Spin Me Right Round
Rotational Symmetry for FPGA-specific AES

- Extended Version -

Felix Wegener[1], Lauren De Meyer[2] and Amir Moradi[1]

[1] Ruhr-University Bochum, Germany, Horst Görtz Institute for IT Security
firstname.lastname@rub.de,
[2] imec - COSIC, KU Leuven, Belgium
lauren.demeyer@esat.kuleuven.be

Abstract. The effort in reducing the area of AES implementations has largely been focused on Application-Specific Integrated Circuits (ASICs) in which a tower field construction leads to a small design of the AES S-box. In contrast, a naive implementation of the AES S-box has been the status-quo on Field-Programmable Gate Arrays (FPGAs). A similar discrepancy holds for masking schemes – a well-known side-channel analysis countermeasure – which are commonly optimized to achieve minimal area in ASICs.
In this paper we demonstrate a representation of the AES S-box exploiting rotational symmetry which leads to a 50% reduction of the area footprint on FPGA devices. We present new AES implementations which improve on the state of the art and explore various trade-offs between area and latency. For instance, at the cost of increasing 4.5 times the latency, one of our design variants requires 25% less look-up tables (LUTs) than the smallest known AES on Xilinx FPGAs by Sasdrich and Güneysu at ASAP 2016.
We further explore the protection of such implementations against side-channel attacks. We introduce a generic methodology for masking any n-bit Boolean functions of degree t with protection order d. The methodology is exact for first-order and heuristic for higher orders.
Its application to our new construction of the AES S-box allows us to improve previous results and introduce the **smallest** first-order masked AES implementation on Xilinx FPGAs, to-date.

Keywords: AES · SCA · DPA · Rotational Symmetry · Threshold Implementations · $d+1$ Masking · FPGA

1 Introduction

Ever since the introduction of Differential Power Analysis (DPA) by Kocher *et al.* [KJJ99], protecting cryptographic devices against Side-Channel Analysis (SCA) has been a challenging and active area of research. A notable category of countermeasures is masking, in which a secret value is distributed among shares, which do not reveal any information about the secret separately. We speak of a d^{th}-order DPA attack when the adversary exploits the statistical moments of the SCA leakages (e.g., power consumption) up to order d. Such estimated statistical moments are expected to be independent of the secret, when sensitive variables are shared into $d+1$ shares.

Masking. In 2003, Ishai *et al.* [ISW03] introduced the d-probing model, in which a very powerful attacker has the ability to probe the exact values of up to d intermediate variables. Security in this model has been related to more realistic adversary scenarios such as the noisy leakage [CJRR99] and the bounded moment leakage model [BDF+17]. However, in 2005 it was noted by Mangard *et al.* [MPO05] that the Boolean masking schemes which are secure in sequential platforms [Tri03, ISW03] still exhibit side-channel leakage when implemented in hardware. This is due to unintended transitions (or *glitches*) on wires before they stabilize. For hardware implementations, the probing model was therefore redefined using glitch-extended probes [RBN+15]. The first masking scheme to achieve provable first-order security in the presence of glitches is Threshold Implementation (TI) [NRR06, NRS11], a particular realization of Boolean masking. As a result, the most challenging task in securing implementations is to mask the non-linear components of a cipher.

Masking schemes are typically introduced by means of a single description of a masked multiplier. Such constructions are easily extended to obtain a construction for a monomial of degree t, but it is not trivial to obtain a non-complete sharing of just any Boolean function. Ueno *et al.* [UHA17a] describe a generic method for constructing $d+1$-share maskings of any function of n variables. However, this method is not efficient for functions of many variables, since the number of output shares is expected to be $\mathcal{O}\left((d+1)^n\right)$. Bozilov *et al.* [BKN18] introduce a more efficient method for $d+1$-share maskings of functions of degree t, but only for functions with exactly $t+1$ variables.

AES S-box. The AES S-box is an algebraically-generated vectorial Boolean function with 8-bit input and 8-bit output. It consists of an inversion in $\mathrm{GF}(2^8)$ followed by an affine transformation over $\mathrm{GF}(2)^8$. Having a small implementation of this S-box is important to achieve compact AES hardware, especially in the context of masked implementations. The tower field decomposition has proved to be a valuable approach to implement the field inversion, resulting in small AES S-boxes by Satoh *et al.* [SMTM01], Mentens *et al.* [MBPV05] and finally Canright [Can05]. More recently, an even smaller S-box was created by Boyar *et al.* [BMP13] using a new logic optimization technique. This S-box implementation is the smallest to date. These S-box designs have all been successfully used to create the state-of-the-art smallest masked AES implementations [BGN+15, CRB+16, GMK17, UHA17b]. However, when it comes to Look-up Table (LUT) based FPGA implementations, these optimized constructions do not perform better than the 8 slices that are required for any 8-bit to 8-bit mapping such as the AES S-box.

Another line of work in this area [Wam14, WHS15, WS17] exploits a property of inversion-based S-boxes that any inversion in $\mathrm{GF}(2^n)$ can be implemented by a Linear Feedback Shift Register (LFSR). The ASIC-based smallest such construction [Wam14] needs on average 127 clock cycles, *i.e.* its latency depends on the given S-box input, hence is vulnerable to timing attacks. The idea has been further developed in [WHS15] leading to 7 clock cycles latency (on average) for one S-box evaluation, which for sure needs more area compared to the original design. The authors also presented a constant-time variant of their design with a latency of 16 clock cycles. The underlying optimizations are not FPGA specific, and achieving SCA-protection by means of masking on such a construction does not seem easily possible[1].

FPGA vs. ASIC. An FPGA design is indeed very different to its ASIC counterpart, most notably in the use of LUTs, which makes the number of inputs to a Boolean function a more defining factor for implementation cost than its algebraic complexity. Since the standardization of Rijndael as the AES, several successful efforts [CG03, BSQ+08, CB12] have been made to reduce its size on FPGAs. In 2016, Sasdrich *et al.* [SG16] introduced

[1]It is based on the fact that every $x \in \mathrm{GF}(2^8)$ is presented by α^n and its inverse by $\left(\alpha^{-1}\right)^n$. So, two LFSRs constantly multiply by α and α^{-1}. When one of them reaches x, the other one is x^{-1}. The concept does not work when x is shared by Boolean masking.

an unprotected AES implementation on Xilinx Spartan-6 FPGAs which occupies 21 slices and remains the smallest FPGA implementation of AES known to date. Notably in such a design, the S-box is naively implemented as an 8-to-8 look-up table. The authors furthermore introduced a variant with 24 slices that additionally realizes shuffling as a SCA-hardening technique. Note that we exclude the designs like [CG03, NBD+10, BGS+11, BGD12, BDGH15] from our comparisons as their constructions relay on the Block RAM (BRAM) modules.

While research on masking mostly targets ASIC designs, some efforts have been made to utilize the specific architecture of an FPGA. In 2012, Moradi and Mischke [MM12] investigated a glitch-free implementation of masking on FPGAs by avoiding the occurrence of glitches with a special enable-logic, which has been further re-developed in [MW15] by Moradi and Wild. Sasdrich *et al.* [SMMG15] used the field-programmability to randomize the FPGA configuration during runtime. Recently, Vliegen *et al.* [VRM17] investigated the maximal throughput of masked AES-GCM on FPGAs. However, their masked S-box is taken from [MPL+11] without further FPGA-specific improvements. We would like to emphasize that several AES masked FPGA designs have been reported in the literature which consider neither the glitches nor the non-completeness property defined in TI [NRS11]. For example, the masked S-box design used in [RWS11] is not different to Canright and Batina's design [CB08] which has been shown to have first-order exploitable leakage [MPO05, MME10].

Our Contribution. This is an extended work of [DMW18], in which we exclusively focus on FPGA devices and in particular those of Xilinx. All our case studies target a Xilinx Spartan-6 FPGA. We exploit a rotational symmetry property of Galois field power maps, *e.g.* the field inversion, to construct a novel structure realizing the AES S-box. This leads to an FPGA footprint of only 4 slices which is – to the best of our knowledge – smaller than any reported FPGA-based design of the AES S-box in the literature. Such an area reduction comes at the cost of a latency of 8 clock cycles for one S-box evaluation. We present several new AES implementations for Xilinx FPGAs. We adapt the currently smallest known FPGA-based AES design of [SG16] to use our S-box construction and achieve a new design that occupies only 17 slices - a 19% reduction over the previous record. We also restructure the smallest known ASIC-based AES design of [JMPS17] to efficiently use the FPGA resources and combine it with our S-box design, leading to another very small footprint of only 63 LUTs for the entire encryption function. Our designs use only FPGA LUTs and other slice-internal components such as slice registers and internal MUXes, but no block RAM (BRAM) which has been used in [BGS+11, NBD+10, BDGH15, BGD12] as a principle feature.

In the second part of this work, we implement our construction with resistance against SCA. To this end, we apply Boolean masking with a minimum number of two shares on a decomposition of the AES S-box, which again exploits the rotational symmetry. We detail a methodology for finding a d^{th} order non-complete masking of n-variable Boolean functions of degree t by splitting them into the minimal number of components necessary to achieve non-completeness. With our new method, the number of output shares is expected to be $\mathcal{O}\left((d+1)^t\right)$, which is far better than that of [UHA17a] when $n \gg t$.

Targeting an optimized implementation with respect to LUT utilization, we introduce a new masked AES design which far outperforms that of [DMW18] with a reduction of at least 20% in all resources (LUTs, flip flops and slices) and the randomness consumption reduced to one third. This is - to the best of our knowledge - the smallest masked AES design on Xilinx FPGAs. We deploy our design on a Spartan-6 and evaluate its SCA resistance by practical experiments.

2 Preliminaries

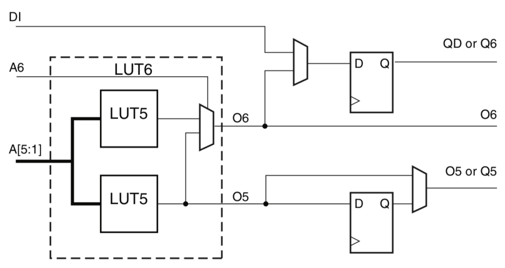
In the following we give an introduction to FPGA technology, Boolean algebra and masking schemes to counteract SCA attacks. Further, we define the notation for the rest of the paper.

2.1 FPGAs

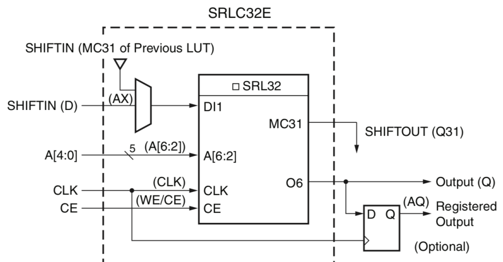
FPGAs are reconfigurable hardware devices consisting of configurable logic blocks (CLB). In modern Xilinx FPGAs, each CLB is further subdivided into two slices that each contain four look-up tables (LUTs), eight registers and additional carry-logic. In the following, we give a bottom-up description of the the structure of Xilinx Spartan-6 FPGAs, but this is similar for series 7 devices and FPGAs of other manufacturers.

2.1.1 LUTs

An FPGA's LUT is a combination of a multiplexer tree and RAM configured in read-only mode. The Xilinx 6 and 7 series contain one type of LUT block, which can be used to create functions with either six input bits and one output bit (O6) or five input bits and two output bits (O6,O5). This is illustrated in Figure 1a.



(a) Spartan-6 LUT block



(b) LUT as a 32-bit shiftregister

Figure 1: The illustrations are taken from [Xil10].

Because of this structure, the algebraic complexity of Boolean functions does not matter in FPGAs as long as the number of inputs is six or fewer. When realizing a vectorial Boolean function on FPGAs, two coordinates that jointly depend on five or fewer inputs can be mapped into one LUT. This puts FPGA design in stark contrast with ASIC design as they clearly demand very different optimization strategies to achieve a low-cost implementation.

There are alternative uses to the circuitry of a LUT. A single LUT[2] can also be configured as a 32-bit shift register with a 5-bit read address port in addition to serial *shiftin* and *shiftout* ports (see Figure 1b). It is also possible for a LUT to be used as 32 addressable RAM cells of two bits each or 64 RAM cells of one bit each.

2.1.2 Slices

When mapping a hardware design to an FPGA, we count the number of occupied slices as a metric for size. As each slice contains not only four LUTs but also further logic gates and registers, this opens up more optimization potential compared to a naive mapping to LUTs exclusively.

[2]Only in particular slice type *SliceM*.

More Inputs. Since each slice consist of four LUTs, it can trivially realize four 6-to-1-bit functions. Further, due to internal multiplexers between the four LUTs, each slice can also implement two 7-to-1-bit functions or one 8-to-1-bit function. As a result, the 8-bit AES S-box can be easily implemented in 8 slices; one for each Boolean coordinate function. In fact, this is the smallest known FPGA implementation of the AES S-box, used in [BSQ+08, SG16].

Memory. A slice also contains eight flip-flops, connected to the O5 and O6 output of each LUT (see Figure 1a). Note that every slice is limited in its functionality by many constraints. For example, while the inputs to four of the eight registers are directly accessible from the slice-external wires, a connection to the other four can only be made via the LUTs.

Types. In Spartan-6 devices we distinguish three different types of slices: The *SliceX* contains only four LUTs and eight flip-flops, while the *SliceL* contains additional carry logic and finally the most complex one, *SliceM*, can be used as a RAM unit with 256 bits of memory in different chunks of addressability or a 128-bit shift register.

2.1.3 Block RAM

Every Spartan-6 FPGA also contains a number of block RAMs (BRAMs), which can each store up to 18k bits of data and each have two independent read/write ports which can be simultaneously used. The ports can be configured to have various widths, ranging from 1 up to 18 bits, based on which the width of the address port is also derived. Each port has its own clock port, and any read/write operation is done in one clock cycle. The output ports can also be configured to have an extra register, with which the clock-to-output time of the read operation is prolonged. The number of BRAMs depends on the type of Spartan-6 device. The smallest device has only 12 BRAMs. Further, multiple BRAM instances can be cascaded to build larger ones. Due to their large storage space, the BRAMs are usually used for high-performance applications. As an example, we refer to fast pipeline implementations (*e.g.* of DES) reported in [GKN+08] which make use of BRAMs to accelerate the exhaustive search.

2.2 Mathematical Foundations

Boolean Algebra. We define $(\mathrm{GF}(2),\ +,\ \cdot)$ as the field with two elements Zero and One. We denote the n-dimensional vector space defined over this field by $\mathrm{GF}(2)^n$. Its elements can be represented by n-bit numbers and added by bit-wise XOR. In contrast, the Galois Field $\mathrm{GF}(2^n)$ contains an additional field multiplication operation. It is well known that $\mathrm{GF}(2)^n$ and $\mathrm{GF}(2^n)$ are isomorphic.

