IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 2, pp. 304–327. DOI:10.46586/tches.v2021.i2.304-327

New First-Order Secure AES Performance Records

Aein Rezaei Shahmirzadi[1], Dušan Božilov[2,3] and Amir Moradi[1]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
firstname.lastname@rub.de
[2] NXP Semiconductors, Leuven, Belgium
[3] imec-COSIC, KU Leuven, Leuven, Belgium
dusan.bozilov@nxp.com,dusan.bozilov@esat.kuleuven.be

Abstract. Being based on a sound theoretical basis, masking schemes are commonly applied to protect cryptographic implementations against Side-Channel Analysis (SCA) attacks. Constructing SCA-protected AES, as the most widely deployed block cipher, has been naturally the focus of several research projects, with a direct application in industry. The majority of SCA-secure AES implementations introduced to the community opted for low area and latency overheads considering Application-Specific Integrated Circuit (ASIC) platforms. Albeit a few, those which particularly targeted Field Programmable Gate Arrays (FPGAs) as the implementation platform yield either a low throughput or a not-highly secure design.
In this work, we fill this gap by introducing first-order glitch-extended probing secure masked AES implementations highly optimized for FPGAs, which support both encryption and decryption. Compared to the state of the art, our designs efficiently map the critical non-linear parts of the masked S-box into the built-in Block RAMs (BRAMs).
The most performant variant of our constructions accomplishes five first-order secure AES encryptions/decryptions simultaneously in 50 clock cycles. Compared to the equivalent state-of-the-art designs, this leads to at least 70% reduction in utilization of FPGA resources (slices) at the cost of occupying BRAMs. Last but not least, we provide a wide range of such secure and efficient implementations supporting a large set of applications, ranging from low-area to high-throughput.

Keywords: Side-Channel Analysis · Masking · FPGA · Threshold Implementation · AES

1 Introduction

Cryptographic implementations in embedded devices are prone to Side-Channel Analysis (SCA) attacks, where the adversary tries to extract the secret by monitoring physical parameters of the cryptographic device like power consumption or electromagnetic radiation. After the introduction of the seminal work by Kocher et al. [KJJ99], an extensive body of literature has been dedicated to SCA attacks targeting both software and hardware implementations. Examples include Correlation Power Analysis (CPA) attack [BCO04], Mutual Information Analysis (MIA) [GBTP08], Moments-Correlating DPA (MC-DPA) [MS16], and Differential Deep Learning Analysis (DDLA) [Tim19]. This naturally necessitates employing proper countermeasures to achieve physical security in addition to analytical security of the underlying cryptographic primitives. Identifying the source of leakage and how to mitigate them have been deeply discussed by a great number of researchers in public literature. Among various known protection mechanisms, due to their sound mathematical

Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-02-23

basis, masking countermeasures are considered as the most convincing approach to provide security against SCA attacks. In a masking scheme, sensitive intermediate values of the cipher are randomized during the execution of the cryptographic operation. In this context, Boolean masking is known as the most popular scheme, in which the key-dependent variables are split into several shares, drawn randomly from a uniform distribution. For the sake of correctness, the sum of the shares should obviously yield the unshared value.

One of the first attempts to construct a masked circuit was presented in [Tri03], where a first-order secure AND gate is introduced. Moreover, Ishai et al. [ISW03] proposed a general algorithm for realizing the masked AND operation at the desired security order. These constructions appear consistent with software implementations, where operations are performed in a sequential manner. In contrast, these implementations exhibit exploitable leakages in hardware due to the lack of atomic gates, i.e., the gates which issue the output at a certain time independent of the given input. This leads to a phenomenon in hardware circuits known as glitches turning theoretically-secure schemes into practically-vulnerable designs [MPO05, MME10]. After many trial and error, the study in [NRR06] addressed several further questions on how to mitigate the effect of glitches and provided an implementation strategy with comprehensible properties, which – if fulfilled – makes the circuit secure even in the presence of glitches. This methodology, called Threshold Implementation (TI), was extended to higher orders in [BGN$^+$14b], with some limitations as addressed in [Rep15].

Evaluation of masked implementation is another challenge, which the community dealt with. The most common approach is the probing security model, introduced in [ISW03], where the number of wires that the attacker can simultaneously observe defines the order of the conduced attack. Consequently, the security order of a circuit is defined as the maximum number of probes that the adversary puts on the circuit without revealing any information about the secrets. While this abstract model is suitable for software implementations, it suffers from certain weaknesses when dealing with hardware implementations. To cope with this issue, the authors of [FGP$^+$18] have introduced the robust probing security model, which extends the original model to cover the physical characteristics of the circuit, including glitches and coupling. Obviously, by placing a probe on a circuit in a glitch-extended probing model, the adversary gains more information compared to the original probing model. In short, to cover the effect of glitches, any probe on a circuit is extended to all gates and input signals which are involved in the computation of the probed signal.

Following such evaluation strategies, several secure masking methodologies have been developed and introduced in order to realize the masked implementation of different cryptographic ciphers. Unsurprisingly, the lion's share of such designs focus on AES as the target algorithm [MPL$^+$11, BGN$^+$14a, BGN$^+$15, GMK16, CRB$^+$16, UHA17, Sug19]. A closer look at the literature on masked AES implementations reveals that most of the proposed methods take either Application-Specific Integrated Circuit (ASIC) or microcontrollers (software implementations) into account as the implementation platform. In contrast, a few works focus on Field Programmable Gate Array (FPGA) designs. Undoubtedly, all ASIC-based schemes can be employed on FPGA as well, but the FPGA resources (e.g., dedicated built-in hardware modules) are not necessarily utilized in an efficient manner.

As an FPGA-specific design, at the cost of extremely high latency, the authors of [WMM20] have presented a tiny masked AES for Xilinx FPGAs. Albeit efficiently utilizing FPGA resources, the design is absolutely not a choice when high throughput is desired, which – to the best of our knowledge – is among the main motivations behind employing an FPGA in various applications. Further, an FPGA-dedicated masked round-based implementation of AES has been presented in [GM11]. Although being a resource-efficient implementation in Xilinx FPGAs, it either reuses the same masks for multiple encryptions to maintain a reasonable throughput or turns into a high-latency design when no mask should be reused. The former might become vulnerable to certain

SCA attacks, while the latter cannot be considered as a highly-performant design.

One of the applications of high-throughput AES (in order of Gigabits per second) is in high-speed networks. To answer this demand, many AES IP cores are available in the market. For example, "Helion" [Hel] offers different purchasable AES IP cores for various FPGAs, ranging from tiny and low-area to large and fast cores. They provide a fast AES encryption with up to 1.8 Gb/s throughput in Spartan-6. We also should highlight that most of the secure protocols in the context of network security rely on AES and many companies employ FPGAs to support a high data rate. For instance, "Atmedia" [Atm] offers FPGA-based "Atmedi Ethernet Encryptor". As a matter of fact, none of these cases provides SCA-secure designs.

1.1 Our Contributions

In this work, we introduce a technique allowing us to realize a 2-share masked version of an 8-to-1 cubic function with an optimally-minimum number of component functions, i.e., small functions generating the output shares. Then, by decomposing the AES S-box and its inverse into two cubic permutations and applying our developed technique, we construct an FPGA-specific 2-share first-order secure AES cipher supporting both encryption and decryption with a low area footprint. Our design's overhead lies in occupying Block RAMs (BRAMs), which are building blocks existing in almost every Xilinx FPGA available in the market. We would like to highlight that our design is the first which 1) is optimized for FPGAs, 2) offers round-based implementation for high-throughput applications, 3) demands a little fresh randomness per cycle, 4) does not reuse the masks, and 5) is glitch-extended probing secure. Compared to the state of the art, we demonstrate that our only-encryption design reduces the number of utilized LookUp Tables (LUTs) and registers at least 70% and 85%, respectively while maintaining the latency. We further point out interesting trade-offs between different overheads like area, latency, and fresh randomness. Further, we verified the security of our designs using the recently-introduced verification tool SILVER [KSM20] under the glitch-extended probing model and provide the result of FPGA-based SCA evaluation experiments for the sake of completeness. All designs and HDL codes, including S-box, its inverse, round-based full cipher, and byte-serial implementations are provided on GitHub.

2 Preliminaries

In this section, we introduce our notations and then provide primarily knowledge about the fundamentals of masking. We also review requirements of masking in hardware and various state-of-the-art techniques used to realize secure hardware implementations. Our explanations also cover a basic introduction to FPGAs and their building blocks.

2.1 Notations and Definitions

We use lower-case and upper-case italic fonts to denote binary variables $x \in \mathbb{F}_2$ and vectors $X \in \mathbb{F}_2^{n>1}$, respectively. We represent matrices with upper-case italic bold $\boldsymbol{X}$, coordinate Boolean functions with lower-case italic sans-serif $\mathsf{f}(.)$, vectorial Boolean functions with upper-case italic sans-serif $\mathsf{F}(.)$, and sets with calligraphic font $\mathcal{F}$. Further, we show i-th share of a variable with subscripts x_i, and j-th bit of a binary vector X with x^j. Similarly, $\mathsf{f}^j(.)$ stands for a coordinate function generating j-th output bit of $\mathsf{F}(.)$.

