Multiplicative Masking for AES in Hardware

Lauren De Meyer[1], Oscar Reparaz[1,2] and Begül Bilgin[1]

[1] imec - COSIC, KU Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[2] Square inc., San Francisco, USA

Abstract. Hardware masked AES designs usually rely on Boolean masking and perform the computation of the S-box using the tower-field decomposition. On the other hand, splitting sensitive variables in a multiplicative way is more amenable for the computation of the AES S-box, as noted by Akkar and Giraud. However, multiplicative masking needs to be implemented carefully not to be vulnerable to first-order DPA with a zero-value power model. Up to now, sound higher-order multiplicative masking schemes have been implemented only in software. In this work, we demonstrate the first hardware implementation of AES using multiplicative masks. The method is tailored to be secure even if the underlying gates are not ideal and glitches occur in the circuit. We detail the design process of first- and second-order secure AES-128 cores, which result in the smallest die area to date among previous state-of-the-art masked AES implementations with comparable randomness cost and latency. The first- and second-order masked implementations improve resp. 29% and 18% over these designs. We deploy our construction on a Spartan-6 FPGA and perform a side-channel evaluation. No leakage is detected with up to 50 million traces for both our first- and second-order implementation. For the latter, this holds both for univariate and bivariate analysis.

Keywords: DPA · Masking · Glitches · Sharing · Adaptive · Boolean · Multiplicative · AES · S-box · Side-channel

1 Introduction

Cryptographic primitives are designed to resist mathematical attacks such as linear or differential cryptanalysis. The designer typically assumes a classic adversarial model, where encryption is treated as a black box, only revealing inputs and outputs to adversaries. When these primitives are deployed in embedded devices, unintended signals such as the instantaneous power consumption or electromagnetic radiation leak sensitive information, effectively turning the black box into a gray box. Side-channel analysis is a cheap and scalable technique that allows the adversary to exploit these signals and extract secret keys or passwords. Hence, cryptography deployed into embedded devices needs not only mathematical but also physical security.

One particularly powerful attack, differential power analysis (DPA) was introduced in 1999 by Kocher *et al.* [KJJ99]. In this type of attack, the adversary feeds different plaintexts to an encryption algorithm using the same key and extracts sensitive information from the power traces he collects. Today, we aim at providing security against d^{th}-order DPA. In a d^{th}-order DPA attack, the adversary exploits any statistical moment of the power consumption up to order d. Since statistical moments are exponentially harder to estimate with the order d given sufficient noise (both in terms of numbers of samples and computational time), having a moderate security target $d = 1, 2$ often suffices in practice, especially when used in conjunction with complementary countermeasures [HOM06, CCD00].

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2018, No. 3, pp. 431–468
DOI:10.13154/tches.v2018.i3.431-468

In a side-channel secure implementation, the goal is to make the leakages of the values handled in the implementation independent of the sensitive inputs and sensitive intermediate variables. At the architectural level this is typically achieved by masking, which means the processed data is probabilistically split into multiple shares in such a way that one can only recover the sensitive data if all of its shares are known. Recovering secrets from shares is exponentially more difficult as noise increases; as this corresponds to estimating higher-order statistical moments with increasing noise levels [CJRR99, GP99].

Previous Work. The earliest masking schemes [GP99, Tri03, ISW03] were shown to be unsuitable for hardware implementations by Mangard *et al.* [MPG05, MPO05]. The vulnerability arises when unintended transitions of a signal or *glitches* occur, caused by non-idealities such as logic gates with non-zero propagation delays or routing imbalances. The glitches problem can be addressed at many levels: either by equalizing signal paths (which normally requires manual access to low-level routing details and a careful characterization of the logic library), by adding synchronization elements (such as registers or signal gating) or by using a masking scheme that is inherently secure under glitches. Extensive research has been done on countermeasures based on secret sharing and multiparty computation that are provably secure even in the presence of glitches. The prevailing schemes are those of Prouff and Roche [PR11] and *Threshold Implementations* (TI) by Nikova *et al.* [NRS11] which use polynomial and Boolean masking respectively. The latter was extended to higher-order security by Bilgin *et al.* (higher-order TI) [BGN+14a]. The similarities and differences between TI and the Private Circuits scheme [ISW03], which provides provable security if the circuit behaves ideally (no glitches), were analysed by Reparaz *et al.* (Consolidated Masking Schemes) [RBN+15]. Reparaz *et al.* also discuss how ISW can be implemented to provide security on hardware. More recently, Gross *et al.* presented Domain Oriented Masking [GMK16], which is also related to the original Private Circuits scheme [ISW03] with additional registers againts glitches and a different randomness consumption. These masking schemes have all been applied to Canright's tower-field AES S-box [Can05] due to its small foot-print and structure, resulting in a multitude of masked AES implementations [MPL+11, BGN+14b, CRB+16, GMK17, UHA17]. Those of Ueno *et al.* [UHA17], De Cnudde *et al.* [CRB+16] and Gross *et al.* [GMK17] are the smallest to date, with the latter requiring much less randomness.