A Boolean function F is defined as $F : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$, while we call $G : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n$ a vectorial Boolean function. A (vectorial) Boolean function can be represented as a look-up table, which is a list of all output values for each of the 2^n input combinations. Alternatively, each Boolean function can be described by a unique representation - so called normal form. Most notably the Algebraic Normal Form (ANF) is the unique representation of a Boolean function as a sum of monomials. In this work, we designate by $m \in \mathrm{GF}(2^n)$ the monomial $x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$ where $(m_0, m_1, \dots, m_{n-1})$ is the bitvector of m. The monomial's algebraic degree is simply its hamming weight: $\deg(m) = \mathrm{hw}(m)$. We can then write the ANF of any Boolean function F as

$$F(x) = \bigoplus_{m \in \mathrm{GF}(2^n)} a_m x_0^{m_0} x_1^{m_1} \dots x_{n-1}^{m_{n-1}}$$

The algebraic degree of F is the largest number of inputs occurring in a monomial with a non-zero coefficient:

$$\deg(F) = \max_{m \in \mathrm{GF}(2^n), a_m \neq 0} \mathrm{hw}(m)$$

Finite Field Bases. We denote the isomorphism between the finite field $\mathrm{GF}(2^n)$ and the vector space $\mathrm{GF}(2)^n$ by $\phi : \mathrm{GF}(2^n) \to \mathrm{GF}(2)^n$. This mapping depends on the basis chosen for $\mathrm{GF}(2^n)$. The vector $\phi(x) = (a_0, \ldots, a_{n-1}) \in \mathrm{GF}(2)^n$ holds the *coordinates* of x with respect to that basis, and we denote by $\phi(x)_i$ the i^{th} coordinate of this vector. A polynomial basis has the form

$$(1, \alpha, \alpha^2, \ldots, \alpha^{n-1})$$

with $\alpha \in \mathrm{GF}(2^n)$ the root of a primitive polynomial of degree n. We denote ϕ^α the isomorphism mapping to a polynomial basis with α. Consider for example $\alpha = 2$. In that case, we have $\phi^2(2^i) = \mathbf{e}_i$ with $\mathbf{e}_i$ the i^{th} unit vector, so the representation of $x \in \mathrm{GF}(2^n)$ in polynomial basis simply corresponds to its binary expansion. In contrast, a normal basis has the form

$$(\beta^{2^0}, \beta^{2^1}, \ldots, \beta^{2^{n-1}})$$

with 2^{n-1} possible choices for $\beta \in \mathrm{GF}(2^n)$. In a normal basis over any finite field, the zero (resp. unit) element is represented by a coordinate vector of all zeros (resp. all ones). An element $\beta \in \mathrm{GF}(2^n)$ can thus form a normal basis if $\bigoplus_{i=0}^{n-1} \beta^{2^i} = 1$. We denote by $\phi_n^\beta(x)$ the isomorphic mapping from $x \in \mathrm{GF}(2^n)$ to its $\mathrm{GF}(2)^n$ representation in normal basis with β, although we sometimes omit β for ease of notation.

The conversion between any polynomial and normal basis is merely a linear transformation which can be represented by a matrix multiplication over $\mathrm{GF}(2)^n$. The matrix can be determined column-wise by mapping each basis element of the original basis to the target basis. Let $Q \in \mathrm{GF}(2)^{n \times n}$ be the matrix mapping from a normal basis with β to a polynomial basis with α, *i.e.* $Q \times \phi_n^\beta(x) = \phi^\alpha(x)$. Then, the i^{th} column of Q is simply $\phi^\alpha(\beta^{2^i})$. The inverse mapping uses the inverse matrix: $Q^{-1} \times \phi^\alpha(x) = \phi_n^\beta(x)$.

2.3 Boolean Masking in Hardware

We denote the s_i-sharing of a secret variable x as $\boldsymbol{x} = (x_0, \ldots, x_{s_i-1})$ and similarly an s_o-sharing of a Boolean function $F(x)$ as $\boldsymbol{F} = (F_0, \ldots, F_{s_o-1})$. Each component function F_i computes one share y_i of the output $y = F(x)$. A correctness property should hold for any Boolean masking:

$$x = \bigoplus_{0 \leq j < s_i} x_j \Leftrightarrow F(x) = \bigoplus_{0 \leq j < s_o} F_j(\boldsymbol{x})$$

We define $\mathcal{S}(x)$ as the set of all correct sharings of the value x. Creating a secure masking of cryptographic algorithms in hardware is especially challenging due to glitches. Despite this major challenge, Nikova *et al.* [NRR06] introduced a provably secure scheme against first-order SCA attacks in the presence of glitches, named Threshold Implementation (TI). A key concept of TI is the non-completeness property which we recall here.

Definition 1 (Non-Completeness)**.** A sharing $\boldsymbol{F}$ is non-complete if any component function F_i is independent of at least one input share.

Apart from non-completeness, the security proof of TI depends on a uniform distribution of the input sharing fed to a shared function $\boldsymbol{F}$. For example, when considering round-based block ciphers, the output of one round serves as the input of the next. Hence, a shared implementation of F needs to maintain this property of uniformity.

Definition 2 (Uniformity)**.** A sharing $\boldsymbol{x}$ of x is uniform, if it is drawn from a uniform probability distribution over $\mathcal{S}(x)$.

We call $\boldsymbol{F}$ a uniform sharing of $F(x)$, if it maps a uniform input sharing $\boldsymbol{x}$ to a uniform output sharing $\boldsymbol{y}$:

$$\exists c : \forall x \in \mathrm{GF}(2)^n, \forall \boldsymbol{x} \in \mathcal{S}(x), \forall \boldsymbol{y} \in \mathcal{S}(F(x)) : Pr(\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{y}) = c.$$

Finding a uniform sharing without using fresh randomness is often tedious [BNN+12, BB16] and may be impossible. Hence, many masking schemes restore the uniformity by re-masking with fresh randomness. When targeting first-order security, one can re-mask s output shares with $s-1$ shares of randomness as such:

$$(F_0 \oplus r_0,\ F_1 \oplus r_1,\ \ldots,\ F_{s-2} \oplus r_{s-2},\ F_{s-1} \oplus \bigoplus_{0 \le j \le s-2} r_j)$$

Threshold Implementation was initially defined to need $s_i \geq td+1$ shares with d the security order and t the algebraic degree of the Boolean function F to be masked. The non-completeness definition was extended to the level of individual variables in [RBN+15], which allowed the authors to reduce the number of input shares to $s_i = d+1$, regardless of the algebraic degree. As a result, the number of output shares s_o increases to $(d+1)^t$. For example, two shared secrets $\boldsymbol{a} = (a_0, a_1)$ and $\boldsymbol{b} = (b_0, b_1)$ can be multiplied into a 4-share $\boldsymbol{c} = (c_0, c_1, c_2, c_3)$ by just computing the cross products.

$$\begin{aligned} c_0 &= a_0 b_0 & c_1 &= a_0 b_1 \\ c_2 &= a_1 b_0 & c_3 &= a_1 b_1 \end{aligned}$$

The number of output shares can be compressed back to $d+1$ after a refreshing and a register stage. This method was first applied to the AES S-box in [CRB+16] and lead to a reduction in area, but an increase in the randomness cost. A similar method for sharing 2-input AND gates with $d+1$ shares is demonstrated by Gross *et al.* in [GMK16, GMK17]. In particular, they propose to refresh only the cross-domain products $a_i b_j$ for $i \neq j$, resulting in a fresh randomness cost of $\binom{d+1}{2}$ units. In [UHA17a], Ueno *et al.* demonstrate a general method to find a $d+1$-sharing of a non-quadratic function with $d+1$ input shares in a non-complete way by suggesting a probabilistic heuristic that produces $(d+1)^n$ output shares in the worst case, where n stands for the number of variables.

2.4 Rotational Symmetry of the AES S-box

Rotational Symmetry of Power Maps. In 2008, Rijmen *et al.* [RBF08] noted a rotational property of power maps in finite fields. More specifically, they showed that every power map based S-box (or vectorial Boolean function) over $\mathrm{GF}(2^n)$ is a rotation-symmetric S-box in a normal basis. For completeness, we repeat the most interesting results and proofs here. We denote by $\mathsf{rot}(v, i)$ the i-times rotation of $v \in \mathrm{GF}(2)^n$ to the right, *i.e.* $\mathsf{rot}(v, 1) = (a_{n-1}, a_0, \ldots, a_{n-2})$ when $v = (a_0, a_1, \ldots, a_{n-1})$. When i is omitted, it is equal to 1.

Definition 3 (Rotation-Symmetry)**.** An n-bit S-box $S : \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n$ is *rotation-symmetric* if and only if $\mathsf{rot}(S(v)) = S(\mathsf{rot}(v))$ for all $v \in \mathrm{GF}(2)^n$.

We consider a normal basis with β:

$$(\beta_0, \beta_1, \beta_2, \ldots, \beta_{n-1}) = (\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{n-1}})$$

This basis allows for an effective realization of squaring. As the order of the multiplicative group is $2^n - 1$, we derive that $\forall x \in \mathrm{GF}(2^n) : x^{2^n - 1} = 1$ by Lagrange's theorem. As a result, we have that $x^{2^n} = x$ for any element in $\mathrm{GF}(2^n)$. This leads to the following lemma.

Lemma 1 ([RBF08])**.** *In a normal basis over* GF(2^n)*, the squaring operation corresponds to a rotation of the coordinates vector:* $\phi_n(x^2) = \mathsf{rot}(\phi_n(x))$

Proof. We make use of the fact that $x = x^{2^n}$ holds for any element in GF(2^n).

$$\begin{aligned} x^2 &= a_0\beta_0^2 + a_1\beta_1^2 \ldots + a_{n-2}\beta_{n-2}^2 + a_{n-1}\beta_{n-1}^2 \\ &= a_0\beta^2 + a_1\beta^{2^2} \ldots + a_{n-2}\beta^{2^n-1} + a_{n-1}\beta^{2^n} \\ &= a_{n-1}\beta + a_0\beta^2 + a_1\beta^{2^2} \ldots + a_{n-2}\beta^{2^{n-1}} \\ &= a_{n-1}\beta_0 + a_0\beta_1 + a_1\beta_2 \ldots + a_{n-2}\beta_{n-1} \end{aligned}$$

Hence, the below equation holds.

$$\phi_n(x^2) = (a_{n-1}, a_0, \ldots, a_{n-2}) = \mathsf{rot}(\phi_n(x), 1)$$

□

Successive application of the above property yields the relation

$$\phi_n(x^{2^i}) = \mathsf{rot}(\phi_n(x), i).$$

Now consider a power map $F(x) = x^k$ over $GF(2^n)$. Clearly, for any power map we have that $F(x)^l = F(x^l)$. Let $S(\phi_n(x)) = \phi_n(F(x))$ be the normal basis S-box over GF$(2)^n$ for which $F(x)$ is an algebraic description. We denote the component Boolean functions by $S_i : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$. By Theorem 9 in [RBF08], S is thus rotation-symmetric, *i.e.* $\mathsf{rot}(S(v)) = S(\mathsf{rot}(v))$ for all $v \in \mathrm{GF}(2)^n$ or equivalently, for each $i \in \{0, \ldots, n-1\}$: $S_i(v) = S_0(\mathsf{rot}(v, i))$. All n output bits of the S-box can be calculated using the same Boolean function S_0. From now on, we denote the Boolean function that calculates the least significant bit of the S-box output as $S^*(v) = S_0(v)$. It is related to the power map function as follows: $S^*\left(\phi_n\left(x\right)\right) = \phi_n\left(F\left(x\right)\right)_0$. We demonstrate the rotational symmetry and show how to calculate the i^{th} coordinate of the power map's normal basis representation:

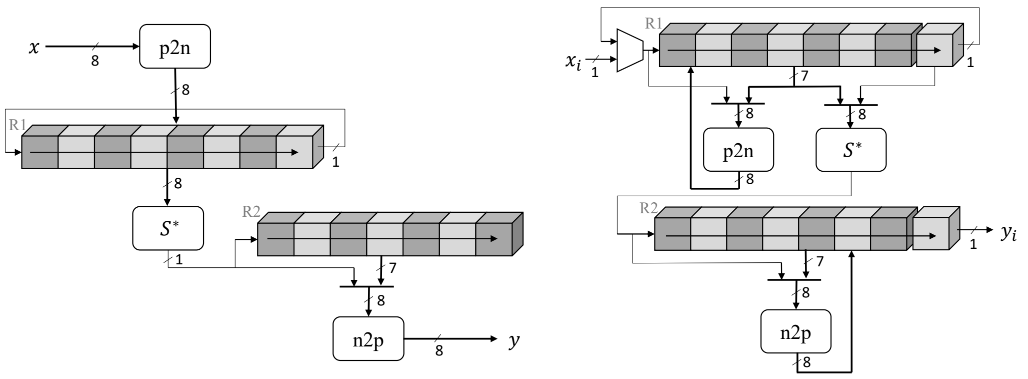
$$\begin{aligned} S_i(\phi_n(x)) = \phi_n\left(F\left(x\right)\right)_i &= \mathsf{rot}\left(\phi_n\left(F(x)^{2^i}\right), -i\right)_i \\ &= \mathsf{rot}\left(\phi_n\left(F\left(x\right)^{2^i}\right), 0\right)_0 \\ &= \phi_n\left(F\left(x\right)^{2^i}\right)_0 \\ &= \phi_n\left(F\left(x^{2^i}\right)\right)_0 \\ &= S^*\left(\phi_n\left(x^{2^i}\right)\right) \\ &= S^*\left(\mathsf{rot}\left(\phi_n\left(x\right), i\right)\right) \end{aligned}$$

Note that ϕ_n and by extension S^* depend on the choice of β, which generates the normal basis, but we omit β here for readability.

As a result, instead of n Boolean functions $S_0, S_1, \ldots, S_{n-1}$ operating in parallel, the power map based S-box S can be evaluated entirely with a single n-to-1-bit function S^* by rotating the input vector bitwise.

3 Unprotected AES on FPGA

It is generally known that an optimal FPGA implementation of the AES S-box requires 32 LUTs in eight slices, as each of its eight coordinate functions is an 8-to-1 mapping (see Section 2.1.2). There is no obvious way to reduce this number, as every linear combination



(a) byte-parallel loading (b) bit-serial loading

Figure 2: Illustration of the bit-serial AES S-box based on rotational symmetry.

of coordinate functions maintains the maximal algebraic degree of seven and depends on all eight inputs. Hence, every coordinate function occupies an entire slice.

Note that Canright's tower field construction [Can05] does not provide an alternative as it is ill-suited for Spartan-6 devices due to the underutilization of six-input LUTs by the operations in $GF(2^4)$ and even $GF(2^2)$. More precisely, realizing the basis conversion, square-scaling, inversion and multiplications can occupy as much as 53 LUTs on an FPGA.

3.1 Optimizing the S-box for FPGA

S-box Structure. We demonstrate that it is indeed possible to realize the AES S-box in fewer LUTs by trading off latency for area. Recall that the AES S-box consists of an inversion in $GF(2^8)$, followed by an affine transform over $GF(2)^8$. For the inversion part, we exploit the rotational symmetry of the power map x^{254} in $GF(2^8)$ as explained in Section 2.4. The structure is illustrated in Figure 2a. Since the AES inversion is defined in a polynomial basis with $\alpha = 2$, we first convert the input byte x to a normal basis using a linear transform ("p2n"). Then, in a bit-wise fashion, we calculate the output of the rotation-symmetric S-box by rotating the first register R1. The single-bit output of S^* is shifted into a second register R2. When all eight bits have been calculated, we use another linear transform to convert the result back into the polynomial basis ("n2p"). This transform is combined with the affine transform of the AES S-box.

S-box Implementation Cost. We examine various normal bases and target a minimal number of LUTs needed to implement the 8-to-8-bit functions p2n and n2p. Note that it is not required to optimize S^* since it is an 8-to-1-bit Boolean function of algebraic degree 7 and requires 4 LUTs (an entire slice) in any normal basis. We exhaustively enumerate all choices of β and pick the one that gives the most optimal implementation of p2n and n2p in terms of LUT count. Since p2n and n2p each have 8 output bits and each LUT can compute at most 2 bits, the minimum number of LUTs required to implement them is 4. We obtain this for $\beta = 145$.[3] By optimizing our implementation for intensive usage of 5-to-2 LUTs, we can implement the affine transformations p2n and n2p and the rotating register R1 in one slice each. More specifically, the affine transforms each consume 4 LUTs. The 8-bit register R1 uses all 8 registers in a slice. The choice between parallel loading and rotational shifting is achieved using the four LUTs of that slice. As mentioned

[3]The algebraic normal forms for S^*, p2n and n2p are given in Appendix A

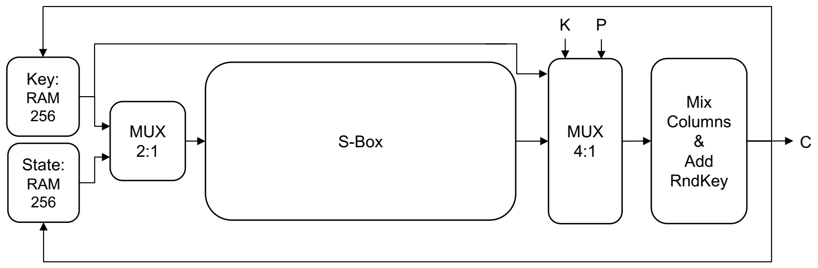


Figure 3: Illustration of the byte-wise AES design by [SG16]. All wires are 8-bit wide. Especially notable is the 8-bit aggregation register in the MixColumns block. The RAM blocks are further divided into two parts of 128 bits which are used in alteration.

previously, S^* itself also occupies 1 slice. Finally, the 7 slice flip-flops for R2 are found in the already used slices for n2p, p2n and S^*. In total, the S-box design occupies 16 LUTs and 15 registers, all fitting into only 4 slices. This means a 50% reduction over the status-quo [BSQ+08, SG16].