Boolean masking is the most popular masking scheme in the literature in which the secret x is split into at least $s+1$ shares $(x_0, \ldots, x_s)$ to provide s-order security against SCA attacks. For the sake of correctness, the sum of the shares must be equal to the original non-shared value, i.e., $x = \bigoplus_{\forall i} x_i$. A possible way to form the *uniform sharing* of x is to draw $(x_0, \ldots, x_{s-1})$ from a uniform distribution and derive the last share x_s as $x \oplus \bigoplus_{\forall i<s} x_i$. When secret data X is presented in a shared form $(X_0, \ldots, X_s)$, the corresponding functions of the cipher should also be adjusted to the underlying sharing fashion. The masked realization of a linear Boolean function $Y = \mathsf{L}(X)$ can be trivially obtained by applying the same function on each set of shares individually $Y_i = \mathsf{L}(X_i)$. The crux of the matter, however, is how to achieve a masked realization of a given non-linear function $\mathsf{F}(.)$. It becomes particularly complicated for target functions with a large algebraic degree as well as for high desired security orders.

2.2 Threshold Implementations

Beyond the assumed-secure masked hardware implementations [Tri03, OMPR05] whose insecurity has been shown [MPO05, MME10], TI [NRR06] is the first technique which could show immunity in presence of glitches. Being based on Boolean masking, TI defines three below given properties, which should be fulfilled to realize a first-order secure hardware implementation. Let $Y = \mathsf{F}(X)$ be the target function with arbitrary input and output sizes. For the sake of simplicity, suppose that input and output should be presented by $s+1$ shares, i.e., the masked realization of $\mathsf{F}(.)$ receives $X_0, \ldots, X_s$ and provides $Y_0, \ldots, Y_s$.

- *Correctness* guarantees that the masked function operates correctly for all possible shared inputs. More precisely, $\forall X_i,\ \mathsf{F}(\bigoplus_{\forall i} X_i) = \bigoplus_{\forall i} Y_i$. Note that this property does not deal with the security of the design.
- *Non-completeness* guarantees that the calculation of any output share does not reveal any information about the secret even in the presence of glitches. For example, $\mathsf{F}(.)$ can be split into a set of so-called component functions $\mathsf{F}_i(.)$, each of which provides an output share Y_i by processing a set of input shares where the i-th share is missing, i.e., $(X_0, \ldots, X_{i-1}, X_{i+1}, \ldots, X_s)$. Hence, every output share is independent of at least one input share, and SCA leakage of $\mathsf{F}_i(.)$ cannot reveal any information about X.
- *Uniformity* implies that – given any uniformly-shared input X – the shared output of the function should not be distinguishable from Y being uniformly shared. Otherwise, the subsequent masked function receives non-uniform shared input, violating the fundamental assumption of Boolean masking.

As proven in [NRR06], non-completeness together with uniformity, leads to first-order secure designs. Classical TI introduced a strategy, called direct sharing, to realize a correct and non-complete masked variant of any Boolean function with algebraic degree t with at least $td+1$ input shares, where d stands for the desired security order. However, obtaining uniform output sharing is challenging as no solid methodology, except *remasking* by means of fresh randomness, is known. It seems that even the uniform sharing of some functions without using fresh randomness does not exist. For example, a uniform TI of 2-input AND gate with three input shares [NRR06] has not been reported yet. The same holds for χ operation of Keccak with three input shares [BDN+13].

Daemen addressed this issue and introduced *changing of the guards* [Dae17] in which achieving uniformity requires fresh randomness only at the initial clock cycles after the device powers up. This technique uses the unrelated part of the shared cipher state as a fresh mask to remask the non-uniform output of a shared non-linear function. The application of this technique has been shown on Keccak χ function [Dae17] and AES S-box [WM18, Sug19]. As a side note, this technique is just to achieve uniformity for a shared function that fulfills non-completeness. In case the shared function violates the non-completeness, changing of the guards cannot be beneficial.

Achieving a non-complete and uniform TI becomes even more challenging for a function with a high algebraic degree due to the high number of necessary input and output shares leading to more area and latency overhead. Thus, the solution that the community came up with is to try to decompose the target function into smaller functions – preferably quadratic – and mask them individually [BNN+15]. However, to avoid the propagation of glitches, this solution forces to add registers between each consecutive masked function, usually leading to a higher latency.

2.3 Probing Security

To evaluate the security of masking schemes, Ishai et al. [ISW03] introduced a simple and abstract model, called probing security model. In this model, the number of probes, that the attacker can put on the given circuit with which to observe the intermediate values, reflects the order of the attack. Hence, in a d-th order secure design, any combination of at most d probes should not reveal any information about the secret. Ignoring micro-architectural leakage, this model perfectly works for software implementations where instructions are executed sequentially, and each operation can be seen as an atomic gate, where the output changes only once per operation. The same, however, does not hold for hardware implementations due to the well-known phenomenon in CMOS circuits called glitches.

Glitches are unwanted signal transitions in the output of a combinatorial circuit, mostly because of unbalanced path delays at the gate's inputs. It is extremely difficult to model the exact effect of the glitches in a given circuit since it is influenced by some factors on which we, as designers, have no control, e.g., process variation. Consequently, there has been a considerable body of work on how to adjust the probing security model in the presence of glitches, where the most convincing one seems to be *glitch-extended* probing model introduced in [FGP+18]. In this model, by probing the output of a gate, the attacker has information about all intermediate values of the combinational circuit involved in the calculation of the probed gate.

Due to its simplicity, this abstract model is widely employed to assess the security of designs and improve them in terms of area overhead, latency, and randomness complexity [BGR18, SM20]. It is also extensively adopted for formal verification [KSM20, BBC+19] due to the fact that this abstraction allows reducing the complexity of security proofs for implementation of masking schemes. We should highlight that in this work, we mainly focus on first-order secure hardware implementations. Therefore, we consider the glitch-extended probing model in our theoretical analyses and use SILVER [KSM20] to verify the security of our constructions.

2.4 Masking with $d + 1$ Shares

While classical TI defines the number of input shares as $td + 1$, it has been shown that less number of shares can be employed to achieve the same security in hardware implementations [RBN+15, GMK16]. More precisely, $d + 1$ input shares can be used to construct a secure masked hardware implementation (under the glitch-extended probing model), regardless of the algebraic degree of the target function. In this method, the given function is split into two parts with a register stage in between to avoid propagation of glitches. For example, the authors of [GMK16] provided a secure masked variant of a 2-input AND gate $x = f(a, b) = ab$ with $d + 1$ shares. An example for $d = 1$ is given in Equation (1), where a_0, a_1, b_0, b_1 denote the input shares, x_0 and x_1 the output shares, and r a single-bit fresh mask.

$$\begin{array}{llll} f_0(a_0,b_0) & = a_0b_0 & \to x'_0 & \\ f_1(a_0,b_1,r) & = a_0b_1 + r & \to x'_1 & x'_0 + x'_1 = x_0 \\ \hline f_2(a_1,b_0,r) & = a_1b_0 + r & \to x'_2 & x'_2 + x'_3 = x_1 \\ f_3(a_1,b_1) & = a_1b_1 & \to x'_3 & \end{array}, \qquad (1)$$

$f_0(.)$ to $f_3(.)$ are known as *component functions*, which are indeed the shared form of $f(.)$. The result of every component function, i.e., x'_0 to x'_3 should be stored in registers and then fed into the *compression layer*, compressing them to $d+1$ output shares by means of XORs.

The authors of [RBN+15] showed that it is not always necessary to use fresh randomness to apply this method. For instance, the first-order security can be achieved for $x = f(a,b,c) = ab + c$ by following the Equation (1) and replacing r with c_0 and c_1 in f_1 and f_2, respectively. Further, it has been shown in [SM20] that the same concept can be used to achieve a uniform and first-order secure 2-input AND without any fresh randomness by replacing c with b in $x = \overline{a}b + c$.

2.5 Field Programmable Gate Array (FPGA)

An FPGA is an integrated circuit designed to be reprogrammable by a netlist as the result of synthesizing a design expressed in a Hardware Description Language (HDL), such as Verilog or VHDL. The reprogrammability of FPGAs distinguish them from ASICs manufactured to do a specific task. An FPGA consists of thousands of Configurable Logic Blocks (CLBs) that can be configured to perform complex combinatorial functions, and a system of programmable interconnects to route signals between CLBs. Most of the recent FPGAs host other features like Block Memories, Digital Signal Processing blocks, and dedicated high-speed serial communication facilities mostly left unused for cryptographic applications. In this work, we mainly focus on Xilinx FPGA families (from 6- and 7-Series), and hence give some related preliminary information, helping to follow the rest of the paper. Note that our designs are general and can be implemented (synthesized) on every FPGA. However, we opted to develop our implementations for Xilinx FPGA families currently available in the market.

Xilinx FPGAs Architecture

Each Xilinx FPGA from the 6- and 7-Series contains a matrix of CLBs, whose number depends on the device size and model. Each CLB consists of two slices, each of which contains four LUTs, eight flip-flops, and miscellaneous logic, regardless of their type. It offers dual-register 6-input LUTs (with one-bit output) that can also be used as two 5-input LUTs with common inputs. Synthesis tools utilize these logic blocks to implement the desired functions if not particularly configured to use certain internal hardware blocks.

Such FPGAs additionally provide other sorts of built-in system-level blocks, including 18 kb BRAM, where each can be used as two independent 9 kb memory blocks. Large amounts of data can be stored on these embedded block memories inside the FPGA, offering fast access to the data. Interestingly, the size of the address and data ports can be adjusted by the user based on their needs. For example, a 9 kb block (which is actually an 8 kb data memory and an additional 1 kb of space for parity) can be configured to have a 13-bit address and 1-bit data port or a 10-bit address and 8-bit data port.