In this paper we follow a different avenue. We do not apply Boolean masking to Canright's tower-field decomposition, but instead, we revisit the well-known concept of switching between different types of masking. Boolean masking schemes are compatible with linear operations but difficult to work out for non-linear functions. Akkar and Giraud [AG01] were the first to propose an adaptive masking scheme for AES at CHES 2001. The idea is to use Boolean masks for the affine operations and multiplicatively masked values for multiplications (or in the case of AES, inversion) and convert between the two types when necessary. At CHES 2002, Golić and Tymen [GT02] presented an inherent weakness of multiplicative masking, namely that it is vulnerable to first-order DPA because the zero element cannot be effectively masked multiplicatively. As a solution to this *zero problem*, they proposed to map each zero element to a non-zero element. The adaptive masking scheme was studied in depth and extended to higher-order security by Genelle *et al.* [GPQ11b]. So far, it has only been used in software implementations.

Our Contribution. We present the first hardware implementation of an adaptively masked AES. We describe glitch-resistant modules that convert between Boolean and multiplicative masking and that attend to the zero problem, based on the algorithmic descriptions provided for software in [GPQ10, GPQ11a, GPQ11b]. While this work focuses on the AES S-box, the methodology can be used to mask any inverse or power map-based S-box [AGR+16].

We optimize the number of inversions used and the randomness cost for first-order and second-order resistant AES, which both achieve a smaller area than the current state-of-the-art masked hardware AES implementations of [CRB+16] and [GMK17], while having comparable randomness and latency requirement. We formally discuss the security of our S-box and its components up to the level current state-of-the-art tools and methods allow. We also deploy our implementations into an FPGA for side-channel evaluation using a non-specific leakage assessment test to analyse practical security in a lab environment with low noise. No leakage is detected with up to 50 million traces, confirming that the security claims hold empirically.

2 Preliminaries

Notation. Multiplication and addition in the field $\mathbb{F}_q = \mathrm{GF}(2^k)$ are denoted by $\otimes$ and $\oplus$ respectively. We use & for multiplication in the field GF(2) (*i.e.* the AND operation). For ease of notation, we sometimes omit $\otimes$ and &. Square brackets $[\cdot]$ in formulas indicate where synchronization via registers or memory elements are used. An element $r \in \mathbb{F}_q$ drawn uniformly at random from $\mathbb{F}_q$ is shown as $r \xleftarrow{\$} \mathbb{F}_q$. We denote $\mathbb{F}_q^* = \mathbb{F}_q \setminus \{0\}$. The expected value of x is denoted $\mathbf{E}[x]$.

2.1 Adversarial Model

We consider a physical adversary model, in which an attacker can probe and observe up to d intermediate wires in each time period. This model is known as the d-probing model [ISW03]. To account for non-ideal (glitchy) circuits, we assume that any probed wire carrying a function output also leaks information about all function inputs up to the last register [RBN+15]. It has been shown in [FRR+10, RP10, DDF14] that security in the d-probing model implies security against d^{th}-order DPA as well given the independent leakage assumption of each share and its corresponding logic from the others.

2.2 Boolean and Multiplicative Masking

A popular countermeasure against d^{th}-order DPA is masking sensitive values by probabilistically splitting them into $d+1$ shares. Let $\diamond$ be some group operation. Then for any $x \in \mathbb{F}_q$ we process the sharing $\boldsymbol{x} = (s_0, \ldots, s_d)$ with $s_0 \diamond s_1 \diamond \ldots \diamond s_d = x$ instead of x itself. Similarly, $\boldsymbol{f(x)} = (f_0(\boldsymbol{x}), \ldots, f_d(\boldsymbol{x}))$ is a shared representation of a function $f(x)$.

Masked representations. We can distinguish different masked representations based on the splitting operation $\diamond$. A common choice is the exclusive-or operation $\oplus$, resulting in a *Boolean sharing*. We use b_i^x to denote Boolean shares of x: *i.e.*

$$\boldsymbol{x} = (b_0^x, \ldots, b_d^x) \qquad \Leftrightarrow \qquad x = \bigoplus_{i=0}^{d} b_i^x$$

In this paper we also use multiplicative sharing, which in a side-channel context is typically defined as

$$\boldsymbol{x} = (p_0^x, \ldots, p_d^x) \qquad \Leftrightarrow \qquad x = \Big(\bigotimes_{i=0}^{d-1} (p_i^x)^{-1}\Big) \otimes p_d^x$$

We refer to this sharing as a *type-I multiplicative sharing*. We further define a *type-II multiplicative sharing*:

$$\boldsymbol{x} = (q_0^x, \ldots, q_d^x) \qquad \Leftrightarrow \qquad x = \bigotimes_{i=0}^{d} q_i^x$$

This notation is more common in secret-sharing. We omit the superscript x when it is clear from context.

Masked operations. In Boolean masking, linear operations can trivially be applied locally on each share:

$$\boldsymbol{x} \oplus \boldsymbol{y} = (b_0^x, \ldots, b_d^x) \oplus (b_0^y, \ldots, b_d^y) = (b_0^x \oplus b_0^y, \ldots, b_d^x \oplus b_d^y)$$

Non-linear operations such as a multiplication on the other hand are less straightforward and much more costly to implement. The opposite situation arises if one uses multiplicative masking. In that case, linear operations are non-trivial but multiplication is local:

$$\boldsymbol{x} \otimes \boldsymbol{y} = (p_0^x, \ldots, p_d^x) \otimes (p_0^y, \ldots, p_d^y) = (p_0^x \otimes p_0^y, \ldots, p_d^x \otimes p_d^y)$$

Finding an efficient but glitch-resistant way to process Boolean shares in a non-linear operation has been a hot topic in the last years. A natural strategy is to switch back and forth between masked representations and perform each operation in its most compatible setting.