We pay for the reduction in area with latency. While the 32-LUT S-box computes the output within one clock cycle, our bit-serialized approach (Figure 2a in 16 LUTs) increases the latency to 8 clock cycles. The linear function p2n is applied immediately to the S-box input x. In cycles 1 to 8, register R1 rotates while S^* serially computes each output bit. The outputs are shifted into R2 bit by bit. In the last cycle, the last output bit is combined with the 7-bit content of R2 as input to the affine transform n2p, which computes the S-box output y. The register bypassing of n2p allows the S-box latency to be 8 cycles and the R2 register to be only 7 bits wide.

3.2 Fully Byte-serial AES

A Grain in the Silicon. We start from the smallest unprotected state-of-the-art AES design for FPGA [SG16] illustrated in Figure 3. The entire implementation requires only 21 slices, of which 15 slices construct the round function and key schedule, including 8 slices for the AES S-box and 2 slices configured as 256-bit memory for the state and key arrays. The round constants are also stored in this memory. The remaining 6 slices make up a heavily optimized control unit with a finite state machine (FSM) of 32 states. Each round in this design requires 147 clock cycles. In the first 50 cycles, the key schedule is performed to compute the entire 128-bit key state of the current round. In the next 97 cycles the round function is computed, using the freshly calculated round key. Most of these clock cycles is spent on the MixColumns operation because it performs 4 S-box evaluations on the fly for each byte of the MixColumns output. The S-box outputs are not stored but discarded and recomputed when needed. Therefore, 64 S-box invocations (instead of 16) are performed. In the last round, MixColumns is omitted and the round function takes only 33 clock cycles. With 65 cycles spent on loading a new plaintext and key, an entire encryption has a latency of $(65 + (50 + 97) \times 9 + 50 + 33) = 1\,471$ clock cycles. For more details on this design, we refer to the original work [SG16].

Latency optimization. We note that the above design can be optimized with respect to latency without sacrificing its minimal area requirement. Instead of performing the key schedule and round function separately in each round, we can interleave them, *i.e.* we compute one key byte and immediately use it to update the corresponding state byte. To do this, we only have to adapt the control logic. We create a new FSM of 16 states and derive the LUT mappings for the control signals and addresses. We decrease the number of

Table 1: Overview of unprotected AES implementations for FPGA

Design	**# LUTs**	**# Flip flops**	**# Slices**	**# CCs***	$\mathbf{f_{max}}^\dagger$
Sasdrich *et al.* [SG16]	84	24	21	1 471	108 MHz
Latency optimized	81	**21**	21	**1 098**	113 MHz
With bit-serial S-box	68	39	**17**	5 538	109 MHz
Fully bit-serialized	**63**	38	19	4 852	**155 MHz**

$*$ Number of clock cycles
† From the Post-PAR Static Timing Report

LUTs from 24 to 21 and the number of flip flops from 16 to 13. The resulting design has a latency of 113 clock cycles per round, except 49 in the last round. Loading of plaintext and key bytes is done in 32 cycles. In total, one encryption requires $(32 + 113 \times 9 + 49) = 1\,098$ clock cycles. Note that this design retains the original 8-LUT S-box. It is summarized in row 2 of Table 1.

Bit-serializing the S-box. We now start from the latency-optimized design and replace the 8-slice byte-parallel S-box with our bit-serialized S-box. Since the AES architecture is byte-serial, we use the S-box from Figure 2a, which can load entire bytes in parallel. We accordingly change the control unit to make use of such an S-box design by means of an extra 3-bit counter to account for the S-box latency. It still contains an FSM of 16 states. This results once again in a control unit of 24 LUTs and 16 flip flops. Each cipher round now has a latency of 589 clock cycles and the last round 205 cycles. Hence, one encryption is completed in $(32 + 589 \times 9 + 205) = 5\,538$ clock cycles. An overview of the post-map area and latency of this designs is shown in row 3 of Table 1. We can fit the entire AES encryption into only 17 slices, a 19% reduction over the state-of-the-art.

3.3 Fully Bit-serial AES

We now combine our bit-serialized AES S-box with the bit-serialized AES implementation of [JMPS17]. We first adopt the S-box for bit-serial loading and then we adopt their AES design for FPGAs, since it originally targets ASIC platforms.

S-box. The structure of the bit-serialized S-box with bit-serial loading is shown in Figure 2b. The conversions to and from the normal basis (p2n and n2p modules) are now realized in 12 LUTs, *i.e.* 3 slices (including the S-box affine). This is more than before because these LUTs also implement the choice between the parallel and shift-serial input to R1 and R2. This new constraint requires a different normal basis than before to achieve the stated size. Again, by exhaustive search, we obtain $\beta = 133$.[4] As a result, shift-registers R1 and R2 only require 16 more flip-flops, for which we can use the same slices. The 8-to-1-bit Boolean function S^* still occupies exactly 4 LUTs of a slice. Therefore, the entire S-box circuit, *i.e.* all elements and components shown in Figure 2b, requires only 16 LUTs and 16 flip-flops fitting into 4 slices (again 50% less area compared to [SG16]).

The S-box now has a latency of 16 cycles. In cycles 1 to 7, input bits are shifted into the first register. In cycle 8, the linear conversion p2n is applied to the 7-bit content of the register and the newest incoming bit at input x_i. The 8-bit result is written to that same register in parallel in the same cycle. In the 8 subsequent cycles (9 to 16), this register is rotated, which allows S^* to evaluate the 8-bit output. The first 7 bits are shifted serially into R2. In cycle 16, the affine conversion n2p is applied to the 7 bits stored in R2 and the last output of S^*. The result is written in parallel to R2. The AES S-box output y is then

[4]The algebraic normal form for S^*, p2n and n2p are given in Appendix B

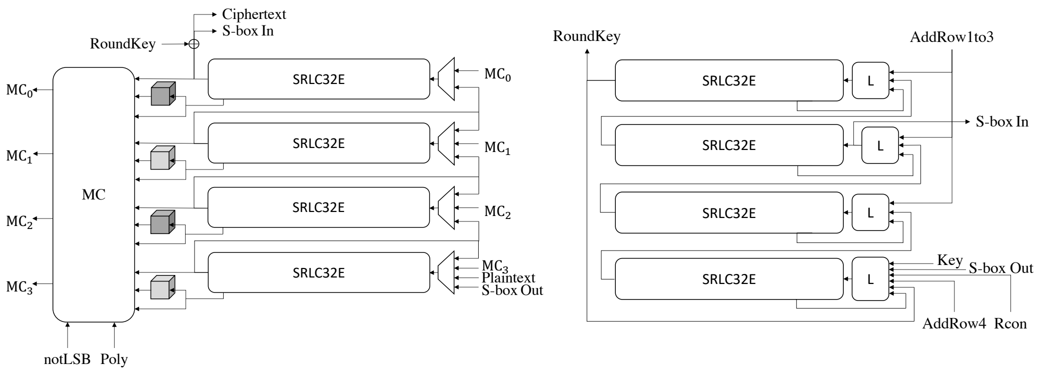


Figure 4: Bit-serial architecture for AES-128. Left: State Array and Round Function, Right: Key Schedule

ready to be shifted out serially over 8 cycles. Note that this can be done in parallel with the feeding of the next S-box input into R1.

Architecture. Our design is shown in Figure 4. We refer to [JMPS17, Fig. 3,4] for the corresponding original architecture. To accommodate for bit-sliding, we instantiate four LUTs as 32-bit shift registers (SRLC32E, see Figure 1b) for both the state and key arrays. Each LUT represents one row of the array and has its own shift enable signal (not drawn). This means that ShiftRows can be implemented without additional area cost by letting row $i \in \{0, 1, 2, 3\}$ shift $8i$ times. This requires 24 clock cycles in total. As shown in Figure 1b, the shift register LUT has both a serial output and a custom read port. In the state array, this port reads the next-to-last bit, which is used in the computation of MixColumns. In the key array, this port reads the 7th bit of each row. The MixColumns is performed in 32 clock cycles as in [JMPS17]. The implementation uses 6 LUTs and 4 flip flops (for the four most significant bits). We plug in the 16-LUT S-box as described in Section 3.1. With a bit-serial loading of the input, the S-box has a latency of 16 clock cycles. The same S-box is shared between the round function and key schedule. The multiplexers in the state array can be implemented using 4 LUTs. The same goes for the operations at the input of each row of the key state. We also have one LUT for the AddRoundKey which also includes two multiplexers to select the serial input to R1. On the one hand, it chooses x_i between the S-box input from the round function and from the key schedule. On the other hand, it chooses the feedback from R1 when R1 should be rotating, *i.e.* the multiplexer shown in Figure 2b.

Finally, we make a controller to supply the control signals, read addresses and round constant to the round function, key schedule and S-box. The controller consists of an FSM with 8 states, which are encoded in a way that minimizes the number of LUTs needed to compute the control signals and addresses. In total, the control unit takes up 24 LUTs and 18 flip flops. This brings the total LUT cost of the AES implementation on a new record of 63 LUTs (see Table 1, row 4). The bit-serial loading of plaintext and key requires 128 clock cycles. Each encryption round is done in 476 cycles, except the last round, which is done in 440 cycles. In total, one encryption takes $(128 + 476 \times 9 + 440) = 4\,852$ clock cycles. It might be surprising that this bit-serialized design is faster than the byte-serialized AES from Section 3.2. This is due to the high latency of the S-box and the fact that the architecture of [SG16] has a “wasteful” MixColumns implementations that evaluates the S-box multiple times.

A note on BRAM. Our construction inherits the architecture of the formerly-smallest design [SG16], where no BRAM is used. Since the only non-linear function in our construction is the 8-bit to 1-bit serialized S-box, dedicating an 18k-bit BRAM to such a small function would be wasteful. As stated in Section 2.1.3, the smallest Spartan-6 device has only 12 of such BRAM instances. Hence, our underlying idea is to realize the AES module in such a way that its insertion to any application would lead to a negligible resource utilization. To this end, we have not made use of any BRAMs in our design.

4 Masking Methodology for Functions of Degree t

The rotational symmetry approach to implement the AES S-box reduces its non-linear proportion significantly. This is especially interesting when we consider the application of masking schemes. It is well known that the non-linear parts of a circuit grow exponentially with the masking order, while linear operations can simply be duplicated and performed on each share independently, *i.e.* a linear increase in the area. Instead of sharing a complete 8-bit to 8-bit mapping, the rotational symmetry approach allows us to mask only a single 8-to-1 Boolean function.

In this section, we introduce a generic methodology for masking any degree-t function. Our descriptions have our AES application in mind, but can be generalized to any algebraic degree and any number of inputs. Moreover, the methodology is not platform-specific and can be used both for ASIC and FPGA implementations.

Masking Cubic Boolean Functions with $d+1$ shares. Each cubic monomial abc can be trivially masked with $d+1$ input shares and $(d+1)^3$ output shares (one for each crossproduct). For example, a first-order sharing (*i.e.* $d=1$) of $z=abc$ is given in (1).

$$\begin{aligned} &z_0 = a_0b_0c_0, \quad && z_1 = a_0b_0c_1, \quad && z_2 = a_0b_1c_0, \quad && z_3 = a_0b_1c_1, \\ &z_4 = a_1b_0c_0, && z_5 = a_1b_0c_1, && z_6 = a_1b_1c_0, && z_7 = a_1b_1c_1 \end{aligned} \tag{1}$$

The result can be compressed back into $d+1$ shares after a refreshing and register stage. Our refreshing strategy resembles that of Domain Oriented Masking [GMK16] in such a way that we apply the same bit of fresh randomness to cross-share terms and do not re-mask inner-share terms:

$$\begin{aligned} z'_0 &= [z_0]_{reg} \oplus [z_1 \oplus r_0]_{reg} \oplus [z_2 \oplus r_1]_{reg} \oplus [z_3 \oplus r_2]_{reg} \\ z'_1 &= [z_4 \oplus r_2]_{reg} \oplus [z_5 \oplus r_1]_{reg} \oplus [z_6 \oplus r_0]_{reg} \oplus [z_7]_{reg} \end{aligned} \tag{2}$$

Note that every term after refreshing *e.g.* z_0 or $z_1 \oplus r_0$, is stored in a dedicated register before going to the XOR chain which produces z'_0 and z'_1.

The most basic way to mask a more general t-degree function is thus to expand each monomial into $(d+1)^t$ shares. However, this is wildly inefficient for a Boolean function which can have as many as 20 monomials (in our case). On the other hand, it is impossible to keep certain monomials together without violating non-completeness. We devise a sharing method that keeps as many monomials as possible together by splitting the function into a *minimum* number of sub-functions. These sub-parts are functions such as for example $z = abc \oplus abd$, for which it is trivial to find a non-complete sharing. For each sub-function we create independent sharings, each with $(d+1)^t$ output shares, and recombine them during the compression stage.

4.1 Sharing Matrices

We introduce a matrix notation in which each column represents a variable to be shared and each row represents an output share domain. Output share j only receives share M_{ij} of variable i. For example, the sharing matrix M of the sharing in Equation (1) is

$$M = \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{array} \tag{3}$$

From this matrix, it is clear that a correct and non-complete sharing for the cubic function $z = abc$ exists, since the 2^3 rows of the matrix are unique, *i.e.* each of the 2^3 possible rows occur in the matrix. Moreover, this Sharing matrix implies a correct and non-complete sharing for any function $z = f(a, b, c)$. Note also that each column is balanced, *i.e.* there are an equal number of 0's and 1's. It is also possible to add a fourth column, such that any submatrix of three columns consists of unique rows:

$$M' = \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{array} \tag{4}$$

Hence, the matrix M' demonstrates the possibility to find a correct and non-complete sharing with eight output shares for any combination of cubic monomials defined over four variables a, b, c, d. Note that the non-completeness follows from the fact that each output share (row) only receives one share of each input (column) by construction. To generalize this observation, we introduce the following concepts:

Definition 4 (Sharing Vector)**.** We call a vector v of length $(d+1)^t$ with entries $v_i \in \{0, \ldots, d\}$ a (t, d)*-Sharing Vector*, if and only if it is balanced, *i.e.* each entry occurs an equal number of times:

$$\forall \tau \in \{0, \ldots, d\} : \#\{i | v_i = \tau\} = (d+1)^{t-1}$$

Definition 5 (Sharing Matrix)**.** We call a $(d+1)^t \times c$ matrix M with entries $M_{ij} \in \{0, \ldots, d\}$ a *(t, d)-Sharing Matrix*, if and only if every column M_j is a (t, d)-Sharing Vector and if every $(d+1)^t \times t$ sub-matrix of M contains unique rows.

4.1.1 How to construct Sharing Matrices

The main question in creating masked implementations is thus how to find such a (t, d)-Sharing Matrix. Below, we present both provable theoretical and experimental results:

Exact.

Lemma 2. *A (t, d)-Sharing Matrix with t columns exists and is unique up to a reordering of rows.*

Proof. A (t,d)-Sharing Matrix has exactly $(d+1)^t$ rows. If the matrix has t columns, then each row is a t-length word with base $d+1$. The existence of such a matrix follows trivally from choosing as its rows all $(d+1)^t$ elements from the set $\{0,\dots,d\}^t$. The uniqueness follows from the fact that the rows must be unique, hence each of the $(d+1)^t$ elements can occur exactly once. Up to a permutation of the rows, this matrix is thus unique. □

Lemma 2 is equivalent to the fact that it is trivial to mask t-variable functions of degree t (*e.g.* $z = abc$) with $(d+1)^t$ output shares but also functions such as $z = abc + abd$ (since c and d can use the same Sharing Vector).