Both read and write operations are fully synchronous, meaning that the input of a BRAM is always registered. An optional output register can be configured to reduce the latency of the circuit. BRAMs also feature True Dual-Port (TDP), implying that there exist two completely independent address and data ports, which can be simultaneously used to access the content of a BRAM.

3 AES S-box

In this section, after reviewing state-of-the-art techniques used to realize secure hardware implementation of the AES S-box, we introduce our methodology to achieve optimal output shares for cubic functions. Afterward, we shortly discuss the decomposition of the AES S-box and employ our methodology to achieve a first-order secure S-box as well as its inverse.

3.1 Related Works

An AES S-box consists of an inversion over $GF(2^8)$ and an affine transformation over $GF(2)$. The inversion is performed on a polynomial basis with the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. The corresponding hardware implementations with a polynomial basis lead to relatively large area and latency overheads. Canright addressed this issue and provided a tower-field representation of the inversion in $GF(2^8)$ [Can05]. He split the AES S-box into some smaller functions and changed the representation and calculation from a polynomial basis to a normal basis in each sub-field. This allowed him to design a compact and efficient inversion circuit over $GF(2^8)$. An interesting point of this technique is that the sub-functions' algebraic degree is at most three, which is the inversion over $GF(2^4)$. This initiated a vast number of research on how to take advantage of the tower-field approach and make a masked AES S-box with limited overheads. This is mainly due to the fact that – as stated in Section 2 – the number of input shares depends on the algebraic degree of the target functions in classical $td + 1$ TIs.

One of the first such works is [MPL+11] where, in order to mask the entire AES S-box with three input shares, all operations in $GF(2^4)$ are also split into $GF(2^2)$ sub-field using the same tower-field approach. This results in having multiplication in $GF(2^2)$ as the only non-linear (quadratic) operation for the entire inversion in $GF(2^8)$, allowing to use only three shares to realize the masked circuit. Consequently, the design has four-stage pipeline architecture for the inversion and an extra stage for the affine transformation combined with the normal-to-polynomial isomorphism. The design has been improved later in [BGN+14a] in terms of area, latency, and randomness complexity, and by giving some trade-offs between area and additional randomness [BGN+15]. In these works, the inversion in $GF(2^4)$ has been used without splitting into sub-field, allowing to remove one register stage. However, the authors had to adapt the number of shares as such an inversion is a cubic function.

Two more compact masked AES S-box designs using $d+1$ sharing have been introduced in [GMK16] based on the same tower-field approach, one with five and the other one with seven pipeline stages. The latter one has less randomness complexity while the former has less area overhead. Two other $d + 1$ masked designs have been presented in [CRB+16] and [UHA17] leading to less area overhead at the cost of more latency and more randomness complexity. Further, a methodology has been presented in [SM20], which allows realizing a 2-share masked variant of the inversion in $GF(2^8)$ without using any fresh randomness but with higher area demands. We should highlight that all the aforementioned designs are based on the tower-field approach.

The changing of the guards [Dae17] which relaxes the demands for fresh randomness, has also been applied on the masked AES S-box designs. The authors of [WM18] decomposed the AES S-box into two cubic functions and by making use of this technique, introduced a 4-share masked AES without fresh randomness. Furthermore, the same has been applied on a 3-share masked design presented in [Sug19] which follows the tower-field approach.

The former suffers from a high latency while the latter has considerable area overhead showing a trade-off between the randomness and latency/area.

It is noteworthy that all aforesaid designs are optimized for an ASIC platform, and all of them have been used in the serialized implementation of AES to show their performance. Indeed, none of them has been designed to efficiently utilize FPGA resources, e.g., slices, LUTs, and BRAMs. Instead, in [DMW18] (improved in [WMM20]) an FPGA-specific masked AES has been presented. Following the same concept as in [WM18], the authors decomposed the inversion in $GF(2^8)$ into two cubic functions and reduced the area footprint on FPGA by exploiting the rotational symmetry [RBF08]. To be able to take advantage of the rotational symmetry, the authors had to construct a bit-serial architecture. As a result, the latency of the design is extremely high and obviously inappropriate for applications with high-throughput requirements.

To the best of our knowledge, the only FPGA-specific masked implementation of the round-based AES with a reasonable latency has been presented in [GM11]. The underlying so-called Block Memory Scrambling (BMS) technique efficiently makes use of BRAMs available in Xilinx FPGAs. For the new randomly selected set of masks, the masked look-up tables, which are stored in BRAMs, are recalculated on the fly parallel to the encryption. Consequently, as long as the entire look-up tables are not recalculated (taking 512 clock cycles), the same masked look-up tables are re-used for encryption. In other words, not only all cipher rounds but also a couple of consecutive encryptions use the same masks, which potentially can increase the risk of SCA collision attacks. In order to mitigate this vulnerability, 512 clock cycles should be the minimum gap between consecutive encryptions, which again lowers down the throughput. Even in such a case, all cipher rounds still share the same masks.

3.2 Technique

In order to minimize the area of the implementation of the S-box (resp. S-box inverse), we need to minimize the number of component functions in the nonlinear layer. Based on [WM18], it is possible to decompose the AES S-box into cubic permutations. Hence, we are interested in finding a minimal number of component functions to realize a 2-share masked form of 8-to-1 cubic functions, i.e., an 8-bit coordinate function. To generalize and to consider the worst-case scenario, we suppose that the target coordinate function contains all cubic terms, namely $\binom{8}{3}$ cubic monomials. This way, the construction which we build here can be used for any 8-bit cubic coordinate function, even if it does not contain all cubic monomials.

The work presented in [WMM20] provides the masked form of such a function with 16 component functions, but with no guarantees of optimality, since the presented algorithm always produces $k \times (d+1)^t$ component functions, with k being an integer. Here, $d = 1$ and $t = 3$, i.e., 2 shares and a cubic coordinate function. In fact, the authors optimized k and not the number of component functions in general. Additionally, another methodology has been given in [BKN19], which is only suitable for sharing functions that have degree $t = n - 1$, where n is the number of input bits. As we have $t = 3$ and $n = 8$, this method obviously cannot be applied in our case. Consequently, following this approach leads to a realization with 128 component functions for an 8-to-1 function, which is far more than necessary for cubic functions.

In order to ease the notation, we utilize the same matrix/table representation of output sharing used in [WMM20] and [BKN19]. While the table notation does not uniquely determine the Algebraic Normal Form (ANF) of the sharing, it is sufficient to argue the correctness and non-completeness properties. In the table notation, the number of columns and rows represents the number of input bits and the number of component functions, respectively. The i-th row contains the sharing indices of input variables that the i-th component function receives, and the j-th column represents the sharing indices of the j-th input variable. A quick example of such a table is given in Equation (2), where function $ab + c$ is shared using 4 component functions, with the sharing table on the left.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{array}{l} f_0(a_0, b_0, c_0) = a_0 b_0 + c_0 \\ f_1(a_0, b_1, c_0) = a_0 b_1 \\ f_2(a_1, b_0, c_1) = a_1 b_0 \\ f_3(a_1, b_1, c_1) = a_1 b_1 + c_1 \end{array} \tag{2}$$

To verify the correctness of a sharing table for a given function, we need to make sure that all shared monomials in the ANF representation are present in the corresponding component functions an odd number of times. Equivalently, given all possible rows of an n-bit input function with $d + 1$ input shares, we need to find a subset of rows that jointly contains all shared monomials. Let us refer to the set of all possible rows as $\mathcal{R}$. Each row of $\mathcal{R}$ contains a number of shared monomials of degree n or less. For example, in an 8-bit input (a, b, c, d, e, f, g, h), row $(0, 1, 0, 0, 1, 1, 1, 0)$ implies that the corresponding component function receives $(a_0, b_1, c_0, d_0, e_1, f_1, g_1, h_0)$. Hence, the component function can have monomials like $a_0 b_1 d_0$, and $e_1 f_1 g_1 h_0$ among the others.

For a specific unshared ANF, we only need to observe its shared monomials. Since we can enumerate each shared monomial as an element $e \in \mathcal{E}$, we can naturally say that for every row $\forall S \in \mathcal{R}, S \subset \mathcal{E}$. The problem of finding minimal sharing now turns into the problem of finding a subset of $\mathcal{R}$ with a minimum size that contains all elements of $\mathcal{E}$.

This is a well-known discrete optimization problem referred to as Set Covering Problem (SCP), and we can utilize discrete optimization methods to solve it. Written more formally, given set $\mathcal{E}$ and a family of rows $\mathcal{R}$, we associate variable $x_S \in \{0, 1\}$ to S denoting if row S is chosen or not. Finding the correct minimal sharing can then be formulated as:

$$\text{minimize} \sum_{\forall S \in \mathcal{R}} x_S \tag{3}$$

$$\text{subject to } \ \forall e \in \mathcal{E}, \exists S \in \mathcal{R} \text{ s.t. } e \in S, \ x_S = 1 \tag{4}$$

Expression (3) is referred to as objective cost or objective function, while the constraints are given with inequalities in expression (4). In our concrete case, we are interested in 8-to-1 cubic functions and their 2-share masked realization. Therefore, there are 2^8 different rows in the sharing table, i.e., $|\mathcal{R}| = 256$, which is also the number of decision variables. A 2-share form of a generic cubic coordinate function with all cubic terms contains $2^3 \times \binom{8}{3} = 448$ cubic shared monomials. Note that we do not need to add elements associated with monomials of degree less than 3, because if we have a valid sharing for such a cubic coordinate function, we can systematically add quadratic and linear monomials into the component functions.

