, 2023. https://opentitan.org/book/hw/ip/aes/doc/theory_of_operation.html#theory-of-operation. Retrieved on April 12th, 2023.

[MMM21] Nicolai Müller, Thorben Moos, and Amir Moradi. Low-latency hardware masking of PRINCE. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, volume 12910 of *Lecture Notes in Computer Science*, pages 148–167. Springer, 2021.

[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.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

[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 Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, 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 Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, 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.

[Nat01] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, 2001.

[NGPM22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. Riding the waves towards generic single-cycle masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):693–717, 2022.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[O'D14] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES s-box. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA,* *USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[SBB$^+$22] Mateus Simoes, Lilian Bossuet, Nicolas Bruneau, Vincent Grosso, Patrick Haddad, and Thomas Sarno. Self-timed masking: Implementing masked s-boxes without registers. In Ileana Buhan and Tobias Schneider, editors, *Smart Card Research and Advanced Applications - 21st International Conference, CARDIS 2022, Birmingham, UK, November 7-9, 2022, Revised Selected Papers*, volume 13820 of *Lecture Notes in Computer Science*, pages 146–164. Springer, 2022.

[SBHM20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-latency hardware masking with application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):300–326, 2020.

[SBM21] Aein Rezaei Shahmirzadi, Dusan Bozilov, and Amir Moradi. New first-order secure AES performance records. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):304–327, 2021.

[SD17] Niels Samwel and Joan Daemen. DPA on hardware implementations of ascon and keyak. In *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*, pages 415–424. ACM, 2017.

[SM21] 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, 2021.

[SP06] Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[Sug19] Takeshi Sugawara. 3-share threshold implementation of AES s-box without fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):123–145, 2019.

[UMHA16] Rei Ueno, Sumio Morioka, Naofumi Homma, and Takafumi Aoki. A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths - toward efficient cbc-mode implementation. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 538–558. Springer, 2016.

[WM18] Felix Wegener and Amir Moradi. A first-order SCA resistant AES without fresh randomness. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 245–262. Springer, 2018.

[Wol16] Claire Wolf. Yosys open synthesis suite, 2016. `http://www.clifford.at/yosys/`. Retrieved on February 2nd, 2021.