Lemma 3. *A $(t,1)$-Sharing Matrix has at most $c = t+1$ columns.*

Proof. We prove this Lemma by showing that the $t+1^{\text{th}}$ column M_t exists and is unique. Consider the Sharing Matrix M from Lemma 2 with t columns and 2^t rows. We reorder the rows as in a Gray Code. This means that every two subsequent rows have only one coordinate (or bit) different. Equivalently, since there are t columns, any two subsequent rows have exactly $t-1$ coordinates in common. Consider for example row i and $i+1$. We have the following properties:

$$\exists! \bar{j} \text{ s.t.} \qquad M_{i,\bar{j}} \neq M_{i+1,\bar{j}} \tag{5}$$

$$\forall j \in \{0,\dots,t-1\} \setminus \{\bar{j}\}: \qquad M_{i,j} = M_{i+1,j} \tag{6}$$

Recall that by definition of Sharing Matrix M, any two rows may have at most $t-1$ coordinates in common. For row i and $i+1$, these coordinates already occur in the first t columns (6), hence for the last column we must have:

$$M_{i,t} \neq M_{i+1,t}$$

Since this condition holds for ever pair of subsequent rows i and $i+1$, we can only obtain the alternating sequence ...010101... as the last column M_t. This column is therefore unique up to an inversion of the bits. An example for $t=3$ is shown below:

$$M = \begin{pmatrix} 0&0&0\\0&0&1\\0&1&0\\0&1&1\\1&0&0\\1&0&1\\1&1&0\\1&1&1 \end{pmatrix} \xrightarrow{\text{Gray Code}} \begin{matrix} 0&0&0\\0&0&1\\0&1&1\\0&1&0\\1&1&0\\1&1&1\\1&0&1\\1&0&0 \end{matrix} \rightarrow M_t = \begin{matrix} 0\\1\\0\\1\\0\\1\\0\\1 \end{matrix} \text{ OR } \begin{matrix} 1\\0\\1\\0\\1\\0\\1\\0 \end{matrix} \tag{7}$$

The example shows clearly that adding both columns to the matrix would violate the Sharing Matrix definition, since a 3-column submatrix including both new columns cannot have unique rows. Hence, the $t+1^{\text{th}}$ column is unique and thus a $(t,1)$-Sharing Matrix has at most $t+1$ columns. Note also that the labels 0/1 in the last column correspond to a partitioning of the rows in the first t columns based on odd or even hamming weight. □

An alternative proof using graph theory is shown in Appendix C.

While the relation between the degree t and the maximum number of columns in a (t,d)-Sharing Matrix is easily described for masking order $d=1$ (cf. Lemma 3), no simple formula can describe the relationship for higher orders. More general $(d+1)$-ary Gray Codes exist, but the proof of Lemma 3 does not result in uniqueness for $d > 1$. We therefore construct an algorithmic procedure for finding Sharing Matrices for higher orders. The results are shown in Table 2.

Search procedure with backtracking. We start from the t-column (t,d)-Sharing Matrix from Lemma 2. To extend this matrix with another column M_t, we keep for each column element $M_{i,t}$ a list $\mathcal{L}_{i,t}$ of non-conflicting values $\in \{0,\ldots,d\}$. For each new column, these lists are initialized to all possible values. Without loss of generalization, we set the first element of the column to zero: $M_{0,t} = 0$. For every row i with $t-1$ common coordinates, this element then needs to be removed from its list $\mathcal{L}_{i,t}$.

If there is a row r with a list of length 1 ($|\mathcal{L}_{r,t}| = 1$), then the unique value in that list is chosen as the value $M_{r,t}$. Again, this value is subsequently removed from all lists $\mathcal{L}_{i,t}$ for which row i has $t-1$ coordinates in common with row r. This process continues until either the column M_t is complete, or until there are only lists of length > 1. In the latter case, any element of the list $\mathcal{L}_{i,t}$ can be chosen as the value $M_{i,t}$. The choice is recorded so that it can later be revoked during backtracking. Whenever a value is assigned to a column element, the remaining lists are updated as before. When a column is fully determined, the next column is added in the same way. As soon as an empty list is obtained for one of the column elements, the algorithm backtracks to the last made choice. If for all possible choices empty lists occur, then the maximum number of columns is obtained and the algorithm stops.

A simplified version of the procedure is shown in Algorithm 3 in Appendix E. Note that optimizations are possible for the algorithm, but we leave this for future work since first-order security is the target in this work. According to the proof of Lemma 3, backtracking is not necessary for $d = 1$.

Table 2: Maximum Number of Columns in (t,d)-Sharing Matrices

Degree t	**Order** $d=1$	**Order** $d=2$	**Order** $d=3$
2	3	4	5
3	4	4	6
4	5	5	5
5	6	6	6*
6	7	7	7*
7	8	8	8*

* Results of greedy search without backtracking

Table 2 shows that the maximum number of columns does not follow a simple formula for $d > 1$. The results in Table 2 without additional indication have been obtained by exhausting all possible choices via backtracking which takes fractions of seconds for $d = 1$ and up to several minutes for $d = 2$ and multiple hours for the parameters $t = 4, d = 3$. As this strategy becomes infeasible with larger matrices, we indicate results of greedy search without backtracking with an asterisk. This choice is made based on the observation that (for smaller parameters), if a solution exists, backtracking was never necessary to find it.

4.1.2 From Sharing Matrices to Sharings

Now consider a mapping $\rho : \{0,\ldots,n-1\} \to \{0,\ldots,c-1\}$ which assigns any input variable x_i to a single column of a Sharing Matrix. That column holds the Sharing Vector of that variable. For a monomial to be shareable according to those Sharing Vectors, each variable of that monomial must be mapped to a different column. We therefore introduce the concept of *compatability* between monomials and a mapping ρ.

Definition 6 (Compatible Mappings)**.** A mapping $\rho : \{0,\ldots,n-1\} \to \{0,\ldots,c-1\}$ is compatible with a monomial $x_0^{m_0} x_1^{m_1} \ldots x_{n-1}^{m_{n-1}}$ of degree $hw(m) = t$ if it maps each variable in the monomial to a different Sharing Vector, *i.e.*

$$\forall i \neq j \in \{0,\ldots,n-1\} \text{ s.t. } m_i = m_j = 1 : \rho(i) \neq \rho(j)$$

Lemma 4. *Consider a set of monomials of degree $\leq t$ (of which at least one monomial has degree t) defined over a set of n variables with ANF*

$$\bigoplus_{m \in GF(2^n)} a_m x_0^{m_0} x_1^{m_1} \ldots x_{n-1}^{m_n-1}$$

and a sharing of each variable x_i into $d+1$ shares. A correct and non-complete sharing of this set of monomials with $(d+1)^t$ output shares exists if and only if a (t,d)-Sharing Matrix can be constructed such that for each variable in the set of monomials, the Sharing Matrix has exactly one column corresponding to its Sharing Vector and such that for each monomial, the (up to) t variables of that monomial have different Sharing Vectors. In other words, there exists a single mapping $\rho : \{0,\ldots,n-1\} \to \{0,\ldots,c-1\}$ that is compatible with each monomial in the ANF:

$$\forall m \in GF(2^n) \text{ s.t. } a_m = 1 : \forall i \neq j \in \{0,\ldots,n-1\} \text{ s.t. } m_i = m_j = 1 : \rho(i) \neq \rho(j)$$

The mapping ρ assigns to each variable x_i column $\rho(i)$ of the Sharing Matrix as Sharing Vector.

The terms with degree lower than t also have to be compatible with the mapping ρ so that their variables are assigned to different Sharing Vectors. However, lower-degree terms naturally do not need to appear in each of the $(d+1)^t$ output shares. Given a monomial of degree $l < t$ and a set of l (t,d)-Sharing Vectors, it is trivial to choose the $(d+1)^l$ output shares for the monomial to appear in.

We note that our Sharing Matrices are very similar to the D_t^n-tables of Bozilov *et al.* [BKN18], who also demonstrated that any t-degree function with $t+1$ input variables can be shared with the minimal $(d+1)^t$ output shares. However, their work only treats the sharing of t-degree functions with exactly $t+1$ input variables. Since our goal is to find a sharing of cubic functions with 8 input variables, we consider here the more general case where both the degree t and the number of variables n are unconstrained.

4.2 Sharing any ANF

Naturally, not any function is compatible with a (t,d)-Sharing Matrix. In what follows, we develop a heuristic method to determine efficient maskings with $d+1$ shares for any degree t-Boolean function starting from its unshared algebraic normal form (ANF). If a compatibility mapping with a single Sharing Matrix cannot be found, our approach is to split the monomials of the ANF into a number of subgroups, each for which a (t,d)-Sharing Matrix and thus a correct and non-complete sharing exists. If the ANF is split into s subgroups, then the number of intermediate shares before compression is $s \times (d+1)^t$. Our methodology finds the optimal sharing in terms of parameter s. We do not claim optimality in the number of intermediate shares, since the minimum is not necessarily a multiple of $(d+1)^t$.

Our Heuristic. We want to minimize the number of parts the ANF should be split into. This is equivalent to restricting the expansion of the number of shares and thus limiting both the required amount of fresh randomness and the number of registers for implementation.

We assume a (t,d)-Sharing Matrix of c columns is known at this point. A procedure for this was described in §4.1 and Algorithm 3. There are c^n possible mappings ρ to assign one of the c Sharing Vectors to each of n variables. In an initial preprocessing step, we iterate through all possible ρ and determine which t-degree monomials are compatible with it. During this process we eliminate redundant mappings (*i.e.* with an identical list of compatible monomials) and the mappings without compatible monomials of degree t.

Note that up to this point (including for algorithm 3), the specific function to be shared does not need to be known.

The next step is function specific: We first attempt to find one mapping that can hold all the monomials of the ANF. Its existance would imply that all the monomials in the ANF can be shared using the same Sharing Matrix (see Lemma 4). This is not always possible and even extremely unlikely for ANFs with many monomials. If this first attempt is unsuccessful, we try to find a *split* of the ANF. A *split* is a set of mappings that jointly are compatible with all monomials in the ANF of the Boolean function, *i.e.* it implies a partition of the ANF into separate sets of monomials, each for which a Sharing Matrix exists. In this search, we first give preference to partitions into a minimal number of subfunctions. With an FPGA target in mind, we also attemp to minimize the number of variables each subfunction depends on. It is trivial to change this for ASIC implementations.

We perform the above described search for all possible normal bases. We note that our search is heuristic and we do not claim optimality except in the number of split groups s.

Implementation Details. We encode mappings and ANFs which are dependent on n inputs as bitvectors with 2^n entries. An entry in the bitvector at position $m \in \mathrm{GF}(2^n)$ corresponds to one monomial $x_0^{m_0}x_1^{m_1}\dots x_{n-1}^{m_{n-1}}$ of degree $t = \mathrm{hw}(m)$ and prescribes whether this monomial is present in the ANF. Recall the ANF of an n-bit Boolean function F:

$$F(x) = \bigoplus_{m \in \mathrm{GF}(2^n)} a_m x_0^{m_0}x_1^{m_1}\dots x_{n-1}^{m_{n-1}}$$

We thus define the bitvector representations

$$\mathrm{rep}(F) = \sum_m a_m 2^m \qquad \text{and} \qquad \mathrm{rep}(\rho) = \sum_m \alpha_m^\rho 2^m$$

where $\alpha_m^\rho = 1$ if monomial m is compatible with mapping ρ. Consider for example the function $F = x_0x_2x_4 \oplus x_1x_5$:

$$\mathrm{rep}(x_0x_2x_4 \oplus x_1x_5) = \left(2^{2^0+2^2+2^4}\right) + \left(2^{2^1+2^5}\right) = \texttt{0x400200000}$$

Now, we can determine whether for example a set of mappings (ρ_1, ρ_2) specifies a two-split for a Boolean function F as follows. Assuming both are represented as a 2^n-bit vector, we check if the following condition holds:

$$\mathrm{rep}(\rho_1) \,|\, \mathrm{rep}(\rho_2) \,|\, \mathrm{rep}(F) = \mathrm{rep}(\rho_1) \,|\, \mathrm{rep}(\rho_2),$$

where $|$ refers to the Boolean OR-operation. The condition evaluates to *true* whenever all monomials of the ANF of F are also compatible monomials with at least one of the mappings ρ_1 or ρ_2.

The preprocessing step is illustrated in Algorithm 1 and creates a list of mappings L. The list initially contains all c^n possible mappings, *i.e.* all assignments of n variables x_i to one of c Sharing Vectors (1). We iterate over L (2). For each monomial m up to the target degree t (3), we check whether it is compatible with the mapping ρ, *i.e.* whether for any two variables in the monomial m they do not have the same Sharing Vector (5). After all compatible monomials for one mapping ρ have been determined, we check for a duplicate - another mapping $\hat{\rho}$ with an identical list of compatible monomials - and eliminate it. We also check whether the mapping ρ is compatible with at least one monomial of the target degree t and otherwise discard it (9,10). The runtime of the entire preprocessing step is bounded by $\mathcal{O}(2^n \cdot c^n)$.

Algorithm 2 demonstrates the search for an l-split of mappings for a specific target function F. Its run-time is $|L|^l = \mathcal{O}(c^{ln})$. In practice, the computation for our first-order

Algorithm 1 Preprocessing of mappings

```
Input: $n$: number of input bits; $t$: deg($F$); $c$: number of columns of $(t, d)$-Sharing Matrix
Output: $L$: list of mappings; $\alpha$: compatibility $\alpha_m^\rho$
 1: $L \leftarrow \{(\rho(0), \dots, \rho(n-1)) | \rho(i) \in \{0, \dots, c-1\}\}$
 2: for $\rho \in L$ do
 3:     for $m \in \mathrm{GF}(2^n)$ s.t. $\mathrm{hw}(m) \leq t$ do
 4:         $\alpha_m^\rho \leftarrow 0$
 5:         if $\rho(i) \neq \rho(j) \forall i \neq j$ s.t. $m_i = m_j = 1$ then
 6:             $\alpha_m^\rho \leftarrow 1$
 7:         end if
 8:     end for
 9:     if $\exists \hat{\rho} \in L$ s.t. $\mathrm{rep}(\hat{\rho}) = \mathrm{rep}(\rho)$ or $\max_{m, \alpha_m^\rho = 1} \mathrm{hw}(m) < t$ then
10:         $L \leftarrow L \setminus \{\rho\}$
11:     end if
12: end for
```

Algorithm 2 Search for a l-split

```
Input: $L$: list of mappings; $\alpha$: compatibility $\alpha_m^\rho$; $F$: target function
Output: $S$: a list of $l$-splits
 1: $S \leftarrow \emptyset$
 2: for $(\rho_1, \dots, \rho_l) \in L^l$ do
 3:     if $\mathrm{rep}(\rho_1) \mid \dots \mid \mathrm{rep}(\rho_l) \mid \mathrm{rep}(F) = \mathrm{rep}(\rho_1) \mid \dots \mid \mathrm{rep}(\rho_l)$ then
 4:         $S \leftarrow S \cup \{(\rho_1, \dots, \rho_l)\}$
 5:     end if
 6: end for
```

secure AES design with the parameters $c = t + 1 = 4$, $l = 2$, $n = 8$ takes $3.08s$ for Algorithm 1 and $5.73s$ for Algorithm 2 on a recent Desktop PC[5].

5 SCA-protected AES on FPGA

In this section, we apply our masking methodology from Section 5 to achieve a first-order secure FPGA-specific design of AES. We describe the structure of our design in detail, compare it to state-of-the-art implementations and demonstrate side channel resistance by practical measurements.