The zero-value problem. The fundamental security flaw of multiplicative masking was first pointed out by Golić and Tymen [GT02]. Multiplicative masking cannot securely encode the value 0. The mean power consumption of a single share p_i^x reveals whether the underlying secret is zero or non-zero, since $\mathbf{E}[p_i^x | x = 0] \neq \mathbf{E}[p_i^x | x \neq 0]$ for any share index i. This means that for any number of shares, the original multiplicative masking scheme is vulnerable to first-order DPA.

2.3 Masking in Hardware

Masking in hardware requires special care. The seminal work of Mangard *et al.* [MPG05, MPO05] showed that glitches can reveal sensitive information in hardware masked implementations that otherwise were expected to be secure.

Non-completeness. The concept of non-completeness appears in the work of Nikova *et al.* [NRS11] and follow-up works on higher-order security [BGN+14a, RBN+15]. Non-completeness between register stages has become a fundamental property for constructing provable-secure hardware implementations even if the underlying logic gates glitch. We recall here the definition of non-completeness: for any shared implementation $\boldsymbol{f}$ operating on a shared input $\boldsymbol{x}$, d^{th}-order non-completeness is satisfied if any combination of up to d shares of $\boldsymbol{f}$ is independent of at least one input share.

Masked Multiplier. Reparaz *et al.* [RBN+15] showed that a d^{th}-order masked multiplication in hardware can be constructed using only $d+1$ shares if the sharings of the inputs are independent (so as to not break non-completeness). One approach to do this is detailed in [GMK16] and is referred to as Domain Oriented Masking (DOM).

Our work uses as a masked AND gate the DOM-*indep* multiplier from [GMK16]. Let $\boldsymbol{x} = (b_0^x, b_1^x)$ and $\boldsymbol{y} = (b_0^y, b_1^y)$ be first-order Boolean sharings of bits x and y. A sharing of the multiplication result $z = x\&y$ is obtained by first calculating four partial products $t_{ij} = b_i^x \& b_j^y$, $i, j \in \{0, 1\}$ as in [ISW03]. When $i \neq j$, t_{ij} is called a cross-domain term and must be refreshed with a randomly drawn bit $r \xleftarrow{\$} \mathrm{GF}(2)$. After a register stage for synchronization, the shares (b_0^z, b_1^z) are computed.

$$\begin{aligned} b_0^z &= b_0^x \& b_0^y \oplus [b_0^x \& b_1^y \oplus r] \\ b_1^z &= b_1^x \& b_1^y \oplus [b_1^x \& b_0^y \oplus r] \end{aligned} \tag{1}$$

The second-order multiplier uses three bits of randomness $r \xleftarrow{\$} (\mathrm{GF}(2))^3$. The inputs and outputs have three shares and there are nine partial products t_{ij}.

$$b_0^z = b_0^x \& b_0^y \oplus [b_0^x \& b_2^y \oplus r_1] \oplus [b_0^x \& b_1^y \oplus r_0]$$
$$b_1^z = [b_1^x \& b_0^y \oplus r_0] \oplus b_1^x \& b_1^y \oplus [b_1^x \& b_2^y \oplus r_2]$$
$$b_2^z = [b_2^x \& b_0^y \oplus r_1] \oplus [b_2^x \& b_1^y \oplus r_2] \oplus b_2^x \& b_2^y$$

Note that we employ the special version of the DOM-*indep* multiplier where only the cross-domain terms are synchronized in registers. For efficiency, these registers are clocked on the negative edge as is done in [GSM17]. This is illustrated for the first-order multiplier in Figure 1.

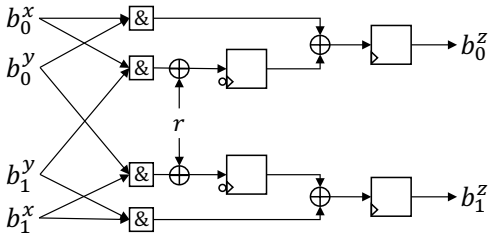

Figure 1: First-order DOM-*indep* multiplier

3 Design of an Adaptively Masked AES S-box

The AES S-box is an inversion in $\mathrm{GF}(2^8)$, followed by an affine transformation over bits. We adopt the idea of adaptive masking, where we use Boolean sharings for linear operations and multiplicative masks for non-linear operations. We thus implement the inversion by first converting the input from Boolean to multiplicative masking. The inversion then becomes a local operation on the individual shares:

$$\boldsymbol{x} = (p_0, \ldots, p_d) \Leftrightarrow \boldsymbol{x^{-1}} = (p_0^{-1}, \ldots, p_d^{-1})$$

We convert back to a Boolean masking to do the affine transformation.

In what follows, we first describe the conversion circuits between Boolean and multiplicative masking. We address the zero problem in § 3.3. An overview of the S-box can be found in Figure 5. While this section is written with AES in mind, the methodology can be applied to any S-box constructed from inversion or another power map in $\mathbb{F}_q$.

3.1 Masking Conversions

Following the strategy of [GPQ11b], we intuitively describe a higher-order conversion between Boolean and multiplicative shares with the following steps. Note that this description is not final and we will deviate from them slightly in § 3.2.

For $k = 1, \ldots, d$:

(a) Expansion: extend the sharing $\boldsymbol{x}$ with a new share of the target masking type. The number of target shares is augmented by one and the total number of shares is now $d + 2$.