While 256 decision variables are too large for an exhaustive search, we can utilize discrete optimization method constraint programming [RBW06], by creating the model based on SCP formulation given with objective cost 3 and constraints 4 using freely available MiniZinc tool [NSB+07] with the Chuffed backend solver [CS14]. Constraint Programming (CP) is a discrete optimization method used to find solutions to satisfiability problems by exploring the search space while trying to satisfy all the given constraints. It can also be adapted to solve optimization problems such as set covering by adding the constraint that each new solution has better objective cost than the previous one, and then repeatedly tries to satisfy new objective cost constraints until the problem becomes infeasible.

Programming in MiniZinc is based on creating an optimization model with respect to the given constraints and objective cost function. To employ SCP, we can model the elements (shared monomials) from $\mathcal{E}$ and sets (component functions) with a cover matrix C, where each row represents one element to be covered. Entries in a single row indicate all sets covering that particular element. If all elements are not covered by the same number of sets, we need to pad to maximal possible length by adding entries at the end with a dummy value. In the case of 8-bit-1 cubic functions, we can enumerate all possible component functions, ranging from 0 to 255 while 256 indicates a dummy value that does not represent any output share. The result will be given as an array of output shares *out*, indicating a list of chosen indices for the input of corresponding component functions. To reduce the search space for the solver, we should enforce the ordering in *out* in such a way that the sharing indices are listed in increasing order. Furthermore, since MiniZinc cannot deal with unknown-size arrays, we must allocate enough space to ensure that the solution is always fitted. In our case, the total number of rows in the sharing table is 256. Note that the order of constraints does not matter in MiniZinc, since they are evaluated jointly. To reiterate, our model consists of the following steps:

1. Create element cover matrix C where each row represents a different element, and entries in each row indicate which sets cover that particular element. If padding with dummy values is needed, we should make sure that all rows have the same cardinality.

2. Allocate output array *out* with enough elements covering the worst-case scenario. This array will contain the chosen covering sets found by the solver.

3. Add a constraint that enforces the ordering in *out*, which helps break the symmetry.

4. Add a constraint that entries of an unused portion of *out* are equal to the dummy value.

5. Add a constraint that enforces to have at least one valid value in each row of *out* to ensure that all elements will be covered.

6. Add objective cost function that minimizes the number of non-dummy elements in *out*. This minimizes the number of sets used for covering, effectively solving our optimization problem.

The model is quite simple and is more or less a direct translation of the Expressions (3) and (4). We had to add symmetry breaking constraints. The first one ensures the ordering of the output, while the second one guarantees that unused indices in the *out* array are all filled with the dummy value. While adding the ordering constraint has no impact on the correctness of the model, it greatly reduces the solving time. After running the model, MiniZinc provides increasingly better quality solutions until the underlying solver determines that the optimal solution has been found or until a user-specified timeout is reached. While we only demonstrate the usefulness of the aforementioned technique on first-order $d + 1 = 2$ sharing of cubic 8-to-1 coordinate functions, it can also be applied in a more general manner for different algebraic degrees and security orders, given that the cover matrix C is correctly initialized, the total number of rows in the sharing table is correctly given, and the array *out* is allocated with enough entries to fit the solution. In fact, this model can be used to solve any set covering problem.

Depending on the problem at hand, CP solver might take quite a while to prove optimality, or in some cases, even to find a feasible solution. However, on a regular PC, for our instance of set covering problem used to find minimal sharing of cubic coordinate functions of 8 bits, the solver finds an optimal solution with 12 output shares within 3 seconds as given below.

$$\left(\begin{array}{cccccccc}0&0&0&0&1&1&0&0\\0&0&0&1&1&0&1&1\\0&0&1&0&0&0&0&1\\0&0&1&1&0&0&1&0\\0&1&0&1&0&1&0&1\\0&1&1&0&1&0&1&0\\1&0&0&0&0&0&1&0\\1&0&1&1&1&1&0&1\\1&1&0&0&1&1&1&1\\1&1&0&1&1&0&0&0\\1&1&1&0&0&1&0&0\\1&1&1&1&0&0&1&1\end{array}\right)\begin{array}{l}\mathsf{f}_0\ (a_0,b_0,c_0,d_0,e_1,f_1,g_0,h_0)\\\mathsf{f}_1\ (a_0,b_0,c_0,d_1,e_1,f_0,g_1,h_1)\\\mathsf{f}_2\ (a_0,b_0,c_1,d_0,e_0,f_0,g_0,h_1)\\\mathsf{f}_3\ (a_0,b_0,c_1,d_1,e_0,f_0,g_1,h_0)\\\mathsf{f}_4\ (a_0,b_1,c_0,d_1,e_0,f_1,g_0,h_1)\\\mathsf{f}_5\ (a_0,b_1,c_1,d_0,e_1,f_0,g_1,h_0)\\\mathsf{f}_6\ (a_1,b_0,c_0,d_0,e_0,f_0,g_1,h_0)\\\mathsf{f}_7\ (a_1,b_0,c_1,d_1,e_1,f_1,g_0,h_1)\\\mathsf{f}_8\ (a_1,b_1,c_0,d_0,e_1,f_1,g_1,h_1)\\\mathsf{f}_9\ (a_1,b_1,c_0,d_1,e_1,f_0,g_0,h_0)\\\mathsf{f}_{10}(a_1,b_1,c_1,d_0,e_0,f_1,g_0,h_0)\\\mathsf{f}_{11}(a_1,b_1,c_1,d_1,e_0,f_0,g_1,h_1)\end{array} \tag{5}$$

We would like to highlight that 12 component functions are not always the optimal solution for an 8-to-1 cubic function. If the ANF is simple, better sharings can certainly be found. For example, 8 component functions would be enough for $\mathsf{f} = abc + def + g + h$.

3.3 Our Masked AES S-box and its Inverse

Our goal is to represent the masked variant of the AES S-box using 2 shares while minimizing the latency (number of register stages) and limiting the demand for the fresh randomness. To this end, we followed the procedure given below.

An inversion X^{-1} over $GF(2^8)$ can also be represented as X^{254}. Since it has been shown that the decomposition of an inversion over $GF(2^q)$ into quadratic permutations when q is a multiple of 4 is not an option [NNR19], similar to [WM18] we focus on decomposition into cubic permutations. In other words,

$$X^{254} = (X^n)^m = (X^m)^n, \qquad \mathrm{HW}(n) = \mathrm{HW}(m) = 3,$$

with $\mathrm{HW}(\alpha)$ being Hamming weight, i.e., the number of 1s in the binary representation of α. The authors of [WM18] provided all such tuples (n, m) as

$$(13, 98), (26, 49), (52, 152), (104, 76), (208, 38), (161, 19), (67, 137), (134, 196),$$

for each of which (m, n) is also a valid tuple. We denote such cubic functions as $\mathsf{F}(X) = X^n$ and $\mathsf{H}(X) = X^m$. Showing the affine transformation of the S-box by $\mathsf{A}(.)$, which follows the inversion, the AES S-box and its inverse can be represented as

$$\begin{aligned}\mathsf{S}(X) =& \mathsf{A} \circ \mathsf{H} \circ \mathsf{F}(X) = \mathsf{G} \circ \mathsf{F}(X),\\ \mathsf{S}^{-1}(X) =& \mathsf{F} \circ \mathsf{H} \circ \mathsf{A}^{-1}(X) = \mathsf{F} \circ \mathsf{W}(X).\end{aligned}$$

We have observed that for all the above-listed tuples, the ANF of the vectorial Boolean function $X \mapsto X^m$ contains all possible cubic monomials. Therefore, considering the technique expressed in Section 3.2, none of the tuples has any advantage compared to the others. Hence, without any specific reason, we have taken the tuple $(n = 26, m = 49)$ for the decomposition. As given in Section 3.2, each coordinate function of $\mathsf{F}(.)$, $\mathsf{G}(.)$, and $\mathsf{W}(.)$ can be masked using 12 component functions, where the compression layer combines (XOR) the output of 6 of them to form one output share.

In the following, we focus on explaining how our technique can be applied on $\mathsf{F}(.)$, which is indeed the same procedure when targeting $\mathsf{G}(.)$ or $\mathsf{W}(.)$. Let $\mathsf{f}^j(.)$ be the j-th coordinate function of $\mathsf{F}(.)$, whose shared form is represented by a set of 12 component functions $\mathcal{F}^j = \left\{\mathsf{f}_i^j(.), 0 \le \forall i \le 11\right\}$. Each $\mathsf{f}_i^j(.)$ receives an 8-bit input X'_i formed by a mixture of input shares while maintaining non-completeness. As stated in Section 3.2 and based on Equation (5), we can exemplary write $X'_0 : \langle x_0^7, x_0^6, x_0^5, x_0^4, x_1^3, x_1^2, x_0^1, x_0^0 \rangle$. As a matter of fact, the same input X'_i is given to the i-th component function of all coordinate functions, i.e., $0 \leq \forall j \leq 7, f_i^j(X'_i)$. Hence, for each $i \in \{0, \ldots, 11\}$ these 8 component functions can be aggregated and form an 8-bit to 8-bit vectorial Boolean function $F'_i(.)$ receiving the aforementioned X'_i as input.