Rotational Symmetry. As noted in [Mor16, NNR18, WM18], the inversion in $\mathrm{GF}(2^8)$ has an algebraic degree of 7 but can be decomposed into two cubic bijections:

$$x^{-1} = x^{254} = (x^{26})^{49}$$

Since masking with $d+1$ shares for a function with degree t requires at least $(d+1)^t$ output shares [RBN+15], we choose to mask the cubic bijections x^{26} and x^{49} instead of realizing x^{-1} in one step. Moreover, since both components of the decomposition are power maps themselves, they can both be implemented using the rotation symmetry approach. Using the same method as before, we can thus find two Boolean functions F^* and G^* such that $F^*(\phi(x)) = \phi(x^{26})_0$ and $G^*(\phi(x)) = \phi(x^{49})_0$.

S-box Structure. We illustrate the structure of the decomposed shared S-box in Figure 6. Our purpose is to reuse as much hardware as possible to minimize the utilized FPGA resources. As before, a (shared) byte enters the circuit bit-serially via the input $\boldsymbol{x_i}$ and is saved to the upper shift register R1. Each byte share is then transformed to a normal basis representation using the affine mapping p2n. By rotation of R1, the power map x^{26}

[5] Averaged over 100 computations

is calculated bit by bit using a shared implementation of Boolean function F^*. The result is shifted bit-wise into the lower register R2 and when completed, the byte is written back into the upper register in parallel. There, it is rotated to calculate the power map x^{49} through shared Boolean function $\boldsymbol{G^*}$. When all eight 2-share bits have been calculated and shifted into the lower register, the resulting shares go through the final affine transform, which transforms back into polynomial basis and applies the AES affine function (n2p). The S-box output shares can be obtained bit by bit on wire $\boldsymbol{y_i}$.

The block $\boldsymbol{F^*/G^*}$ can compute either shared Boolean function $\boldsymbol{F^*}$ (corresponding to power map x^{26}) or Boolean function $\boldsymbol{G^*}$ (corresponding to power map x^{49}). Its functionality is determined by a control selection bit.

5.1 Implementation

Since our *fully bit-serialized* design (cf. Table 1; row 4) occupies the smallest area in LUTs and exhibits a lower latency than the byte-serial *with bit-serial S-box* design based on [SG16] (cf. Table 1; rows 3), we choose to mask this design rather than the byte-serialized architecture. In general, it may not be true that a smaller area footprint for an unprotected design results in a smaller footprint for the SCA-protected design, but the two designs in this case are only different in their linear components, for which the cost increase with SCA protection is linear. A similar reasoning holds for the latency.

$\boldsymbol{G^*/F^*}$. Figure 5 shows the masking of the non-linear block G^*/F^* in more detail. Note its significant optimization compared to Figure 5 in [DMW18]. A control bit *sel* chooses whether this block computes $\boldsymbol{G^*}$ or $\boldsymbol{F^*}$. We split each cubic function G^* and F^* into two parts $\left[G^A, G^B\right]$ and $\left[F^A, F^B\right]$ and share them according to the $(3,1)$-Sharing Matrix (4) and Equations (1) and (2).

Functions F^A, F^B, G^A and G^B were found using the algorithm described in Section 4.2 for all possible normal bases. For both F^* and G^*, we found that the minimum number of mappings needed for a split is two.

We combine G^A with F^A and let the control bit *sel* pick one of the two. We do the same with G^B and F^B. The possibility to incorporate the selection bit *sel* in the first stage of both parts A and B can be attributed to the fact that we performed the search for 2-splits of both functions F^* and G^* simultaneously. This minimizes the registers needed between the first and second stage considerably since each part creates immediately the minimum number of eight output shares. These results were found for a normal basis with $\beta = 205$. For the exact equations we refer to Appendix D.

Each individual output share (or register input) depends on one share of each input (*i.e.* 8 bits) and the control bit *sel*. As stated before, we only refresh the cross-domain shares. The six cross-domain shares thus depend on 10 variables in total and the shares z_0 and z_7 depend only on 9 variables. Since the number of LUTs can double for each additional input variable, a standard LUT mapping could require as much as 16 LUTs for the cross-domain shares and 8 LUTs for the other two shares. However, since F^A, F^B, G^A and G^B are only cubic functions, we were able to find a more optimal mapping manually. For block F^A/G^A, we can implement each cross-domain share with 7 LUTs and the inner-domain shares with 6 LUTs, resulting in a total cost of 54 LUTs. The second part of the split (F^B/G^B) has less monomials in the ANF and can be implemented with only 5 LUTs per share, which brings the total cost to 40 LUTs. The resulting 2×8 output shares are stored in a register to prevent propagation of glitches. Finally, the shares of the two blocks are compressed into $d+1 = 2$ shares y_0 and y_1 using two 8-bit XORs. Each of those can be implemented using 2 LUTs. In total, the entire circuit of $\boldsymbol{G^*/F^*}$ thus occupies 16 registers and $54{+}40{+}4 = 98$ LUTs and exhibits a latency of one clock cycle (due to the compression).

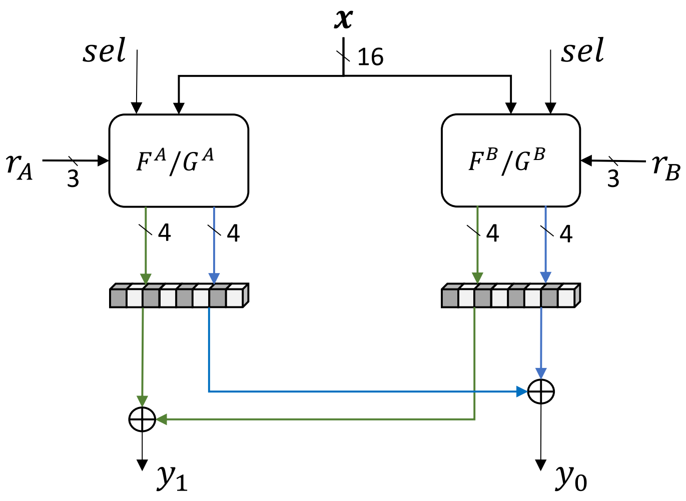


Figure 5: Illustration of the masked realization of the functions F^*/G^*.

Masked S-box. The masked S-box (Figure 6) has a latency of 26 cycles. In clock cycles 1 to 8, input $\boldsymbol{x}$ is shifted bit-serially into the upper register R1. In cycle 8, we also apply the affine transform p2n. The evaluation of $\boldsymbol{G^*}$ takes one clock cycle because of the register stage between expansion and compression of shares. We use the block as a pipeline, so the upper register R1 rotates continuously in clock cycles 9 to 16, feeding its content to $\boldsymbol{G^*}$ and the results are shifted bit-serially into R2 in clock cycles 10 to 17. The 7 most significant bits (in 2 shares) of the lower register R2 and the result of the last $\boldsymbol{G^*}$ computation are written to the upper register R1 in cycle 17 as well. Then, register R1 rotates again in cycles 18 to 25 and the results of $\boldsymbol{F^*}$ are shifted into R2 in clock cycles 19 to 26. The final affine transform is done in cycle 26. Result $\boldsymbol{y}$ can then be taken out bit-serially in 8 cycles, but this can be done in parallel with the loading of the next S-box input $\boldsymbol{x}$ into R1.

Vulnerability Potential. When R1 rotates, the input of $\boldsymbol{F^*}/\boldsymbol{G^*}$ instantly changes, and this may result in first-order leakage. As an example, consider $x_1x_2x_6$ as one of the terms in the ANF of G^B (see Appendix D). Let us denote the value of $(x_1, x_2, x_3, x_6, x_7)$ at one clock cycle by (a, b, c, d, e). Based on Equation (1), one of the eight terms in a 2-share realization is $z_2 = a_0b_1d_0$. In the next clock cycle, register R1 rotates and (x_1, x_2, x_6) have the values (b, c, e), hence the same circuit evaluates $z_2 = b_0c_1e_0$. This means that such a piece of circuit observes b_1 in one clock cycle, and b_0 in the next clock cycle. Hence, during the transition (positive edge of the clock) the leakage of the circuit can depend on both shares b_0 and b_1, hence breaking the non-completeness and inducing first-order leakage.

In order to avoid this issue, we pre-charge the input of $\boldsymbol{F^*}/\boldsymbol{G^*}$ before every shift in register R1. To this end, we employ an extra register at $\boldsymbol{F^*}/\boldsymbol{G^*}$'s input (see Figure 6), which is triggered at the *negative* edge of the clock, and reset (clear asynchronously) when clock is high. During the first half of the clock cycle (when clock is high) this pre-charge register clears the input of $\boldsymbol{F^*}/\boldsymbol{G^*}$. Once the clock changes to low, the value in R1 (already shifted) is stored in the register, hence given to $\boldsymbol{F^*}/\boldsymbol{G^*}$. At the next positive edge of the clock, R1 shifts and at the same time the pre-charge register is cleared, thereby pre-charging the $\boldsymbol{F^*}/\boldsymbol{G^*}$ input. This construction prevents any race between R1 being shifted and the pre-charge register being cleared. Even if R1 is shifted earlier (since its clock should have low skew) this transition does not pass through the pre-charge register, and $\boldsymbol{F^*}/\boldsymbol{G^*}$'s input stays unchanged.

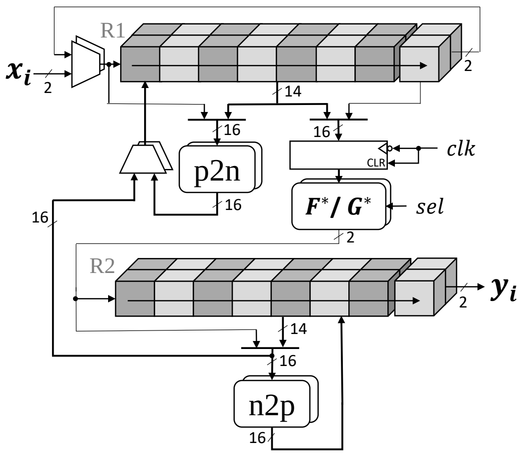


Figure 6: Illustration of the first-order secure AES S-box based on rotational symmetry when decomposed into x^{26} and x^{49}.

As a disadvantage, this construction can theoretically halve the maximum clock frequency. However, we have observed that $\boldsymbol{F^*/G^*}$ is not involved in the critical path of the circuit realizing the full AES encryption. Hence, the maximum clock frequency is not very much affected, and can even be maintained if the duty cycle of the clock is properly adjusted.

With respect to implementation, the $\boldsymbol{F^*/G^*}$ block requires 98 LUTs and 16 flip-flops. In addition, for each share we need 7 LUTs for both p2n and n2p, 1 LUT for the addition of the round key and 4 LUTs for the multiplexer that chooses the parallel input to R1. Each share also requires two 8-bit registers (R1 and R2) as well as one 8-bit register for the precharging of the $\boldsymbol{F^*/G^*}$ input. Therefore, our masked S-box can be implemented with $(98 + 2 \times (7 + 7 + 4 + 1)) = 136$ LUTs and $(16 + 2 \times (8 + 8 + 8)) = 64$ flip-flops. Further, the S-box has a fresh randomness cost of $2 \times 3 = 6$ bits per $\boldsymbol{F^*/G^*}$ evaluation, *i.e.* 6 bits per clock cycle. Each group of 3 bits is used in one part of the shared Boolean function as in Equation (2) (see Figure 5 with $r_i \in \mathrm{GF}(2)^3$).

Full-AES. We integrate the S-box into the same bit-serial AES design as used in Section 3. The state and key array and linear components of the AES cipher (MixColumns, AddRoundKey and ShiftRows) have simply been duplicated for each share separately. This results in occupying $23 \times 2 = 46$ LUTs and $4 \times 2 = 8$ registers. The latency of ShiftRows and MixColumns stays the same as for an unmasked design. When plugging in the masked S-box, we also need to adapt our control logic since the S-box latency has changed and we require an extra control signal to select $\boldsymbol{G^*}$ or $\boldsymbol{F^*}$. This new control unit uses 31 LUTs and 20 flip-flops. The design has a latency of 676 cycles per round with a shorter last round of 640 cycles. In total, with 128 cycles of loading, one encryption takes 6 852 cycles. The total footprint of our masked AES (post-map) is 92 flip-flops and 230 LUTs when the key schedule is masked and 220 LUTs when it is not.

Results. It is difficult to compare these results to state-of-the-art masked AES implementations [BGN+15, CRB+16, GMK17, UHA17b] since they target an ASIC platform. We can let Xilinx map these designs to Spartan-6 resources, but unlike our design, they have not been optimized specifically for this purpose. In Table 3, we do this first for various masked S-box implementations. The results from other works are obtained by synthesis, translate and map using Xilinx default settings apart from the `KEEP HIERARCHY` constraint which is turned on to prohibit optimization across shares [RBG+15], as is common practice with masked implementations [De 18, §2.4.1]. We stress that no optimization for FPGA has been done for these designs. When comparing these results to the ASIC numbers reported in the original works, the stark contrast between the worlds of ASICs and FPGAs is clearly confirmed. Moreover, the FPGA footprint is strongly influenced by the coding style of the creators (*e.g.* extent of hierarchy use, clock gating vs. clock enabling, . . .), which is obviously different for each of the designs. We also see clearly the advantage of the new sharing method for the Boolean function G^*/F^* compared to [DMW18], both in resource requirements and randomness consumption.

We should emphasize that all the considered designs are expected to provide only first-order security with minimum number of shares for the state and key arrays. The random bits, which we report in Table 3, are corresponding to the number of fresh random bits required at each clock cycle. Since the other designs have a (pipelined) byte-serial S-box, the number of required fresh masks per clock cycle is the same as those required for every S-box evaluation. However, since in our design the S-box is bit-serial and does not form a pipeline, the number of required fresh masks per S-box invocation is different.

We further report the same performance figures for the corresponding full AES encryption-only implementations in Table 4.[6] Note that for all these designs, both the state and key arrays are shared.

Table 3: Comparison of first-order secure AES S-boxes, mapped for Spartan-6.

Design	**# LUTs**	**# FFs**	**# Slices**	**# Random bits**
Bilgin *et al.* [BGN+15]	361	92	177	32
Gross *et al.* [GMK17]	327	208	242	18
Cnudde *et al.* [CRB+16]	340	144	283	54
Ueno *et al.* [UHA17b]	302	96	218	64
[DMW18]	182	96	95	18
This work	144	64	67	6

Table 4: Comparison of first-order secure AES implementations, mapped for Spartan-6.

Design	**# LUTs**	**# FFs**	**# Slices**	**# CCs***	$\mathbf{f_{max}}^\dagger$
Bilgin *et al.* (nimble) [BGN+15]	1198	611	475	246	127 MHz
Gross *et al.* [GMK17]	595	734	366	246	103 MHz
Cnudde *et al.* [CRB+16]	1191	642	553	276	181 MHz
[DMW18]	293	124	162	6852	103 MHz
This work	230	92	108	6852	120 MHz

$*$ Number of clock cycles
† From the Post-PAR Static Timing Report

A note about block RAM. As stated in Section 3.3, we have intentionally avoided the utilization of any BRAMs in our constructions. As a side note, if a BRAM is supposed to be used in a masked implemented, its inputs must fulfill the non-completeness property [NRS11]. Therefore, we would require 8 such distinct BRAM instances, that – as formerly stated – would result in wasting their available storage space.

5.2 SCA Evaluation

Measurement Setup. For practical evaluations, we implement our full AES encryption design on the target Spartan-6 FPGA of the SAKURA-G platform [sak], a commonly known and employed board for SCA evaluations. By means of a digital oscilloscope at a

[6]We do not have access to the design of the full AES implementation of [UHA17b].

sampling rate of 625 MS/s, we measure the power consumption of the target FPGA, which is clocked at a frequency of 6 MHz, through the dedicated on-board AC amplifier. Due to the very low power consumption of our design (particularly since the state and key arrays are stored in shift register LUTs), we additionally employ an AC amplifier[7] with 10 dB gain. During the measurements, the masked AES core receives the shared plaintext and the shared key and sends back the shared ciphertext.