(b) Synchronize the shares in a register

(c) Compression: Remove one share from the source sharing by partially unmasking. The number of source shares shrinks by one and the total number of shares is again $d + 1$.

Boolean to Multiplicative. More specifically, consider a conversion from Boolean to type-I multiplicative shares. After k iterations of the above steps, we have an intermediate sharing

$$\boldsymbol{x} = (p_0, \ldots, p_{k-1}, b_k, \ldots, b_d) \quad \text{where} \quad x = \Big(\bigotimes_{i=0}^{k-1} p_i^{-1}\Big) \otimes \Big(\bigoplus_{i=k}^{d} b_i\Big)$$

The number of target (multiplicative) shares is k and the number of source (Boolean) shares is $d+1-k$. In the expansion phase, we add a new multiplicative share by drawing a random p_k and multiplying it with all Boolean shares:

$$b'_i = p_k \otimes b_i \quad \text{for } i = k, \ldots, d \tag{2}$$

We now obtain a $d+2$ sharing

$$\boldsymbol{x} = (p_0, \ldots, p_k, b'_k, \ldots, b'_d) \quad \text{where} \quad x = \Big(\bigotimes_{i=0}^{k} p_i^{-1}\Big) \otimes \Big(\bigoplus_{i=k}^{d} b'_i\Big)$$

In the compression phase, we remove Boolean share b'_k by adding it to another Boolean share b'_{k+1}:

$$b''_{k+1} = b'_k \oplus b'_{k+1} \tag{3}$$

which brings us to a $d+1$ sharing

$$\boldsymbol{x} = (p_0, \ldots, p_k, b''_{k+1}, b'_{k+2}, \ldots, b'_d)$$

with $k+1$ target (multiplicative) shares and $d-k$ source (Boolean) shares. After d iterations, the sharing has been converted to $\boldsymbol{x} = (p_0, \ldots, p_{d-1}, b_d)$ such that $x = \big(\bigotimes_{i=0}^{d-1} p_i^{-1}\big) \otimes b_d$, which is equivalent to a type-I multiplicative sharing of x with $p_d = b_d$.

Multiplicative to Boolean. For the opposite conversion from multiplicative to Boolean shares, we consider a type-II multiplicative sharing, but the procedure for type-I is identical, apart from d additional inversions. Note that the first iteration starts with $k = 1$ and $b''_d = q_d$. In iteration k, we have the intermediate sharing

$$\boldsymbol{x} = (q_0, \ldots, q_{d-k}, b'_{d-k+1}, \ldots, b'_{d-1}, b''_d)$$

with k target (Boolean) shares and $d+1-k$ source (multiplicative) shares. In the expansion phase, a new Boolean share b'_{d-k} is added by splitting b''_d into $b'_d \oplus b'_{d-k}$ with b'_{d-k} randomly drawn. The $d+2$ shares of x are then

$$\boldsymbol{x} = (q_0, \ldots, q_{d-k}, b'_{d-k}, \ldots, b'_d) \quad \text{where} \quad x = \Big(\bigotimes_{i=0}^{d-k} q_i\Big) \otimes \Big(\bigoplus_{i=d-k}^{d} b'_i\Big)$$

In the compression phase, multiplicative share q_{d-k} is removed by multiplication with all Boolean shares:

$$b_i = q_{d-k} \otimes b'_i \quad \text{for } i = d-k, \ldots, d$$

resulting in the $d+1$ sharing

$$\boldsymbol{x} = (q_0, \ldots, q_{d-k-1}, b_{d-k}, \ldots, b_d) \quad \text{where} \quad x = \Big(\bigotimes_{i=0}^{d-k-1} q_i\Big) \otimes \Big(\bigoplus_{i=d-k}^{d} b_i\Big)$$

with $k+1$ target (Boolean) shares and $d-k$ source (multiplicative) shares.

We provide high-level descriptions for both conversions in pseudocode below. These pseudocodes are slightly different from the higher-order generalizations in [GPQ11b] (Algorithms 1 and 2) but representative of their first- and second-order descriptions.

Algorithm 1 Boolean to Multiplicative

Input: $\boldsymbol{x} = (b_0, \ldots, b_d)$
Output: $\boldsymbol{x} = (p_0, \ldots, p_d)$

```
for i = 0 to d − 1 do
    p_i ←$ F_q^*
    for j = i to d do
        b_j ← b_j ⊗ p_i
    end for
    **Register Stage**
    b_{i+1} ← b_{i+1} ⊕ b_i
end for
p_d ← b_d
```

Algorithm 2 Multiplicative to Boolean

Input: $\boldsymbol{x} = (q_0, \ldots, q_d)$
Output: $\boldsymbol{x} = (b_0, \ldots, b_d)$

```
b_d ← q_d
for i = d − 1 downto 0 do
    b_i ←$ F_q
    b_d ← b_d ⊕ b_i
    **Register Stage**
    for j = i to d do
        b_j ← b_j ⊗ q_i
    end for
end for
```

Conversions in Hardware: Dealing with glitches. The register stage between the expansion and compression phases is necessary because of the presence of glitches in hardware circuits. Without this register, the non-completeness of the conversion is broken and we have no security guarantees. Consider for example equations (2) and (3). Together, they compute the following

$$\begin{aligned} b''_{k+1} &= [p_k b_k] \oplus [p_k b_{k+1}] \\ &= p_k(b_k \oplus b_{k+1}) \end{aligned}$$

Without a register, the signal p_k might arrive late to the multiplication. As a result, two of the shares of x are combined on one wire $b_k \oplus b_{k+1}$ and the security is reduced by one order.