In short we have 12 functions $F'_{0 \leq i \leq 11}(.)$ as the sharing of $F(.)$, which fulfill both correctness and non-completeness. Next, we need to deal with two facts: glitch-extended probing security, and uniformity which should be satisfied by remasking. As stated, the sharing of each coordinate function is done by 12 component functions, while each 6 are combined in the compression layer to make one output share. Hence, trivially 5 fresh mask bits can be used to make the compression layer (of each coordinate function) glitch-extended probing secure (see Section 2.4 and [RBN+15]). This further yields a uniform output sharing of the underlying coordinate function. Therefore, 40 fresh mask bits are required for each shared $F(.)$. We follow the below-given procedure to decrease this to 8 fresh mask bits, which is indeed the minimum to uniformly refresh the output sharing of a 2-share vectorial Boolean function with 8-bit output, i.e., $F(.)$.

For simplicity, let us denote the output of a component function $f_i^j(.)$ by y_i^j, and represent the output of all component functions by the matrix $\boldsymbol{Y}$ as

$$\boldsymbol{Y} = \left(\begin{array}{cccc} y_0^7 & y_0^6 & \cdots & y_0^0 \\ y_1^7 & y_1^6 & \cdots & y_1^0 \\ \vdots & \vdots & \ddots & \vdots \\ y_5^7 & y_5^6 & \cdots & y_5^0 \\ \hline y_6^7 & y_6^6 & \cdots & y_6^0 \\ y_7^7 & y_7^6 & \cdots & y_7^0 \\ \vdots & \vdots & \ddots & \vdots \\ y_{11}^7 & y_{11}^6 & \cdots & y_{11}^0 \end{array}\right).$$

Without losing generality, we suppose that the output of the first six component functions of each coordinate function (i.e., $y_0^j, \ldots, y_5^j$) are combined in the compression layer, of course, after being stored in registers (see Section 2.4). Then, obviously, the next six outputs $y_6^j, \ldots, y_{11}^j$ are combined to make the other output share. We also denote an 8-bit fresh random mask by $\langle r^0, \ldots, r^7 \rangle$, and divide the remasking process into two steps. In the first step, the matrix $\boldsymbol{R}$ is formed as follows.

$$\boldsymbol{R} = \left(\begin{array}{c} \\ \boldsymbol{R}^{0,\ldots,5} \\ \\ \hline R^6 \\ R^7 \end{array}\right) = \left(\begin{array}{c} R^0 \\ R^1 \\ R^2 \\ R^3 \\ R^4 \\ R^5 \\ \hline R^6 \\ R^7 \end{array}\right) = \left(\begin{array}{cccccccc} r^0 & r^0 & r^0 & r^0 & r^0 & r^0 & r^0 & 0 \\ r^1 & r^1 & r^1 & r^1 & r^1 & r^1 & 0 & r^1 \\ r^2 & r^2 & r^2 & r^2 & r^2 & 0 & r^2 & r^2 \\ r^3 & r^3 & r^3 & r^3 & 0 & r^3 & r^3 & r^3 \\ r^4 & r^4 & r^4 & 0 & r^4 & r^4 & r^4 & r^4 \\ r^5 & r^5 & 0 & r^5 & r^5 & r^5 & r^5 & r^5 \\ \hline r^6 & 0 & r^6 & r^6 & r^6 & r^6 & r^6 & r^6 \\ 0 & r^7 & r^7 & r^7 & r^7 & r^7 & r^7 & r^7 \end{array}\right) \quad (6)$$

By taking the first 6 rows of $\boldsymbol{R}$, i.e., $\boldsymbol{R}^{0\ldots5}$, we make the remasking matrix $\boldsymbol{R}' = \begin{pmatrix} \boldsymbol{R}^{0,\ldots,5} \\ \boldsymbol{R}^{0,\ldots,5} \end{pmatrix}$, with which the output of component functions are remasked and stored in registers as $\boldsymbol{Y}' = \left[\boldsymbol{Y} \oplus \boldsymbol{R}'\right]$. This is actually enough to satisfy glitch-extended probing security of the compression layer since each column of the upper half of $\boldsymbol{Y}'$ are combined, whose at least

5 values are refreshed. In other words, placing a glitch-extended probe on the XORs of the compression layer does not lead to any set of probes involving both shares of an input bit.

However, this does not necessarily lead to a jointly uniform output sharing. For example, the last two output shares (two left most columns in $\boldsymbol{Y'}$, respectively $\boldsymbol{Y}$ and $\boldsymbol{R'}$) are remasked by the same value $r^0 \oplus r^1 \oplus r^2 \oplus r^3 \oplus r^4 \oplus r^5$, meaning no joint uniformity. Therefore, we need to add more fresh masks, which we decided to apply in the compression layer. The reason behind this decision is our implementation platform. As stated in Section 2.5, our goal is to fit as many as possible functions into the BRAMs, with restricted input/output size (address/data ports). As given, each $F'_i(.)$ is an 8-bit to 8-bit function, and its output is XORed with the corresponding R^j for remasking, with $j = i \mod 6$, i.e., $F'_0(.) \oplus R^0, \ldots, F'_5(.) \oplus R^5$ and $F'_6(.) \oplus R^0, \ldots, F'_{11}(.) \oplus R^5$. Since each R^j depends only on one fresh mask bit r^j, each $F'_i(.) \oplus R^{i \mod 6}$ can be seen as a 9-bit to 8-bit function perfectly fitting into the targeted BRAMs.

If we denote the rows of the matrix $\boldsymbol{Y'}$ by 8-bit vectors $Y'^j, 0 \leq j \leq 11$, applying the fresh masks R^6 and R^7 (defined in Equation (6)) in the compression layer, the two 8-bit output shares of $F(.)$ are generated as

$$Y_0 = R^6 \oplus R^7 \oplus \bigoplus_{0 \leq j \leq 5} Y'^j, \qquad Y_1 = R^6 \oplus R^7 \oplus \bigoplus_{6 \leq j \leq 11} Y'^j.$$

If we denote $\bigoplus\limits_{0 \leq j \leq 7} r^j$ by r', the 8-bit output shares of $F(.)$ are actually refreshed by

$$\Big\langle r' \oplus r^0,\ \ r' \oplus r^1,\ \ r' \oplus r^2,\ \ r' \oplus r^3,\ \ r' \oplus r^4,\ \ r' \oplus r^5,\ \ r' \oplus r^6,\ \ r' \oplus r^7 \Big\rangle$$

leading to a jointly uniform sharing as long as $\langle r^0, \ldots, r^7 \rangle$ are independent of each other. We should highlight that since in our scheme, some component functions of different coordinate functions receive the same fresh masks, as stated in [SM20], the output shares (Y_0, Y_1) should be stored in registers before being given to the subsequent functions. Otherwise, the design would not be necessarily glitch-extended probing secure.

The same technique can be used to make a 2-share masked variant of $G(.)$ and $W(.)$ as they are also 8-bit to 8-bit cubic functions. As a result, we can construct a 2-share masked implementation of the AES S-box and its inverse. The general structure of such a design is shown in Figure 1. Note that the "Indices Selector" module receives two input shares X_0 and X_1, and based on our technique explained in Section 3.2 provides $X'_{0 \leq i \leq 11}$ as the inputs for $F'_i(.)$ (resp. $G'_i(.)$).

In order to analyze our construction, we made use of the open-source verification tool SILVER [KSM20]. Since it just accepts ASIC net-lists, we implemented our masked S-box (Figure 1) by an HDL code and synthesized it by Synopsys Design Compiler with the NanGate 45 nm ASIC standard cell library, as dictated by SILVER. The net-list contains 4 690 gates and 8 825 signals. The verification tool running on a machine with 16 CPU cores and 128 GB of RAM required 42 minutes to report first-order security of our design under the glitch-extended probing model as well as the uniformity of its output sharing.

As an important fact to highlight, remasking of the decomposed functions (here $F(.)$ and $G(.)$) are conducted with two clock cycles distance. Since it is supposed that fresh mask bits $r^0, \ldots, r^7$ are updated at each clock cycle, we can connect the same 8-bit signal to the fresh mask input port of both decomposed functions. Considering a pipeline design processing consecutive inputs $X^0, \ldots, X^5$, the fresh masks used by $G(.)$ on X^0 are also used by $F(.)$ on X^2 and so on (i.e., with a distance of two pipeline stages). However, when X^2 reaches $G(.)$, i.e., after two clock cycles, the fresh masks are entirely updated. Hence, this does not affect the security of the design, and we are in consequence able to just use 8 fresh masks per clock cycle in our constructed masked AES S-box (resp. its inverse).