Each of the required 18-bit fresh masks are provided by a dedicated 31-bit LFSR with the feedback polynomial $x^{31} + x^{28} + 1$. Such an LFSR has a maximum cycle $2^{31} - 1$ with only two taps [WM12], hence should suffice for more than 2 billion measurements. Each LFSR is implemented by means of only 3 LUTs, of which two are employed as shift register and the last one to make the feedback signal, *i.e.* the entire fresh mask generation is realized in $18 \times 3 = 54$ LUTs. We arbitrarily initialize the LFSRs (not null) right after the FPGA power-up. They are supplied with the same clock as the masked AES core, but operate on the negative edge of the clock. This is done to reduce the effect of the LFSR transitions on the SCA measurements associated to the masked AES core [CRB+16].

Evaluation. Most of the related state-of-the-art schemes evaluate the masked design by means of fixed-versus-random t-test [GJJR11, CDG+13, SM15]. It has recently been shown that such evaluations on masked hardware with only 2 shares can yield misleading results [CEM18]. In other words, when the measurement noise is low, such a t-test may always show detectable leakage independent of the implementation and the underlying masking scheme. Since our design is also prone to this issue due to its very low resource requirements, we conduct attacks instead of such leakage assessment techniques. To this end, in order to relax the necessity of having a detailed and accurate power consumption model, we decide to perform Moments-Correlating DPA [MS16] (MC-DPA) which is a more robust and theoretically more accurate form of Correlation-Enhanced Collision Attack [MME10]. In short, we perform first- and second-order collision Moment-Correlation DPA attacks by considering the leakage of one S-box evaluation as the model and thereby performing the attack on another S-box evaluation. It is noteworthy that such linear collision attacks recover the linear difference between the associated keys [Bog08].

PRNG OFF. We first turn off the LFSR PRNG (for the fresh masks) as well as the initial masking of the plaintext and key to emulate an unprotected implementation. The sample trace shown in Figure 7a covers eight S-box evaluations of the first encryption round (indeed of the first two state rows). We also present the signal-to-noise ratio (SNR) curves estimated based on the value of the plaintext bytes in Figure 7b. To this end, we follow the procedure explained in [MOP07]. The SNR curves show a clear dependency on the plaintext bytes, and hence the S-box inputs. Using 10 000 traces and considering the leakage of the second S-box evaluation (of state byte no. 4) as the model, we conduct a first-order MC-DPA on the third S-box (of state byte no. 8), which yields the correlation curves shown in Figure 7c. The results indicate that very few traces are required to correctly identify the difference between the corresponding key bytes. We further repeat the same experiment for two other cases: (a) LFSR PRNG on and initial masking off, (b) LFSR PRNG off and initial masking on. For both cases we again observe clearly-distinguishable SNR curves (although with lower amplitude, *i.e.* 0.02 compared to 13 in Figure 7b). The same MC-DPA attacks also successfully recover the correct key difference using at most 100 000 traces.

PRNG ON. When both the LFSR PRNG and initial masking are active, we collect 10 000 000 traces, each covering only the above-selected two S-box evaluations[8]. Following the same scenario as in the case PRNG off, we perform both first-order and second-order

[7]ZFL-1000LN+ from Mini-Circuits

[8]Due to the high latency of the entire encryption, the measurement process is relatively slow. We also have to cover at least two S-box evaluations (for collision MC-DPA) leading to long power traces. This limited our analysis with respect to the number of collected traces.

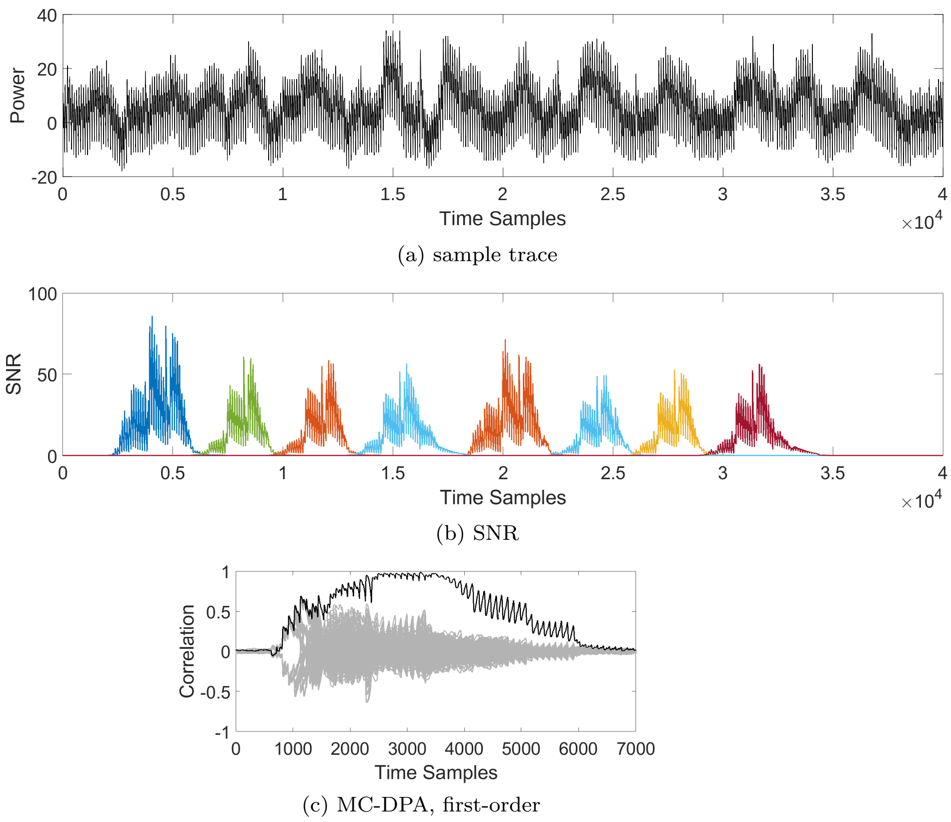
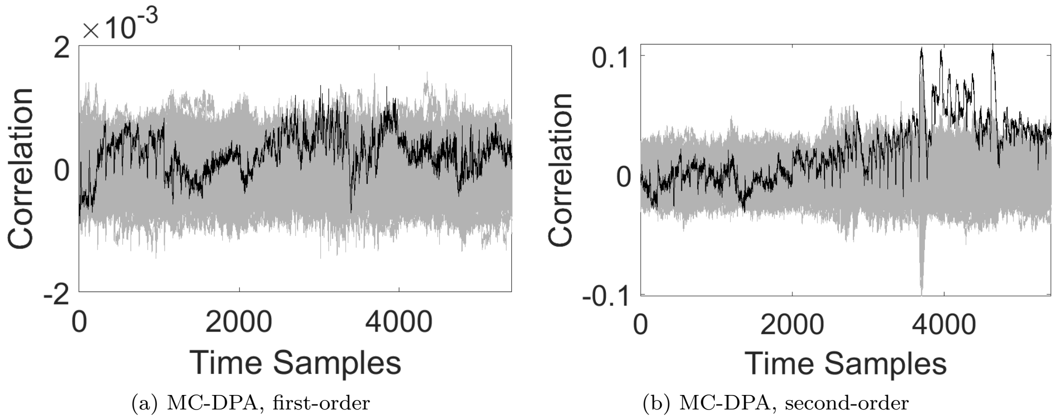


Figure 7: PRNG and initial masking disabled, 10 000 traces, (a) sample trace, (b) SNR curves based on 8 plaintext bytes with the order from left to right: byte no. 0, 4, 8, 12, 1, 5, 9, 13, (c) first-order Moments-Correlating DPA result targeting S-box no. 8 with model S-box no. 4, the black curve belonging to the correct key difference.

MC-DPA attacks. The corresponding results are shown in Figure 8 and show clearly that the countermeasure is effective at providing protection against first-order side-channel analysis. On the other hand, a second-order attack does succeed, as can be expected. This confirms that our measurement setup is sound.

5.3 Discussion

Higher-Order Resistance. It is noticeable in Figure 8b that the second-order attack succeeds with very low number of, *e.g.* 10 000 traces. This is due to two facts: (a) masking with minimum number of two shares has in general a strong vulnerability to second-order attacks [CFE16], (b) higher-order attacks are sensitive to the noise level [PRB09] and our design (due to its extremely low resource utilization) has a very low switching noise particularly when the masked S-box is evaluated the entire circuit stops till the termination of the S-box. Hence, the S-box is the sole source of leakage at that time. Further, our utilized LFSR PRNG (again using shift register LUTs) does not add a remarkable amount of noise to the measurements. The number of traces required to successfully perform a second-order attack is expected to rapidly grow with decreasing the SNR, since accurately estimating higher-order statistical moments requires a larger amounts of samples compared to lower-order moments in presence of noise [PRB09]. Our first-order secure implementation should therefore be combined with hiding countermeasures, such as random shuffling and



(a) MC-DPA, first-order (b) MC-DPA, second-order

Figure 8: PRNG and initial masking enabled, Moments-Correlating DPA result targeting S-box no. 8 with model S-box no. 4, (a) first-order with 10 000 000 traces, (b) second-order with 10 000 traces.

noise modules. As an example we refer to [EGMP17], where the design of such a noise generator on the same FPGA type is given. A combination of lowering the SNR and restricting the number of encryptions performed with the same key should be able to avoid higher-order attacks in practice.

Design Transfer. Our design is directly transferable to more modern Xilinx devices of the 7 Series as they contain the same general architecture. Most notably, the Spartan 7 can feature as little as 938 slices. In fact, we transfered our first-order protected design onto the smallest Spartan 7 device. Here it occupies 209 LUTs and 92 flip flops in 84 slices at a frequency of 118 MHz - a slight improvement over the Spartan 6 results. The reduction in the number of occupied slices can be attributed to the usage of Vivado 2018.3 to synthesize, place and route our design, which contains many algorithmic improvements over the older ISE 14.7 software used in Section 5.1. Transferring our design to a different vendor would be a time consuming process as all Xilinx-specific primitives need to be remapped. However, on a conceptional level the transfer is possible whenever 6-input LUTs are available. This allows a transfer to ALTERA FPGAs based on adaptive logic modules (ALM). On the other hand, MicroSemi and Lattice devices which utilize 4-input LUTs cannot directly benefit from our design, but our methodlogy still applies. Obviously, each vendor-specific FPGA structure might allow other custom optimizations not discussed here.

Real-World Applications. Our implementations target very low area at the cost of latency. Since area is considered relatively cheap with recent technologies, our design may not be of interest for just any application. However, there are also many use cases where low area and low power consumption are very important and low throughput is acceptable, for example in the Internet of Things. Applications include remote measurement and smart metering, especially when powered by solar energy. Also car key fobs are an excellent use case example. The need for side channel protection was shown by the Keeloq attacks in [EKM+08]. Moreover, whenever reconfigurability of the product after shipment might be necessary, an FPGA can be used instead of an ASIC and our designs are applicable. The importance of such a feature was recently demonstrated by Tesla, when they updated their key fobs after the attack from [WMA+19].

6 Conclusion

Our contribution is manifold. First, we made several FPGA-specific AES implementations which compromise between the latency and area requirements. We improved the latency of the formerly smallest known AES on Xilinx FPGAs [SG16]. Furthermore, we achieved a new size record by replacing its S-box with our bit-serial rotational design fitting into only 17 slices, while the former record by Sasdrich *et al.* [SG16] requires 21 slices - a 19% size reduction. This can be fully attributed to cutting the size of the S-box by half from 8 slices to 4.

Second, with respect to masking as an SCA countermeasure, we developed an effective heuristic to find sharings of any Boolean function with $d+1$ shares by splitting its ANF into a minimum number of sub-components, each of which can be shared with a Sharing Matrix.

Third, we applied our heuristic to our AES S-box construction to obtain an FPGA-specific masked AES. We further reduce the area overhead by exploiting the rotational symmetry of a cubic decomposition of the inversion in $GF(2^8)$. Our first-order secure AES S-box requires only 144 LUTS, while the masked AES encryption requires 230 LUTs - a new area record on FPGAs. However, we should emphasize that such low area footprints come at the cost of high latency. More precisely, our designs are suitable for applications with no high throughput needs. Moreover, the byte-serial AES designs we compare to, have not yet been optimized for FPGA-specific implementations. This remains an interesting directon for future work. To promote further research as well as for comparison purposes, the HDL code of our implementations is publicly available online[9].

Acknowledgments

The work described in this paper has been supported in part by the German Federal Ministry of Education and Research BMBF (grant nr. 16KIS0666 SysKit_HW) and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Lauren De Meyer is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO).

References

[BB16] Tim Beyne and Begül Bilgin. Uniform first-order threshold implementations. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2016.

[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.

[9]https://github.com/emsec/RotationalSymmetry

[BDGH15] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Wei He. Exploiting FPGA block memories for protected cryptographic implementations. *TRETS*, 8(3):16:1–16:16, 2015.

[BGD12] Shivam Bhasin, Sylvain Guilley, and Jean-Luc Danger. From cryptography to hardware: Analyzing embedded xilinx BRAM for cryptographic applications. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Workshops Proceedings, Vancouver, BC, Canada, December 1-5, 2012*, pages 1–8. IEEE Computer Society, 2012.

[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.

[BGS$^+$11] Shivam Bhasin, Sylvain Guilley, Youssef Souissi, Tarik Graba, and Jean-Luc Danger. Efficient dual-rail implementations in FPGA using block rams. In Peter M. Athanas, Jürgen Becker, and René Cumplido, editors, *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011*, pages 261–267. IEEE Computer Society, 2011.

[BH12] Andries Brouwer and Willem Haemers. *Spectra of Graphs*, chapter Chapter 12: Distance-Regular Graphs, page 178. Springer New York, 2012.

[BKN18] Dusan Bozilov, Miroslav Knezevic, and Ventzislav Nikov. Optimized threshold implementations: Securing cryptographic accelerators for low-energy and low-latency applications. *IACR Cryptology ePrint Archive*, 2018:922, 2018.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *J. Cryptology*, 26(2):280–312, 2013.

[BNN$^+$12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3x3 and 4x4 s-boxes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012.

[Bog08] Andrey Bogdanov. Multiple-differential side-channel collision attacks on AES. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2008.

[BSQ$^+$08] Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2008.

[Can05] David Canright. A very compact s-box for AES. In Rao and Sunar [RS05], pages 441–455.

[CB08] David Canright and Lejla Batina. A very compact "perfectly masked" S-Box for AES. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, 6th*

International Conference, ACNS 2008, New York, NY, USA, June 3-6, 2008. Proceedings, volume 5037 of *Lecture Notes in Computer Science*, pages 446–459, 2008.

[CB12] Junfeng Chu and Mohammed Benaissa. Low area memory-free FPGA implementation of the AES algorithm. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 623–626. IEEE, 2012.

[CDG+13] Jeremy Cooper, Elke De Mulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, and Pankaj Rohatgi. Test Vector Leakage Assessment (TVLA) Methodology in Practice. International Cryptographic Module Conference, 2013.

[CEM18] Thomas De Cnudde, Maik Ender, and Amir Moradi. Hardware masking, revisited. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2), 2018. to appear.

[CFE16] Cong Chen, Mohammad Farmani, and Thomas Eisenbarth. A tale of two shares: Why two-share threshold implementation seems worthwhile - and why it is not. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 819–843, 2016.

[CG03] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Wiener [Wie99], pages 398–412.

[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.

[De 18] Thomas De Cnudde. *Cryptography Secured against Side-Channel Attacks*. PhD thesis, KU Leuven, 2018.

[DMW18] Lauren De Meyer, Amir Moradi, and Felix Wegener. Spin me right round: Rotational symmetry for FPGA-specific AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3), 2018.

[EGMP17] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The first thorough side-channel hardware trojan. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 755–780. Springer, 2017.

[EKM+08] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the keeloqcode hopping scheme. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2008.

[GH15] Tim Güneysu and Helena Handschuh, editors. *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*. Springer, 2015.

[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.

[GKN+08] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Trans. Computers*, 57(11):1498–1513, 2008.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2017.

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

[JMPS17] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. Bit-sliding: A generic technique for bit-serial implementations of SPN-based primitives - applications to AES, PRESENT and SKINNY. 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 687–707. Springer, 2017.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener [Wie99], pages 388–397.

[MBPV05] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-Box. 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 323–333. Springer, 2005.