3.2 Specific Inversion Circuits

Why we use two types of multiplicative masking: Consider a type-I multiplicative masking, *i.e.* $\boldsymbol{x} = (p_0^x, p_1^x, \ldots, p_d^x) \Leftrightarrow x = \left(\bigotimes_{i=0}^{d-1}(p_i^x)^{-1}\right) \otimes p_d^x$. To obtain a type-I masking of its inverse x^{-1}, we can locally invert all shares p_i^x using $d+1$ unshared $\mathbb{F}_q$ inverters. Converting back to Boolean masking then requires d more $\mathbb{F}_q$ inverters. However, the following formula shows that we can do the entire masked inversion with only *one* unshared $\mathbb{F}_q$ inverter:

$$x^{-1} = \left(\left(\bigotimes_{i=0}^{d-1}(p_i^x)^{-1}\right) \otimes p_d^x\right)^{-1} = \left(\bigotimes_{i=0}^{d-1} p_i^x\right) \otimes (p_d^x)^{-1}$$

Indeed, by only locally inverting the last share p_d^x of a type-I multiplicative masking of x, we obtain a type-II multiplicative sharing of its inverse x^{-1}:

$$\boldsymbol{x^{-1}} = (q_0^{(x^{-1})}, q_1^{(x^{-1})}, \ldots, q_d^{(x^{-1})}) = (p_0^x, p_1^x, \ldots, (p_d^x)^{-1})$$

Note that regardless of the security order d, only *one* unshared inverter is required this way.

We now look in more detail at the first- and second-order implementations of the conversions.

First-order. The complete first-order masked inversion including the resulting circuits for first-order conversions between Boolean and multiplicative masking is shown in Figure 2. The left side of the figure converts a Boolean sharing $\boldsymbol{x} = (b_0, b_1)$ to a type-I multiplicative sharing (p_0, p_1) such that $x = p_0^{-1} p_1$. With a non-zero $r_0 \xleftarrow{\$} \mathbb{F}_q^*$, the multiplicative shares are calculated as

$$\begin{aligned} p_0 &= r_0 \\ p_1 &= [b_0 r_0] \oplus [b_1 r_0] \end{aligned}$$

The right side of the circuit converts a type-II multiplicative masking of x^{-1} into a Boolean masking. This requires another random $r_1 \xleftarrow{\$} \mathbb{F}_q$:

$$\begin{aligned} b'_0 &= r_1 q_0 \\ b'_1 &= [q_1 \oplus r_1] q_0 \end{aligned}$$

These procedures are identical to those described in Algorithms 1 and 2.

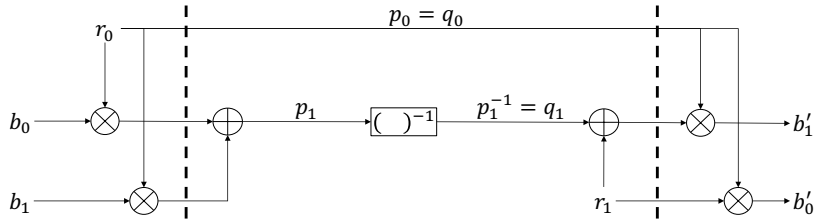

Figure 2: First-order shared implementation of an inversion in $\mathbb{F}_q$. The dashed lines depict registers.

Second-order. Adopting the same algorithms for $d + 1 = 3$ shares does not provide second-order secure conversions (see Appendix A). We require an extra refreshing of additive shares. Figure 3 depicts our circuit for the second-order shared inversion in $\mathbb{F}_q$. The conversion from a Boolean to a type-I multiplicative sharing is depicted on the left side of the figure. The conversion requires three units of randomness: $r_0, r_1 \xleftarrow{\$} \mathbb{F}_q^*$ and the extra refreshing $u \xleftarrow{\$} \mathbb{F}_q$. The multiplicative shares are as follows:

$$\begin{aligned} p_0 &= r_0 \\ p_1 &= r_1 \\ p_2 &= \Big[r_1\big([r_0 b_0] \oplus [r_0 b_1 \oplus u]\big)\Big] \oplus \Big[r_1\big([r_0 b_2] \oplus u\big)\Big] \end{aligned}$$

For the opposite conversion (shown on the right side of Figure 3), we start from a type-II multiplicative masking. This means we only need to invert the last share, p_2. We calculate the Boolean shares of x^{-1} as

$$\begin{aligned} b'_0 &= [r_3 \oplus u] q_0 \\ b'_1 &= [r_2 q_1 \oplus u] q_0 \\ b'_2 &= \big[[q_2 \oplus r_2] q_1 \oplus r_3\big] q_0 \end{aligned}$$

The conversion again uses three units of randomness, $r_2, r_3, u \xleftarrow{\$} \mathbb{F}_q$, although we can recycle the refreshing mask u from the Boolean to multiplicative conversion. Each conversion thus uses only 2.5 units of randomness.