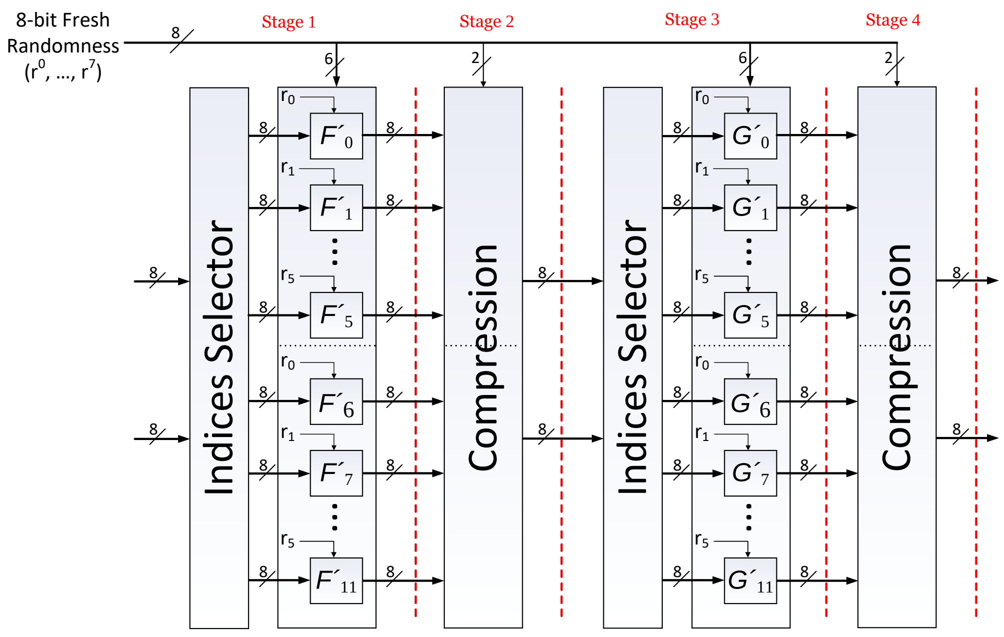


Figure 1: General structure of our 2-share masked AES S-box. Register stages are marked by red dashed lines.

4 Design

In this section, we demonstrate how our masked S-box and its inverse can efficiently fit into BRAMs as the FPGA building blocks. This achieves a secure and resource-efficient AES implementation in a round-based fully pipeline fashion supporting both encryption and decryption. We should highlight that, albeit focusing on Xilinx 6- and 7-Series FPGAs, our designs are general and can be adapted to different FPGAs.

4.1 Our FPGA-Specific Masked AES

As mentioned in Section 3.3, both AES S-box and its inverse can be decomposed into two cubic permutations. Each one can be uniformly shared by 12 combined component functions, each seen as a 9-bit to 8-bit function, including the corresponding fresh mask. The shared S-box and its inverse can be combined as depicted in Figure 2, where the "Indices Selector" block, which is just the wiring, provides 8 bits out of the 16-bit input shares for each component function. The "`E/D`" signal selects whether the S-box or its inverse should be performed. We should stress that the sharing indices are independent of the desired functionality, i.e., S-box or inverse. As shown by the graphics, the `E/D` signal is only connected to the component functions.

Each block highlighted with the blue dashed line (Figure 2) can be seen as a function with 10-bit input and 8-bit output, where the input is formed by the corresponding 8-bit mixture of input shares (provided by the "Indices Selector" for $F'_i(.)$, resp. $G'_i(.)$ or $W'_i(.)$), a single fresh mask bit, and `E/D` signal. Hence, each such block can be perfectly realized by a 9 kb BRAM, which can be configured as a memory block with 10-bit address and 8-bit data ports[1]. Due to the unknown internal architecture of BRAM, non-completeness should be fulfilled at the address port of each BRAM, which is indeed the case in our

[1]Indeed with 9-bit data port when the redundant part parity is used.

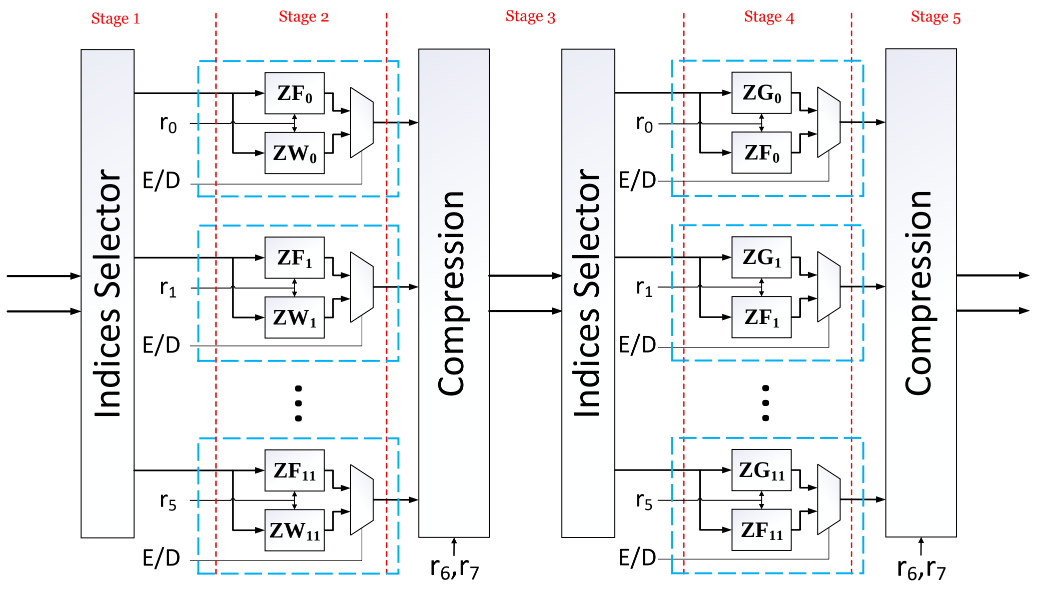


Figure 2: General structure of the combined masked AES S-box and its inverse. Red dotted lines indicate register stages, and the blue dashed lines the BRAMs.

design. We assigned the `E/D` signal to the Most Significant Bit (MSB) of the address port, and the rest is filled by the single-bit fresh mask and the 8-bit mixed input shares. As shown in Figure 1 and stated in Section 3.3, the output of the compression function should be stored in a register before being given to the next masked non-linear function. The same holds for the input of the component functions if it is provided by a linear function, otherwise violating the non-completeness. For example, it is the case in a round-based implementation, where the S-box input (in encryption) is provided by the `MixColumns` of the former round. Therefore, it is essential to place a register stage at the input of the S-box (resp. S-box inverse). Since the registers are transparent to the "Indices Selector" modules, we could efficiently use the BRAM's internal registers for this purpose. Hence, we configured the BRAMs to have the optional register at the output port as well (red dashed lines). This way, both input and output ports of a BRAM are stored in the internal registers, so there is no need to use any external registers, hence keeping the number of utilized slices low. As a result, the entire design shown in Figure 2 fits into 24 BRAMs.

Another beneficial feature of BRAMs is its True Dual-Port (TDP), shortly introduced in Section 2.5. In our case, it implies that we can perform two S-box (resp. S-box inverse) operations with two independent inputs (including independent fresh masks) at the same time using the aforementioned 24 BRAMs. Note that since each BRAM port receives two independent sharings associated with two independent bytes (S-box inputs), the non-completeness property still holds. This allows us to implement a round-based AES cipher supporting both encryption and decryption using 10 instances of such a construction. More precisely, 8 instances realize 16 S-boxes (or the inverse) for the data path, and 2 other instances implement 4 S-boxes required for the `KeyExpansion`, hence in total 240 BRAMs. As stated in Section 3.3, each S-box module (resp. its inverse) requires 8-bit fresh randomness per clock cycle, translating to 160 bits per clock cycle for the entire round-based encryption/decryption implementation.

Figure 3 shows a block diagram of our AES encryption/decryption design. Due to its round-based architecture, the data path for encryption is straightforward. Sequentially, `AddRoundKey`, `SubBytes`, `ShiftRows`, and `MixColumns` are performed, where the `MixColumns` is ignored in the last round by the dedicated multiplexer. However, we have

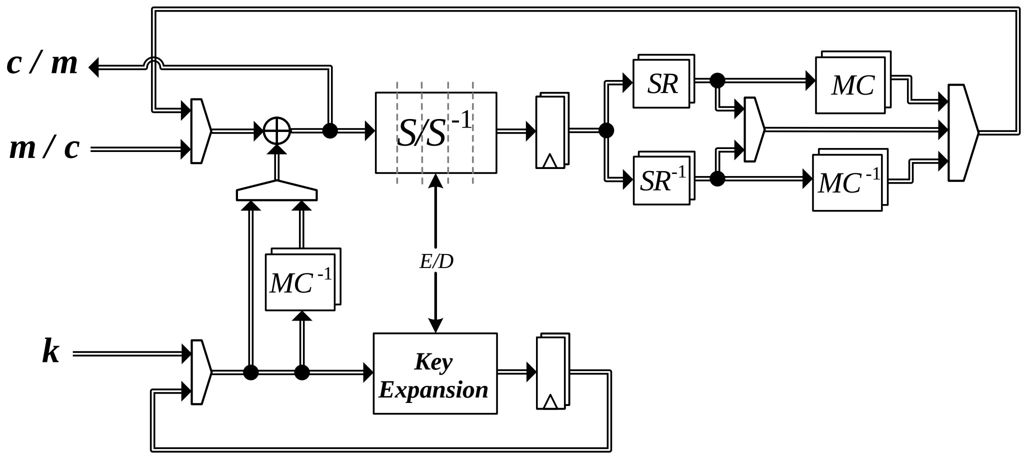


Figure 3: 2-share masked round-based AES-128 supporting encryption and decryption.

done some optimizations to realize decryption as well. We have added an extra instance of `InvMixColumns` before the `AddRoundKey`. During the first decryption round, the given key is regularly added to the ciphertext to be loaded into the `InvSubBytes`. Due to the linearity of `InvMixColumns`, we can rearrange the order of operations for subsequent decryption rounds, since $MC^{-1}(X \oplus K) = MC^{-1}(X) \oplus MC^{-1}(K)$. Therefore, by applying `InvMixColumns` before `AddRoundKey`, we require to pass the round keys through an additional `InvMixColumns` module. This allows us to efficiently use the resources essential for the encryption. Overall, each encryption/decryption round needs 5 clock cycles. Since the BRAM internal register stages form a pipeline, the design can process 5 independent consecutive plaintext/ciphertext and provide the corresponding ciphertext/plaintext after 50 clock cycles. Note that since the `KeyExpansion` is not pipelined, the key used for all 5 consecutive inputs should be the same. We refer to this construction as the "fully-pipeline" design. For only-encryption design, the blocks dedicated to decryption can be easily removed, e.g., `InvMixColumns`, `InvShiftRows`, and corresponding multiplexers. It is worth to mention that the only-encryption design makes use of the same number of BRAMs.