[MM12] Amir Moradi and Oliver Mischke. Glitch-free implementation of masking in modern fpgas. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2012, San Francisco, CA, USA, June 3-4, 2012*, pages 89–95. IEEE, 2012.

[MME10] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-enhanced power analysis collision attack. In Stefan Mangard and François-Xavier Standaert, editors, *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 125–139. Springer, 2010.

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

[Mor16] Amir Moradi. Advances in Side-channel Security, 2016.

[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 Rao and Sunar [RS05], pages 157–171.

[MS16] Amir Moradi and François-Xavier Standaert. Moments-correlating DPA. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, pages 5–15. ACM, 2016.

[MW15] Amir Moradi and Alexander Wild. Assessment of Hiding the Higher-Order Leakages in Hardware - What Are the Achievements Versus Overheads? In Güneysu and Handschuh [GH15], pages 453–474.

[NBD$^+$10] Maxime Nassar, Shivam Bhasin, Jean-Luc Danger, Guillaume Duc, and Sylvain Guilley. BCDL: A high speed balanced DPL for FPGA with global precharge and no early evaluation. In Giovanni De Micheli, Bashir M. Al-Hashimi, Wolfgang Müller, and Enrico Macii, editors, *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 849–854. IEEE Computer Society, 2010.

[NNR18] Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Decomposition of permutations in a finite field. *IACR Cryptology ePrint Archive*, 2018:103, 2018.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *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.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.

[PRB09] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009.

[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.

[RBG$^+$15] Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Jean-Luc Danger, and Debdeep Mukhopadhyay. From theory to practice of private circuit: A cautionary note. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, pages 296–303. IEEE Computer Society, 2015.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew Robshaw, editors, *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.

[RS05] 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*. Springer, 2005.

[RWS11] Francesco Regazzoni, Yi Wang, and François-Xavier Standaert. Fpga implementations of the aes masked against power analysis attacks. In *Constructive Side-Channel Analysis and Secure Design - 2nd International Workshop, COSADE 2011, Darmstadt, Germany, February 24-25, 2011. Revised Selected Papers*, pages 56–66, 2011.

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

[SG16] Pascal Sasdrich and Tim Güneysu. A grain in the silicon: SCA-protected AES in less than 30 slices. In *27th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2016, London, United Kingdom, July 6-8, 2016*, pages 25–32. IEEE Computer Society, 2016.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Güneysu and Handschuh [GH15], pages 495–513.

[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 Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, volume 9064 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2015.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with s-box optimization. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[Tri03] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.

[UHA17a] Rei Ueno, Naofumi Homma, and Takafumi Aoki. A systematic design of tamper-resistant galois-field arithmetic circuits based on threshold implementation with (d + 1) input shares. In *47th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2017, Novi Sad, Serbia, May 22-24, 2017*, pages 136–141. IEEE Computer Society, 2017.

[UHA17b] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward more efficient dpa-resistant AES hardware architecture based on threshold implementation. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2017.

[VRM17] Jo Vliegen, Oscar Reparaz, and Nele Mentens. Maximizing the throughput of threshold-protected AES-GCM implementations on FPGA. In *IEEE 2nd International Verification and Security Workshop, IVSW 2017, Thessaloniki, Greece, July 3-5, 2017*, pages 140–145. IEEE, 2017.

[Wam14] Markus Stefan Wamser. Ultra-small designs for inversion-based S-Boxes. In *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*, pages 512–519. IEEE Computer Society, 2014.

[WHS15] Markus Stefan Wamser, Lukas Holzbaur, and Georg Sigl. A petite and power saving design for the AES S-Box. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 661–667. IEEE Computer Society, 2015.

[Wie99] 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*. Springer, 1999.

[WM12] Roy Ward and Timothy C.A. Molteno. Table of linear feedback shift registers. Technical Report 2012-1, University of Otago, 2012. http://www.physics.otago.ac.nz/reports/electronics/ETR2012-1.pdf.

[WM18] Felix Wegener and Amir Moradi. A first-order SCA resistant AES without fresh randomness. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-25, 2018*, 2018.

[WMA$^+$19] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):66–85, 2019.

[WS17] Markus Stefan Wamser and Georg Sigl. Pushing the limits further: Sub-atomic AES. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2017, Abu Dhabi, United Arab Emirates, October 23-25, 2017*, pages 1–6. IEEE, 2017.

[Xil10] Xilinx. Spartan-6 FPGA configurable logic block user guide. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf, 2010.

A ANFs for Byte-Serial Unprotected S-box

The following results are valid in a normal basis with $\beta = 145$. To allow replication of our results we share S^* both as ANF and in a machine-readable notation (*i.e.* the 256-bit vector).

$$\begin{aligned}
\mathrm{rep}(S^*) &= \texttt{0x1c14813636f5767d6abc937b490334efd066cb1449f7ad147f30286c8bbef414} \\
S^*(x) &= x_1 \oplus x_2 \oplus x_1x_3 \oplus x_2x_3 \oplus x_0x_2x_3 \oplus x_1x_2x_3 \oplus x_0x_1x_2x_3 \oplus x_0x_4 \oplus x_1x_4 \oplus x_0x_1x_4 \\
&\oplus x_2x_4 \oplus x_0x_2x_4 \oplus x_0x_1x_2x_4 \oplus x_3x_4 \oplus x_0x_3x_4 \oplus x_0x_1x_3x_4 \oplus x_0x_1x_2x_3x_4 \oplus x_1x_5 \\
&\oplus x_0x_1x_5 \oplus x_0x_2x_5 \oplus x_1x_2x_5 \oplus x_0x_1x_3x_5 \oplus x_0x_2x_3x_5 \oplus x_2x_4x_5 \oplus x_0x_2x_4x_5 \\
&\oplus x_3x_4x_5 \oplus x_0x_3x_4x_5 \oplus x_1x_3x_4x_5 \oplus x_0x_1x_3x_4x_5 \oplus x_2x_3x_4x_5 \oplus x_0x_2x_3x_4x_5 \\
&\oplus x_1x_2x_3x_4x_5 \oplus x_1x_6 \oplus x_2x_6 \oplus x_3x_6 \oplus x_1x_3x_6 \oplus x_0x_1x_3x_6 \oplus x_0x_2x_3x_6 \\
&\oplus x_0x_1x_2x_3x_6 \oplus x_4x_6 \oplus x_0x_4x_6 \oplus x_1x_4x_6 \oplus x_2x_4x_6 \oplus x_0x_2x_4x_6 \oplus x_1x_2x_4x_6 \\
&\oplus x_0x_1x_2x_4x_6 \oplus x_3x_4x_6 \oplus x_0x_1x_3x_4x_6 \oplus x_1x_2x_3x_4x_6 \oplus x_1x_5x_6 \oplus x_2x_5x_6 \oplus x_3x_5x_6 \\
&\oplus x_0x_3x_5x_6 \oplus x_0x_1x_3x_5x_6 \oplus x_1x_2x_3x_5x_6 \oplus x_0x_1x_2x_3x_5x_6 \oplus x_0x_4x_5x_6 \oplus x_1x_4x_5x_6 \\
&\oplus x_0x_2x_4x_5x_6 \oplus x_1x_2x_4x_5x_6 \oplus x_2x_3x_4x_5x_6 \oplus x_1x_2x_3x_4x_5x_6 \oplus x_0x_1x_2x_3x_4x_5x_6 \oplus x_7 \\
&\oplus x_0x_7 \oplus x_1x_7 \oplus x_0x_1x_7 \oplus x_0x_2x_7 \oplus x_1x_2x_7 \oplus x_0x_1x_2x_7 \oplus x_1x_3x_7 \oplus x_2x_3x_7 \\
&\oplus x_0x_2x_3x_7 \oplus x_4x_7 \oplus x_0x_4x_7 \oplus x_3x_4x_7 \oplus x_0x_1x_3x_4x_7 \oplus x_1x_2x_3x_4x_7 \oplus x_5x_7 \oplus x_0x_5x_7 \\
&\oplus x_0x_1x_5x_7 \oplus x_2x_5x_7 \oplus x_0x_2x_5x_7 \oplus x_1x_2x_5x_7 \oplus x_3x_5x_7 \oplus x_0x_3x_5x_7 \oplus x_2x_3x_5x_7 \\
&\oplus x_0x_1x_2x_3x_5x_7 \oplus x_1x_4x_5x_7 \oplus x_0x_1x_4x_5x_7 \oplus x_2x_4x_5x_7 \oplus x_0x_2x_4x_5x_7 \oplus x_0x_1x_2x_4x_5x_7 \\
&\oplus x_0x_3x_4x_5x_7 \oplus x_0x_1x_3x_4x_5x_7 \oplus x_0x_2x_3x_4x_5x_7 \oplus x_1x_2x_3x_4x_5x_7 \oplus x_6x_7 \oplus x_1x_6x_7 \\
&\oplus x_0x_1x_6x_7 \oplus x_2x_6x_7 \oplus x_0x_2x_6x_7 \oplus x_1x_2x_6x_7 \oplus x_0x_3x_6x_7 \oplus x_1x_3x_6x_7 \oplus x_2x_3x_6x_7 \\
&\oplus x_0x_2x_3x_6x_7 \oplus x_1x_2x_3x_6x_7 \oplus x_4x_6x_7 \oplus x_1x_4x_6x_7 \oplus x_2x_4x_6x_7 \oplus x_0x_2x_4x_6x_7 \\
&\oplus x_1x_2x_4x_6x_7 \oplus x_0x_1x_2x_4x_6x_7 \oplus x_0x_3x_4x_6x_7 \oplus x_1x_3x_4x_6x_7 \oplus x_2x_3x_4x_6x_7 \\
&\oplus x_0x_2x_3x_4x_6x_7 \oplus x_0x_5x_6x_7 \oplus x_1x_5x_6x_7 \oplus x_2x_5x_6x_7 \oplus x_0x_2x_5x_6x_7 \oplus x_3x_5x_6x_7 \\
&\oplus x_0x_1x_2x_3x_5x_6x_7 \oplus x_1x_4x_5x_6x_7 \oplus x_2x_4x_5x_6x_7 \oplus x_1x_3x_4x_5x_6x_7 \oplus x_0x_1x_3x_4x_5x_6x_7 \\
&\oplus x_2x_3x_4x_5x_6x_7
\end{aligned}$$

Furthermore, we provide the equations for the conversion from a polynomial base of $\mathrm{GF}(2^8)$ with $\alpha = 2$ to a normal base with $\beta = 145$ (p2n) and the conversion back concatenated with the affine function of the AES S-box (n2p).

$$\begin{aligned}
\mathrm{p2n}_0(x) &= x_0 \oplus x_1 \oplus x_3 \oplus x_6 \\
\mathrm{p2n}_1(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_6 \oplus x_7 \\
\mathrm{p2n}_2(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
\mathrm{p2n}_3(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus x_6 \\
\mathrm{p2n}_4(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus x_5 \\
\mathrm{p2n}_5(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \\
\mathrm{p2n}_6(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \\
\mathrm{p2n}_7(x) &= x_0 \oplus x_3 \oplus x_4 \oplus x_6
\end{aligned}$$

$$\begin{aligned}
\mathrm{n2p}_0(x) &= x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \oplus 1 \\
\mathrm{n2p}_1(x) &= x_1 \oplus x_3 \oplus x_4 \oplus x_6 \oplus x_7 \oplus 1 \\
\mathrm{n2p}_2(x) &= x_1
\end{aligned}$$

$$\begin{aligned}
\mathrm{n2p}_3(x) &= x_1 \oplus x_2 \oplus x_3 \\
\mathrm{n2p}_4(x) &= x_2 \\
\mathrm{n2p}_5(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus 1 \\
\mathrm{n2p}_6(x) &= x_0 \oplus x_4 \oplus 1 \\
\mathrm{n2p}_7(x) &= x_1 \oplus x_2 \oplus x_4 \oplus x_5
\end{aligned}$$

B ANFs for Bit-Serial Unprotected S-box

The following results are valid in a normal basis with $\beta = 133$. To allow replication of our results we share S^* both as ANF and in a machine-readable notation (*i.e.* the 256-bit vector).

$$\begin{aligned}
\mathrm{rep}(S^*) &= \texttt{0x70355d75860553518544703c10a90ad5ef30c359047bf6e4cccce9c4635703a8} \\
S^*(x) &= x_0x_1 \oplus x_0x_2 \oplus x_0x_1x_2 \oplus x_3 \oplus x_0x_3 \oplus x_4 \oplus x_0x_4 \oplus x_1x_4 \oplus x_2x_4 \oplus x_1x_2x_4 \oplus x_3x_4 \\
&\oplus x_0x_3x_4 \oplus x_0x_2x_3x_4 \oplus x_1x_2x_3x_4 \oplus x_1x_5 \oplus x_1x_2x_5 \oplus x_0x_1x_2x_5 \oplus x_3x_5 \oplus x_0x_1x_3x_5 \\
&\oplus x_0x_2x_3x_5 \oplus x_1x_2x_3x_5 \oplus x_0x_1x_2x_3x_5 \oplus x_1x_4x_5 \oplus x_0x_1x_4x_5 \oplus x_1x_2x_4x_5 \oplus x_0x_1x_2x_4x_5 \\
&\oplus x_1x_3x_4x_5 \oplus x_0x_1x_3x_4x_5 \oplus x_1x_2x_3x_4x_5 \oplus x_0x_1x_2x_3x_4x_5 \oplus x_1x_6 \oplus x_0x_2x_6 \oplus x_1x_2x_6 \\
&\oplus x_0x_1x_2x_6 \oplus x_0x_3x_6 \oplus x_1x_3x_6 \oplus x_2x_3x_6 \oplus x_0x_2x_3x_6 \oplus x_1x_2x_3x_6 \oplus x_0x_1x_2x_3x_6 \oplus x_4x_6 \\
&\oplus x_0x_4x_6 \oplus x_0x_1x_4x_6 \oplus x_2x_4x_6 \oplus x_0x_2x_4x_6 \oplus x_1x_2x_4x_6 \oplus x_1x_3x_4x_6 \oplus x_5x_6 \oplus x_0x_1x_5x_6 \\
&\oplus x_2x_5x_6 \oplus x_1x_2x_5x_6 \oplus x_3x_5x_6 \oplus x_0x_3x_5x_6 \oplus x_1x_2x_3x_5x_6 \oplus x_0x_1x_2x_3x_5x_6 \oplus x_2x_4x_5x_6 \\
&\oplus x_0x_2x_4x_5x_6 \oplus x_3x_4x_5x_6 \oplus x_0x_3x_4x_5x_6 \oplus x_1x_3x_4x_5x_6 \oplus x_0x_1x_3x_4x_5x_6 \oplus x_0x_2x_3x_4x_5x_6 \\
&\oplus x_1x_2x_3x_4x_5x_6 \oplus x_0x_1x_2x_3x_4x_5x_6 \oplus x_7 \oplus x_1x_7 \oplus x_2x_7 \oplus x_1x_2x_7 \oplus x_0x_1x_2x_7 \oplus x_0x_3x_7 \\
&\oplus x_0x_1x_3x_7 \oplus x_4x_7 \oplus x_0x_1x_4x_7 \oplus x_0x_2x_4x_7 \oplus x_0x_1x_2x_4x_7 \oplus x_2x_3x_4x_7 \oplus x_1x_5x_7 \\
&\oplus x_0x_1x_5x_7 \oplus x_2x_5x_7 \oplus x_0x_2x_5x_7 \oplus x_2x_3x_5x_7 \oplus x_0x_2x_3x_5x_7 \oplus x_1x_2x_3x_5x_7 \oplus x_1x_4x_5x_7 \\
&\oplus x_1x_2x_4x_5x_7 \oplus x_3x_4x_5x_7 \oplus x_1x_3x_4x_5x_7 \oplus x_0x_1x_2x_3x_4x_5x_7 \oplus x_6x_7 \oplus x_2x_6x_7 \oplus x_1x_2x_6x_7 \\
&\oplus x_3x_6x_7 \oplus x_0x_3x_6x_7 \oplus x_2x_3x_6x_7 \oplus x_1x_2x_3x_6x_7 \oplus x_4x_6x_7 \oplus x_1x_4x_6x_7 \oplus x_0x_3x_4x_6x_7 \\
&\oplus x_1x_3x_4x_6x_7 \oplus x_0x_1x_2x_3x_4x_6x_7 \oplus x_5x_6x_7 \oplus x_1x_5x_6x_7 \oplus x_2x_5x_6x_7 \oplus x_0x_2x_5x_6x_7 \\
&\oplus x_1x_2x_5x_6x_7 \oplus x_3x_5x_6x_7 \oplus x_1x_3x_5x_6x_7 \oplus x_0x_1x_3x_5x_6x_7 \oplus x_2x_3x_5x_6x_7 \oplus x_1x_2x_3x_5x_6x_7 \\
&\oplus x_4x_5x_6x_7 \oplus x_1x_4x_5x_6x_7 \oplus x_2x_4x_5x_6x_7 \oplus x_0x_2x_4x_5x_6x_7 \oplus x_2x_3x_4x_5x_6x_7 \\
&\oplus x_0x_2x_3x_4x_5x_6x_7 \oplus x_1x_2x_3x_4x_5x_6x_7
\end{aligned}$$