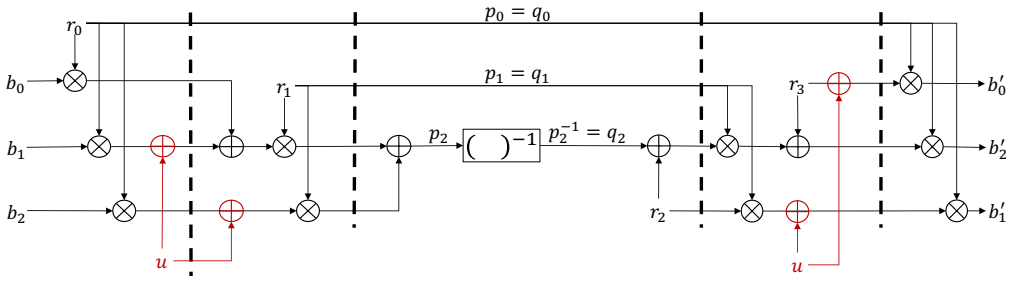

Figure 3: Second-order shared implementation of an inversion in $\mathbb{F}_q$. The dashed lines depict registers.

Our procedures differ slightly from those of Genelle *et al.* [GPQ11b], especially in the smaller use of randomness (we expand on this in Appendix A). For a general randomness strategy for higher-order conversions, we refer to [GPQ11b], but we note that their randomness cost is not necessarily optimal for each target security order d. A custom approach can result in a lower cost.

3.3 The Zero Problem

We now describe how to circumvent the zero problem of multiplicative masking. Both in MPC literature [DK10] and in software masking [GPQ10], it has been proposed to map each zero element in $\mathbb{F}_q$ to a non-zero element in $\mathbb{F}_q^*$ using a Kronecker Delta function *before* converting to multiplicative masks.

In the AES S-box, we need to do an inversion in $\mathbb{F}_q$. Both the zero and unit element of $\mathbb{F}_q$ are their own inverses:

$$x^{-1} = x \text{ for } x \in \{0, 1\}$$

It is therefore sufficient to replace each zero element by a "one" before the inversion and change it back afterwards. Consider a Kronecker delta function $\delta(x)$:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

We can write the inversion of any $x \in \mathbb{F}_q$ as follows:

$$x^{-1} = (x \oplus \delta(x))^{-1} \oplus \delta(x)$$

We thus require a circuit that computes a shared Kronecker delta function $\boldsymbol{\delta(x)}$. Its output (a sharing of "zero" or a sharing of "one") is to be added to the input of the conversion from Boolean to multiplicative masking and to the output of the conversion from multiplicative to Boolean masking (see Figure 5). This way, any zero element goes through the $\mathbb{F}_q$ inversion as a "one" and is thus never shared multiplicatively.

The Kronecker delta function $\delta(x)$ can be calculated with an n-input AND, or equivalently, a $\log_2(n)$-level 2-input AND tree with the inverted bits of x as input:

$$\delta(x) = \bar{x}_0 \& \bar{x}_1 \& \bar{x}_2 \& \ldots \& \bar{x}_{n-1}$$

The circuit is shown for $n = 8$ in Figure 4 with $\boldsymbol{x_i}$ a sharing of the i^{th} bit of x. In software, it has been realized using masked table lookups [GPQ10] and bit-slicing [GPQ11a]. We implement each AND gate with a DOM-*indep* multiplier [GMK16]. We denote by r_j the

randomness needed for each gate. As each multiplier requires one register stage, the entire circuit of Figure 4 takes three clock cycles (regardless of the number of shares).

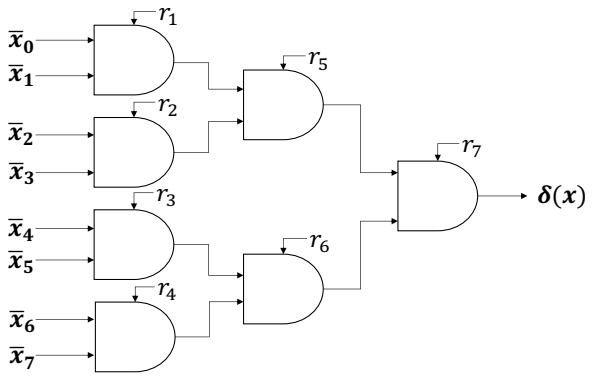

Figure 4: Circuit for the shared Kronecker delta function $\boldsymbol{\delta(x)}$ for $n = 8$

We note that a trade-off can be made here between latency and area. It is possible to reduce the depth of the tree (and thus the number of clock cycles) at the cost of a larger fan-in for the AND gates, which results in a considerable increase in area for shared implementations. In this paper, we choose to work only with 2-input AND gates in order to minimize circuit area.

First-order optimizations. In a straightforward first-order secure implementation of $\delta(x)$, each input bit has two shares and each DOM-AND gate requires 1 extra random bit $r_j \xleftarrow{\$} \mathrm{GF}(2)$. The circuit thus receives a total of 23 bits. That is a lot of entropy for a function that outputs only 2 bits. In order to bring down the randomness cost of the circuit, we decide to recycle some of the bits across the multiplication gates. A theoretical framework for this was presented in [FPS17]. Following this would result in a total randomness cost of 5 units: one bit in each of the three layers and one bit each for the refreshing after layer 1 and after layer 2. We now push the cost even further by using custom optimizations.