4.2 Comparison

In order to be able to truly compare our design with the state of the art, we have taken our round-based encryption architecture and replaced the S-box with that of [GMK16], [GMK16], [CRB+16], and [SM20]. Note that none of these works provided either a round-based architecture or its performance results. For most of them, we could use the source code of the implementations available on GitHub provided by the authors. We particularly had to reform the encryption data path to be able to host the S-box design of [SM20], as its output sharing is not jointly uniform, and it makes use of an adapted form of `MixColumns` to achieve uniform sharing. Nevertheless, the corresponding comparative performance results are shown in Table 1, clearly indicating low-resource utilization of our design at the cost of occupying BRAMs. Interestingly, the overhead to turn our only-encryption design to additionally cover decryption is limited to around 1.9 k LUTs. Note that in all such designs, we kept the hierarchy of implementation; otherwise, the essential non-completeness property might be violated. This, however, in FPGAs means that different modules cannot share a slice, although they use different registers and LUTs. Since state-of-the-art designs make use of several modules for the masked S-box, this naturally leads to higher overhead, while in our design all such functions are realized by BRAMs.

Further, we would like to highlight the number of required fresh randomness of our

Table 1: Comparison of high-performance implementations excluding PRNG (with key masking, mapped on Xilinx Spartan-6)

Design	Slice	Reg	LUT	BRAM	Lat.	Rand.	Clock	Thr.put
	[#]	[#]	[#]	[#] 9 kb	[cycles]	[bits]	[MHz]	[Gbit/s]
[GMK16]	3726	4127	7315	0	50	560	131	1.676
[GMK16]	3296	4685	6757	0	80	360	137	1.753
[CRB+16]	3414	3407	7105	0	60	1080	134	1.715
[SM20]	4722	5328	7783	0	70	0	131	1.676
[GM11][a]	?	1566	2888	16	512[b]	51.2[c]	100[d]	0.025
[*new*] full E/D	1708	527	3778	240	50	160	102	1.305
[*new*] full E	1033	527	2068	240	50	160	116	1.484
[*new*] half E	529	651	1152	96	70	64	116	0.636
[*new*] quar. E	297	588	663	48	90	32	119	0.338

[a] Based on Xilinx Virtex-II Pro, and without key masking.
[b] Without mask reuse.
[c] Averaged over 10 clock cycles. 512 fresh mask bits are required at the start of each scrambling phase.
[d] Nominal frequency, it is not mentioned in [GM11].

design, which is the lowest compared to the other designs with the same latency, i.e., 50 clock cycles per encryption. On the downside, the maximum clock frequency of our design (resp. maximum throughput) is slightly less than the other designs. This is due to the delay (maximum clock frequency) of BRAM, which is more than that of LUTs.

4.3 PRNG

One fact which has been commonly ignored in the relevant state of the art is the area required to generate the fresh masks. Here, we try to include this factor in our comparison. Round-based implementations need a high number of fresh masks per clock cycle, which is hard to generate by means of True Random Number Generators (TRNGs), due to their naturally low throughput, high cost, and dependency to physical properties [YRG+18]. Alternatively, Pseudo-Random Number Generators (PRNGs) can be used, which are seeded by a TRNG at power-up. Therefore, we followed the concept applied in [DMW18] to efficiently construct a PRNG using the FPGA building blocks. More precisely, for each required fresh mask bit (updated every clock cycle), we implemented a Linear Feedback Shift Register (LFSR) with feedback polynomial $x^{31} + x^{28} + 1$. Based on the instruction given in [DMW18], each LFSR can be implemented by only 3 LUTs in Xilinx FPGAs: two LUTs as Shift-Register LUT and another one for the feedback function. In Table 2 we report the same comparative performance figures including the PRNG. As shown, the amount of demands for fresh randomness plays a crucial role in the area overhead, although – to the best of our knowledge – such LFSRs are the most FPGA-efficient constructions. We should emphasize that it is the first time such a comparison, including PRNGs, is presented.

4.4 Area-Latency-Randomness Trade-off

As stated, our fully-pipeline design needs 240 instances of 9 kb BRAMs. Such a number of BRAMs are available on mid-size Xilinx Spartan-6, Spartan-7, and Artix-7 FPGAs, which are known as the cost-optimized solutions. For example, XC6SLX75, XC7S50, and XC7A50T are the smallest devices of the aforementioned families, with enough BRAMs for our fully-pipeline design [Xil]. In order to adapt our design to the smaller FPGAs, we provided two other design variants so-called “half-pipeline” and “quarter-pipeline”. The

Table 2: Comparison of high-performance implementations including PRNG (with key masking, mapped on Xilinx Spartan-6)

Design	**Slice** [#]	**Reg** [#]	**LUT** [#]	**BRAM** [#] 9 kb
[GMK16]	4989	4127	8995	0
[GMK16]	4072	4685	7837	0
[CRB$^+$16]	5580	3407	10345	0
[SM20]	4722	5328	7783	0
[*new*] fully-pipeline E/D	2054	527	4258	240
[*new*] fully-pipeline E	1335	527	2548	240
[*new*] half-pipeline E	943	651	1344	96
[*new*] quarter-pipeline E	368	588	759	48

first one realizes a two-column serial architecture, i.e., 8 masked S-boxes are processed at the same time, hence reducing the BRAM utilization to 96 instances. However, due to the registers involved in the BRAMs, we could still make a pipeline design, which processes 3 plaintexts simultaneously in 70 clock cycles. Note that those 8 masked S-boxes are shared with the `KeyExpansion` as well. Namely, in every cipher round, the first 2 clock cycles the S-boxes receive the state associated to the first plaintext, one more clock cycle that of the `KeyExpansion`, and the next 4 clock cycles the states of the second and third plaintexts, i.e., 7 clock cycles per cipher round.

In our quarter-pipeline design, which forms a column-serial architecture, 4 masked S-boxes (resp. 48 BRAMs) are instantiated. It also realizes a pipeline and performs two encryptions at the same time in 90 clock cycles. Table 1 and Table 2 include the performance figures of these two designs as well. Clearly, the resource utilization decreases at the cost of lower throughput.

These designs can fit into smaller Xilinx FPGAs, including the smallest Artix-7 (XC7A12T). However, the smallest Spartan-6 (XC6SLX4) and smallest Spartan-7 (XC7S6) have 24 and 20 instances of 9 kb BRAMs. This motivated us to build another area-optimized encryption design making use of a single masked S-box, i.e., byte serial. Based on the explanations given in Section 4.1, each masked S-box and its inverse need 24 BRAMs. At first glance, it seems impossible to fit a masked S-box of our design into such smallest FPGAs. However, looking at Figure 1 it can be seen that each $F'_i(.)$ and $G'_i(.)$ can fit into a single BRAM, each of which makes use of a separate port (see our explanation about True Dual-Port (TDP) in Section 2.5). As a result, we made an only-encryption version of our masked S-box construction using 12 BRAMs. This allows us to fit our design into all Xilinx FPGAs, even XC7S6, the smallest Spartan-7. Accordingly, we provided a performance table listing the resource utilization and throughput of our design compared to the state of the art in Table 3. Of course, the smallest design is the one presented in [WMM20], which has 30 times lower throughput compared to almost every other known implementation. Compared to the other designs with comparable throughput, our design outperforms the others with respect to resource utilization (except for BRAM). We would like to highlight that the smallest Spartan-6 (XC6SLX4) has only 600 slices. Hence, this device may not be able to easily host other designs, particularly the one presented in [SM20]. However, our design only occupies 41% of its available slices. As stated, the HDL sources of all our designs are available on GitHub.

Table 3: Comparison of area-optimized (byte-serial) implementations including PRNG (with key masking, mapped on Xilinx Spartan-6)

Design	**Slice** [#]	**Reg** [#]	**LUT** [#]	**BRAM** [#] 9 kb	**Lat.** [cycles]	**Rand.** [bits]	**Clock** [MHz]	**Thr.put** [Mbit/s]
[GMK16]	433	726	894	0	216	28	110	65.1
[GMK16]	351	754	792	0	246	18	134	69.7
[CRB+16]	418	690	905	0	276	54	133	61.6
[SM20]	631	866	1021	0	246	0	136	70.7
[DMW18]	254	124	347	0	6852	18	103	1.9
[WMM20]	140	92	248	0	6852	6	120	2.2
[*new*] E	203	529	476	12	216	8	138	81.7

5 Analysis

As stated earlier, we verified the security of our constructions using SILVER [KSM20] under the glitch-extended probing model. However, since SILVER is not able to handle designs making use of FPGA-dedicated building blocks, e.g., BRAM, we provided an ASIC-based representation of our masked S-box and performed the evaluations by SILVER on the resulting netlist. As a side note, no verification tool (including SILVER) can analyze the implementation of a full cipher. Instead, such tools can only handle a part of the circuit, e.g., an S-box or small gadgets. Hence, we evaluated our constructions by experiments conducted on an FPGA evaluation board and collecting power consumption traces.