Furthermore, we provide the equations for the conversion from a polynomial base of $\mathrm{GF}(2^8)$ with $\alpha = 2$ to a normal base with $\beta = 133$ (p2n) and the conversion back concatenated with the affine function of the AES S-box (n2p).

$$\begin{aligned}
\mathrm{p2n}_0(x) &= x_0 \oplus x_3 \\
\mathrm{p2n}_1(x) &= x_0 \oplus x_1 \oplus x_5 \oplus x_6 \oplus x_7 \\
\mathrm{p2n}_2(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_5 \\
\mathrm{p2n}_3(x) &= x_0 \oplus x_4 \oplus x_6 \oplus x_7 \\
\mathrm{p2n}_4(x) &= x_0 \oplus x_1 \oplus x_3 \oplus x_5 \oplus x_7 \\
\mathrm{p2n}_5(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_6 \\
\mathrm{p2n}_6(x) &= x_0 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \\
\mathrm{p2n}_7(x) &= x_0 \oplus x_5 \oplus x_7
\end{aligned}$$

$$\begin{aligned}
\mathrm{n2p}_0(x) &= x_6 \oplus 1\\
\mathrm{n2p}_1(x) &= x_1 \oplus 1\\
\mathrm{n2p}_2(x) &= x_0 \oplus x_1 \oplus x_2\\
\mathrm{n2p}_3(x) &= x_0 \oplus x_2 \oplus x_4\\
\mathrm{n2p}_4(x) &= x_1 \oplus x_2 \oplus x_3\\
\mathrm{n2p}_5(x) &= x_0 \oplus x_3 \oplus x_4 \oplus x_5 \oplus 1\\
\mathrm{n2p}_6(x) &= x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_7 \oplus 1\\
\mathrm{n2p}_7(x) &= x_0 \oplus x_6
\end{aligned}$$

C Masking and Graph Colouring

In Section 4, we raised the question of how many columns a (t, d)-Sharing Matrix can have. We can connect this problem to that of finding balanced colourings of a graph.

Graph Colouring. Consider a graph $(\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V} = \{0, \ldots, d\}^t$ corresponding to the rows of a t-column Sharing Matrix M. In other words, the vertices of $\mathcal{G}$ are words of length t with base $d+1$. There are $(d+1)^t$ vertices in total. Let two vertices in $\mathcal{G}$ be connected by an edge when their labels differ in exactly one coordinate, *i.e.* their Hamming distance is one[10]. Such a graph is called a Hamming graph $H(t, d+1)$. The case $d = 1$ is better known as a Hypercube graph [BH12]. It automatically follows that each pair of connected vertices $\{v_1, v_2\} \in \mathcal{E}$ have exactly $t-1$ coordinates in common. Recall, that in a (t, d)-Sharing Matrix, no two rows may have t common elements. The problem of finding column $t+1$ is thus equivalent to assigning to each vertex v a label $\mathcal{L}(v) \in \{0, \ldots, d\}$ such that $\forall \{v_1, v_2\} \in \mathcal{E} : \mathcal{L}(v_1) \neq \mathcal{L}(v_2)$. An example of such a labeling for $t = 3$ and $d = 1$ was shown in Eqn 4. Hence, if we can find a valid $(d+1)$-colouring $\mathcal{L}$ of the graph $H(t, d+1)$, then this implies the existence of a (t, d)-Sharing Vector that can be added to the Sharing Matrix M as extra column.

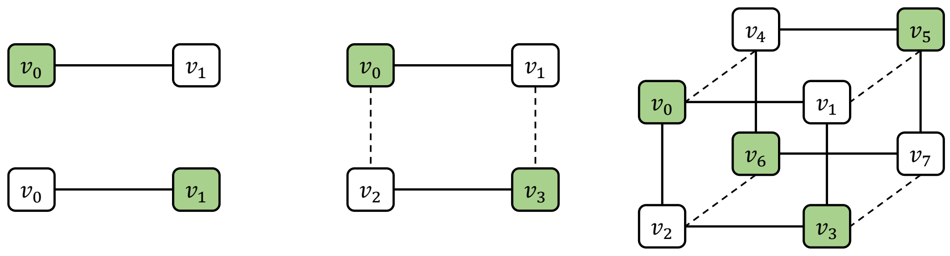
Given this equivalence, we can also provide an alternative proof for Lemma 3:

Proof. We consider the case $d = 1$, *i.e.* the vertices of $H(t, 2)$ are bitvectors of length t and $H(t, 2)$ defines a t-dimensional hypercube. We show the existence and uniqueness of the $t+1^{\text{st}}$ column by showing the existence and uniqueness of a 2-colouring of the graph. It is well known that all hypercube graphs are bipartite, *i.e.* can be coloured with only two colours. This proves the existence of a $t+1$-column $(t, 1)-$Sharing Matrix for any t. Next, we show the uniqueness of this column by showing that the 2-colouring of a hypercube graph is unique up to an inversion of the colours. Figure 9 depicts two 1-hypercubes $(t = 1)$ and shows clearly that a 2-colouring of the vertices is unique up to an inversion of the colours. We refer to the colouring as $\mathcal{L}^t$ and its inverse $\bar{\mathcal{L}}^t$. By definition, they have two properties:

$$\forall \{v_i, v_j\} \in \mathcal{E} : \mathcal{L}^t(v_i) \neq \mathcal{L}^t(v_j) \text{ and } \bar{\mathcal{L}}^t(v_i) \neq \bar{\mathcal{L}}^t(v_j) \tag{8}$$

$$\forall v_i : \mathcal{L}^t(v_i) \neq \bar{\mathcal{L}}^t(v_i) \tag{9}$$

[10]Note that we use the general (non-binary) notion of Hamming Weight which counts the number of different coordinates (not bits)



(a) Hypercube, $H(1,2)$ (b) Hypercube, $H(2,2)$ (c) Hypercube, $H(3,2)$

Figure 9: (a) unique 2-colouring of $H(1,2)$ up to inversion (b) extension of $H(1,2)$ to $H(2,2)$ (c) extension of $H(2,2)$ to $H(3,2)$. Dashed lines indicate new edges.

Now, we show by induction that a $t+1$-dimensional hypercube only has a unique colouring $\mathcal{L}^{t+1}$ and its inverse $\bar{\mathcal{L}}^{t+1}$. Consider a t-dimensional hypercube graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which can only be coloured using $\mathcal{L}^t$ or $\bar{\mathcal{L}}^t$. From this graph, we construct a hypercube graph of dimension $t+1$ with vertices $\mathcal{V}' = \mathcal{V} \times \{0,1\}$ and edges

$$\mathcal{E}' = \{\{(v_i,0),(v_i,1)\}, \forall v_i \in \mathcal{V}\} \cup \{\{(v_i,k),(v_j,k)\}, \forall \{v_i,v_j\} \in \mathcal{E}, k \in \{0,1\}\}$$

Naturally, a valid colouring $\mathcal{L}^{t+1}$ has to agree with either $\mathcal{L}^t$ or $\bar{\mathcal{L}}^t$ on the subgraphs $\mathcal{G}_0, \mathcal{G}_1$ with nodes $\mathcal{V} \times \{0\}$ and $\mathcal{V} \times \{1\}$, as both are isomorphic to $\mathcal{G}$, hence

$$(\mathcal{L}^{t+1}|_{\mathcal{G}_0},\ \mathcal{L}^{t+1}|_{\mathcal{G}_1}) \in \{(\mathcal{L}^t,\mathcal{L}^t),(\bar{\mathcal{L}}^t,\bar{\mathcal{L}}^t),(\mathcal{L}^t,\bar{\mathcal{L}}^t),(\bar{\mathcal{L}}^t,\mathcal{L}^t)\}$$

Now, edges of the form $\{(v_i,0),(v_i,1)\}$ and the colouring property (8) prohibit the choice of equal labelings. Hence, only two possibilties for $\mathcal{L}^{t+1}$ remain, which are identical up to an inversion:

$$(\mathcal{L}^{t+1}|_{\mathcal{G}_0},\ \mathcal{L}^{t+1}|_{\mathcal{G}_1}) = (\mathcal{L}^t,\bar{\mathcal{L}}^t),$$
$$(\bar{\mathcal{L}}^{t+1}|_{\mathcal{G}_0},\ \bar{\mathcal{L}}^{t+1}|_{\mathcal{G}_1}) = (\bar{\mathcal{L}}^t,\mathcal{L}^t),$$

□

As before, the proof cannot be generalized for $d > 1$. In Section 4.1, we therefore provided specific numbers in Table 2. With this Appendix, we mean to show that the problem of finding non-complete maskings is related to finding the number of $d+1$-colourings of Hamming graphs. To the best of our knowledge, there is not yet a formula to describe this number. We note that not all colourings can be transformed to columns for the Sharing Matrix, since many of them are equivalent up to a renaming of the colours.

D ANFs for Masked S-box

The following 2-splits are valid in a normal basis with $\beta = 205$.

$$F^A(x) = x_2x_0 \oplus x_2x_1 \oplus x_3x_1 \oplus x_3x_2x_0 \oplus x_3x_2x_1 \oplus x_4x_0 \oplus x_4x_2 \oplus x_4x_2x_0 \oplus x_4x_2x_1 \oplus x_5x_2 \oplus x_5x_4 \oplus x_6x_0 \oplus x_6x_2x_1 \oplus x_6x_4x_0 \oplus x_6x_5x_4 \oplus x_7x_1 \oplus x_7x_3 \oplus x_7x_3x_0$$

$$
\begin{aligned}
&\oplus x_7x_3x_1 \oplus x_7x_3x_2 \oplus x_7x_4x_1 \oplus x_7x_4x_2 \oplus x_7x_5 \oplus x_7x_5x_2 \oplus x_7x_5x_4\\
F^B(x) &= x_2x_1x_0 \oplus x_4 \oplus x_4x_3x_1 \oplus x_5x_0 \oplus x_5x_1x_0 \oplus x_5x_4x_1 \oplus x_6 \oplus x_6x_5x_1 \oplus x_7x_6x_2\\
&\oplus x_7x_6x_4\\
G^A(x) &= x_2x_0 \oplus x_2x_1 \oplus x_3x_1 \oplus x_3x_2x_1 \oplus x_4x_0 \oplus x_4x_2 \oplus x_4x_2x_0 \oplus x_4x_2x_1 \oplus x_5x_3\\
&\oplus x_5x_4x_2 \oplus x_6x_2x_0 \oplus x_6x_2x_1 \oplus x_6x_3x_2 \oplus x_6x_4 \oplus x_6x_4x_0 \oplus x_6x_5x_3 \oplus x_6x_5x_4\\
&\oplus x_7x_1 \oplus x_7x_2x_0 \oplus x_7x_3x_0 \oplus x_7x_4 \oplus x_7x_4x_0 \oplus x_7x_4x_1 \oplus x_7x_4x_2 \oplus x_7x_5x_3\\
G^B(x) &= x_0 \oplus x_1x_0 \oplus x_2x_1x_0 \oplus x_3x_1x_0 \oplus x_4x_1x_0 \oplus x_4x_3 \oplus x_5x_1x_0 \oplus x_5x_2x_1 \oplus x_5x_4x_0\\
&\oplus x_6 \oplus x_6x_4x_3 \oplus x_6x_5x_1 \oplus x_7x_6x_2 \oplus x_7x_6x_4
\end{aligned}
$$

To allow a convenient replication of our results we additionally provide the functions in a machine-readable notation (*i.e.* the 256-bit vector).

rep(F^A) = `0x0000000000000000000100110014170400010000000200420001001000726460`

rep(F^B) = `0x000000000001001000000000000000000000000040000000100040000a04010080`

rep(G^A) = `0x0000000000000000000001000017022400010100000310600010010000724460`

rep(G^B) = `0x000000000001001000000000000000000000000040100000100020048010808 8a`

Furthermore, we provide the equations for the conversion from a polynomial base of GF(2^8) with $\alpha = 2$ to a normal base with $\beta = 205$ (p2n) and the conversion back concatenated with the affine function of the AES S-box (n2p).

$$
\begin{aligned}
\text{p2n}_0(x) &= x_0 \oplus x_2 \oplus x_7\\
\text{p2n}_1(x) &= x_2 \oplus x_6\\
\text{p2n}_2(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_7\\
\text{p2n}_3(x) &= x_0 \oplus x_4 \oplus x_5 \oplus x_6\\
\text{p2n}_4(x) &= x_4 \oplus x_7\\
\text{p2n}_5(x) &= x_1 \oplus x_3 \oplus x_4 \oplus x_7\\
\text{p2n}_6(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus x_6 \oplus x_7\\
\text{p2n}_7(x) &= x_0 \oplus x_2 \oplus x_5 \oplus x_6
\end{aligned}
$$

$$
\begin{aligned}
\text{n2p}_0(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus 1\\
\text{n2p}_1(x) &= x_0 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_7 \oplus 1\\
\text{n2p}_2(x) &= x_1 \oplus x_2 \oplus x_3 \oplus x_6 \oplus x_7\\
\text{n2p}_3(x) &= x_3 \oplus x_5 \oplus x_6 \oplus\\
\text{n2p}_4(x) &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_7\\
\text{n2p}_5(x) &= x_1 \oplus x_7 \oplus 1\\
\text{n2p}_6(x) &= x_0 \oplus x_4 \oplus 1\\
\text{n2p}_7(x) &= x_0 \oplus x_3 \oplus x_6 \oplus x_7
\end{aligned}
$$

E Finding Sharing Matrices

Algorithm 3 Backtracking Procedure for constructing (t, d)-Sharing Matrices

```
1: $M \leftarrow$ from Lemma 2
2: $c \leftarrow t$
3: while True do
4:     for $i \in \{1, \ldots, (d+1)^t - 1\}$ do
5:         $\mathcal{L}_{i,c} \leftarrow \{0, \ldots, d\}$
6:     end for
7:     $\mathcal{L}_{0,c} \leftarrow \{0\}$
8:     while $M_c$ not completely determined do
9:         if $\exists r$: $\mathcal{L}_{r,c} = \emptyset$ then
10:            Break
11:        else if $\exists r$: $|\mathcal{L}_{r,c}| = 1$ then
12:            $M_{r,c} \leftarrow \mathcal{L}_{r,c}[0]$
13:        else
14:            Pick $r, l$ (& record backtrackpoint)
15:            $M_{r,c} \leftarrow \mathcal{L}_{r,c}[l]$
16:        end if
17:        for $i \in \{1, \ldots, (d+1)^t - 1\} \setminus \{r\}$ do
18:            if $\#\{j : M_{i,j} = M_{r,j}\} = t - 1$ then
19:                $\mathcal{L}_{i,c} \leftarrow \mathcal{L}_{i,c} \setminus \{M_{r,c}\}$
20:            end if
21:        end for
22:    end while
23:    if $M_c$ not completely determined then
24:        if Backtracking possible then
25:            Jump to last backtrackpoint
26:        else
27:            Stop Algorithm
28:        end if
29:    end if
30:    $c \leftarrow c + 1$
31: end while
```