We rewrite the DOM equations (1) and note that they have a special property:

$$\begin{aligned} b_i^z &= b_i^x b_i^y \oplus [b_i^x b_{i\oplus 1}^y \oplus r] \\ &= b_i^x y \oplus r \end{aligned}$$

The DOM gate thus uses its inputs somewhat asymmetrically since the output shares depend only on the unmasked second input y and not on its sharing. This means that any randomness that has been used to mask y *before* arriving at this gate, disappears from its output sharing $\boldsymbol{z}$. Hence, we can reuse this randomness in the next layer. In our case, we use the more significant bit (depicted as the lower input to an AND gate in Fig. 4) as the "second input" and we conclude that the second layer of the Kronecker implementation removes any dependence of the data on r_2 and r_4. In contrast, reusing r_1 (or r_3) in layer two is not advisable. Moreover, for a first-order implementation (only univariate matters), the upper and lower two gates in the first layer have independent inputs and outputs, and can therefore use the same randomness as long as layer two does not.

We propose the following use of randomness:

$$r_1 = r_3 \xleftarrow{\$} \mathrm{GF}(2) \qquad r_5 \xleftarrow{\$} \mathrm{GF}(2) \qquad r_7 = r_1$$
$$r_2 = r_4 \xleftarrow{\$} \mathrm{GF}(2) \qquad r_6 = [r_5 \oplus r_2]$$

We are thus able to reduce the randomness consumption of the first-order Kronecker delta implementation from 7 to only 3 bits. We refer to Appendix C for the probability distributions of intermediate and output wires of this circuit with our randomness optimization. We verified that these probability distributions are independent of the secret input. Moreover, we note that these probability distributions are the same as in the circuit without randomness optimization.

Second-order optimizations. A second-order implementation uses three bits of randomness per multiplication: $r_j = (r_{j0}, r_{j1}, r_{j2}) \xleftarrow{\$} (\mathrm{GF}(2))^3$. Again, instead of consuming 21 bits of extra randomness in the circuit, we propose a recycling of the bits. Following the framework of [FPS17] would require five groups of three fresh random bits, *i.e.* 15 bits. Our customization is more restricted in the higher-order case because of the possibility of multivariate leakage. We still have the special composability property of the DOM gates, but the gates in the first layer can no longer be considered independent. We propose the following:

$$r_1, r_2, r_3, r_4 \xleftarrow{\$} (\mathrm{GF}(2))^3$$
$$r_{50} = r_{30}, r_{51} = r_{41}, r_{52} = [r_{32} \oplus r_{42}]$$
$$r_{60} = r_{10}, r_{61} = r_{21}, r_{62} = [r_{12} \oplus r_{22}]$$
$$r_{70} = [r_{11} \oplus r_{31}], r_{71} = [r_{20} \oplus r_{40}], r_{72} \xleftarrow{\$} \mathrm{GF}(2)$$

We thus reduce the randomness consumption of the second-order Kronecker delta implementation from 21 to 13 bits. The probability distributions of relevant (pairs of) wires can again be found in Appendix C.

3.4 The S-box

We summarize the AES S-box circuit in Figure 5. The local inversion is based on the smallest unshared AES S-box implementation to date by Boyar, Matthews and Peralta [BMP13]. More details on our adaptation of this circuit are given in Appendix B. The registers are depicted with grey dotted lines. In a first-order implementation each conversion has a latency of one cycle, whereas in a second-order implementation, it is two clock cycles. The

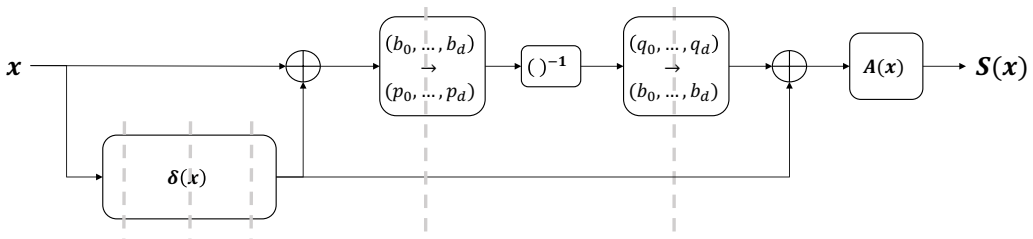

Figure 5: First-order adaptive masking implementation of the AES S-box. The dotted grey lines depict registers.

S-box input needs to be fed to the $\boldsymbol{\delta(x)}$ circuit three clock cycles before the first conversion. This could cost us three cycles of S-box latency as well as three stages of $8 \times (d+1)$-bit registers. Instead, we reorganize the state array and key schedule such that the Kronecker delta function can be *precomputed*. We describe this in the next Section.

4 AES Architecture and Control

The ShiftRows, MixColumns and AddRoundKey stages in AES are all linear and thus trivially masked by instantiating $d+1$ copies, one for each share of the state and key schedule. Following previous masked AES implementations, we use a byte-serialized architecture with a pipelined S-box as shown in Figure 5. Note that instead of the serialized architecture from [MPL+11], we use a similar architecture to that of [GMK16, Fig. 5] since it exhibits a more compact and efficient datapath. We adapt [GMK16] to accommodate for our S-box that needs a three-cycle precomputation of the Kronecker delta function.

4.1 State Array

The byte-serialized architecture from [GMK16] is very efficient in terms of clock cycles, since it performs the MixColumns, ShiftRows and AddRoundKey operations in parallel to SubBytes. Figure 6 (left) shows the state array with its *normal* meandering movement during the SubBytes operation in black full lines and the ShiftRows functionality in blue dotted lines. The column of registers that is the input of the MixColumns operation is indicated by a red striped frame, whereas the registers receiving the output of MixColumns once cycle later are specified by a full red frame.