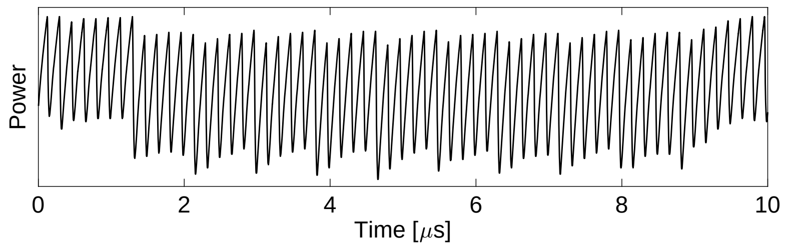
5.1 Setup

We implemented our fully-pipeline round-based masked AES encryption on the target Spartan-6 FPGA of the SAKURA-G board [SAK]. By means of a digital oscilloscope monitoring the voltage drop over a 1 Ω resistor placed in the Vdd path of the target Spartan-6 FPGA, we collected power consumption traces. The voltage drop is amplified by a dedicated on-board AC amplifier and is captured at a sampling rate of 500 MS/s, while the FPGA is clocked at the frequency of 6 MHz.

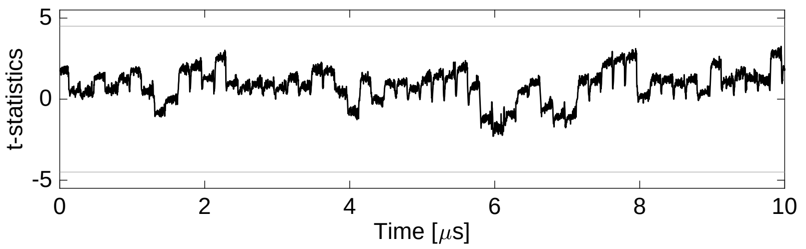
5.2 Evaluation and Results

Welch's t-test is a well-known methodology in the area of SCA, commonly used both as a distinguisher (for instance, in classical DPA attack [KJJ99]) and as an evaluation method. In the Test Vector Leakage Assessment (TVLA) methodology [GJJR11], the target device receives fixed or random plaintext in a non-deterministic fashion, and the power consumption traces are captured for those two groups. Eventually, Welch's t-test can be applied independently on every single point of the traces of two groups, i.e., a univariate analysis. This method, so-called the fixed-versus-random t-test, has been widely used in the literature to evaluate the security of masked implementations [SM15, BGN+14b, SMMG15, LMW14]. In fact, the t-test assesses the detectability of leakage of the design implemented on the target device without performing the actual attack. Although it gives a confidence level of the existence of leakage in the target device, it provides no information about the hardness/easiness of the attack exploiting the leakage to recover the secret, if there is a detectable leakage.

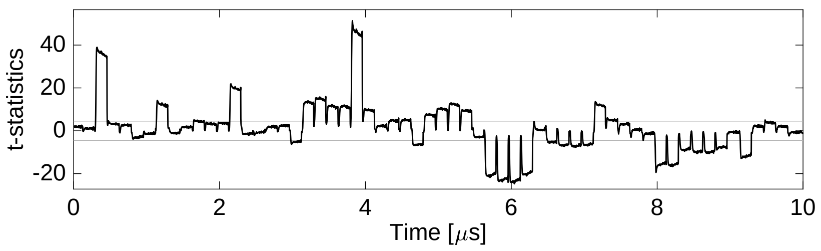
We collected 100 million power traces with randomly-selected fixed or random plaintexts, where a sample trace can be seen in Figure 4(a), covering the entire encryption process. Subsequently, we performed the univariate fixed-versus-random t-test at first and second orders, and obtained the curves of the t-statistics, shown in Figure 4(b) and Figure 4(c),



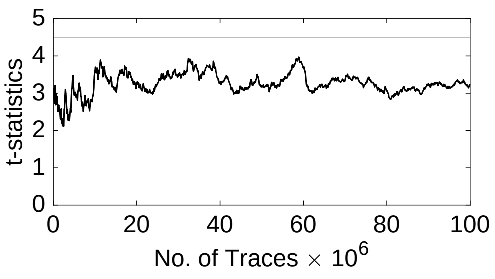
(a) A sample trace indicating pipeline stages and cipher rounds



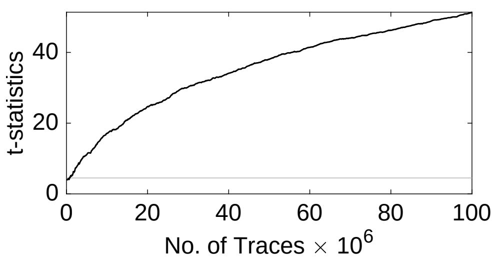
(b) First-order leakage assessment



(c) Second-order leakage assessment



(d) First-order over no. of traces



(e) Second-order over no. of traces

Figure 4: First- and second-order leakage assessment results of our fully-pipeline AES encryption module, using 100 million power traces.

respectively. As expected, the design exhibits second-order leakage, while the first-order leakage is not observed. Figure 4(d) and Figure 4(e) depict the maximum *t*-value over the number of collected traces. It can be seen that the first-order *t*-value does not pass the threshold by increasing the number of evaluated traces, while this is not the case for the second-order scenario. Since the analysis results of our other designs (half-pipeline and quarter-pipeline) are almost the same, we omit reporting the corresponding figures.

6 Conclusions

In this work, we have presented a methodology that allows us to form a first-order secure realization of an 8-to-1 cubic function using two shares with at most 12 component functions. We have also demonstrated how we can take advantage of this technique and FPGA building blocks to construct a resource-efficient implementation of the AES cipher supporting both encryption and decryption, considering the effect of glitches in hardware platforms. We realized different cipher implementation variants and compared their trade-offs in terms of area footprint, latency, and randomness complexity. These designs are suitable for any sort of FPGAs, ranging from low-resource ones to those optimized for the highest performance. We compared our designs with the state of the art and showed that the amount of required fresh randomness plays a non-negligible role in the area overhead. In short, our FPGA-specific constructions outperform the implementations known through public literature, particularly including the area required for the generation of fresh masks. Our designs are indeed the only glitch-extended probing secure implementations that use FPGA BRAMs.

Acknowledgments

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

References

[Atm] Atmedia. https://www.atmedia.de/en/index-2.html. Accessed on 2020-12-18.

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults. In *ESORICS 2019*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

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

[BDN+13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013.

[BGN+14a] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. A More Efficient AES Threshold Implementation. In *AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 267–284. Springer, 2014.

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

[BGN+15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-Offs for Threshold Implementations Illustrated on AES. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(7):1188–1200, 2015.

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

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

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

[Can05] David Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

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

[CS14] Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. https://github.com/chuffed/chuffed.

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

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

[FGP$^+$18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

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

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

[GM11] Tim Güneysu and Amir Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2011.

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

[Hel] Heliontech. https://www.heliontech.com/. Accessed on 2020-12-18.

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

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

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

[LMW14] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. Gate-Level Masking under a Path-Based Leakage Metric. In *CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 580–597. Springer, 2014.

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

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

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

[MS16] Amir Moradi and François-Xavier Standaert. Moments-Correlating DPA. In *Theory of Implementation Security - TIS@CCS 2016*, pages 5–15. ACM, 2016.

[NNR19] Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Decomposition of permutations in a finite field. *Cryptogr. Commun.*, 11(3):379–384, 2019.

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

[NSB$^+$07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-Box. In *FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.

[RBF08] Vincent Rijmen, Paulo S. L. M. Barreto, and Décio L. Gazzoni Filho. Rotation symmetry in algebraically generated cryptographic substitution tables. *Inf. Process. Lett.*, 106(6):246–250, 2008.

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

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., USA, 2006.

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

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

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

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

[SMMG15] Pascal Sasdrich, Oliver Mischke, Amir Moradi, and Tim Güneysu. Side-Channel Protection by Randomizing Look-Up Tables on Reconfigurable Hardware - Pitfalls of Memory Primitives. In *COSADE 2015*, volume 9064 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2015.

[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.

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

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

[UHA17] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward More Efficient DPA-Resistant AES Hardware Architecture Based on Threshold Implementation. In *COSADE 2017*, Lecture Notes in Computer Science, pages 50–64. Springer, 2017.

[WM18] Felix Wegener and Amir Moradi. A First-Order SCA Resistant AES Without Fresh Randomness. In *COSADE 2018*, volume 10815 of *Lecture Notes in Computer Science*, pages 245–262. Springer, 2018.

[WMM20] Felix Wegener, Lauren De Meyer, and Amir Moradi. Spin Me Right Round Rotational Symmetry for FPGA-Specific AES: Extended Version. *J. Cryptol.*, 33(3):1114–1155, 2020.

[Xil] Xilinx. Cost-Optimized Portfolio Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/cost-optimized-product-selection-guide.pdf. Accessed on 2020-10-16.

[YRG$^+$18] Bohan Yang, Vladimir Rozic, Milos Grujic, Nele Mentens, and Ingrid Verbauwhede. ES-TRNG: A High-throughput, Low-area True Random Number Generator based on Edge Sampling. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):267–292, 2018.