The S-box input is taken from State 00, while the Kronecker delta input starts computing three cycles beforehand on State 30. In order to have State 30 ready for the Kronecker function, we have to put the MixColumns operation in the second column (instead of the first column as in [GMK16]). ShiftRows is performed when the sixteenth and last S-box output enters the state. We also adapt the ShiftRows connections such that all bytes end up one column to the right of the actual ShiftRows result. This means that the normally first column is the first MixColumns input (state bytes 01,11,21,31) and the normally last column now occupies state bytes 00,10,20 and 30. During the next four clock cycles, we rotate the state by returning byte 00 to the state input (33) untouched. After those four cycles, the state columns are restored to their correct order and the first S-box input is

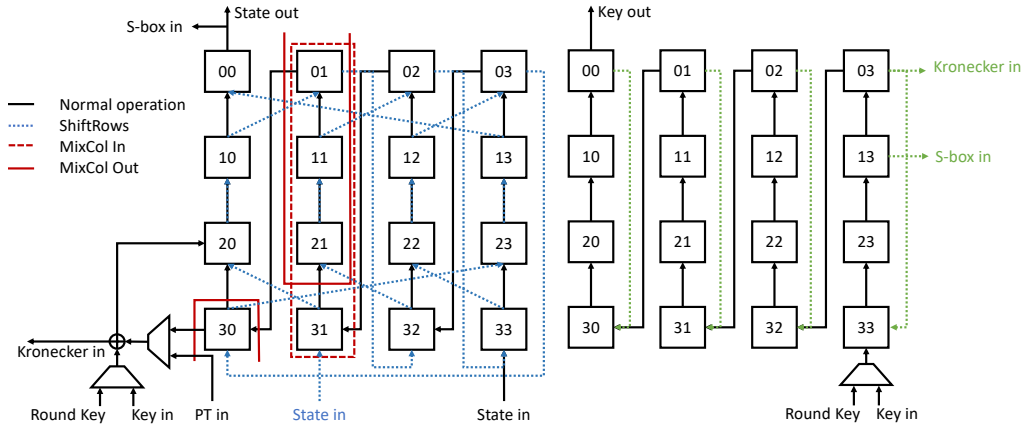

Figure 6: State and Key Array

ready in State 00. Moreover, its output to the Kronecker function is also ready at this point. The key schedule is synchronized with the state in a way that the partial Round Key to be used in that clock cycle corresponds to State 30. The AddRoundKey stage is embedded in the connection between State 30 and State 20 and its output is the input to the Kronecker delta function.

4.2 Key Array

The key array is depicted in Figure 6 (right) and is identical to that of [GMK16, Fig. 5]. The normal meandering operation is indicated in black full arrows, while the rotating movement is illustrated by green dotted arrows. The key state rotates in order to put its last column through the AES S-box. Note that this key array requires a lot fewer multiplexers than that of [MPL+11] because the direction of the normal operation corresponds to that of the rotations. The Round Key byte that is used in the AddRoundKey stage is constructed in three different ways, depending on which state byte it is added to:

Key 00 $\oplus$ S-box Out $\oplus$ RCon	for the first state byte
Key 00 $\oplus$ S-box Out	for the next three bytes
Key 00 $\oplus$ Key 03	for the remaining 12 bytes

The result is fed back into the key state as Key 33.

4.3 Control

We now go into more detail on the scheduling of the 24 clock cycles (0 to 23) that make up one encryption round when the S-box latency is four cycles (as in our second-order implementation). Table 1 details the control of the register movement and Table 2 shows how various inputs to the states and the S-box change.

Table 1: State and key control during one round of encryption

Cycle	State Shift	MixColumns	Key Shift
0-2	Meander	No	Meander
3	Meander	Yes	Meander
4-6	Meander	No	Meander
7	Meander	Yes	Meander
8-10	Meander	No	Meander
11	Meander	Yes	Meander
12-15	Meander	No	Meander
16-21	Meander	No	Rotate
22	ShiftRows	No	Rotate
23	Meander	Yes	Rotate

The 16 bytes of the state register are fed to the S-box in cycles 3 to 18 of each round of encryption. This means the Kronecker delta function receives the same 16 bytes three cycles before that: in cycles 0 to 15. During these cycles, the key state follows its meandering movement and Key 00 is used to construct the Round Key byte. In the remaining clock cycles (from cycle 16 until cycle 23), the key array is rotating. The last column of the array is fed through the Kronecker delta function in cycles 17 to 20 and through the S-box in cycles 20 to 23, which means their outputs are ready for the first four Round Key calculations four cycles later: in cycles 0 to 3.

The state receives its S-box outputs in cycles 7 to 22. In the last cycle (22), we do the adapted ShiftRows that puts each state byte one extra column to the right. The first

MixColumns operation is in the next cycle (23), which means the first input byte to the Kronecker delta function (in State 30) is ready in cycle 0. During cycles 23 to 2, State 00 holds bytes of the last column and is thus fed back into State 33. The MixColumns operation occurs four times every four cycles, *i.e.* in cycles 23, 3, 7 and 11 (except in the last round of encryption).

Table 2: State and key inputs during one round of encryption (except during loading)

Cycle	Round Key	Kronecker In	SBin	State In	S20
0	K00 ⊕ SBout ⊕ Rcon	S30 ⊕ RndKey	-	S00	Krncker In
1-2	K00 ⊕ SBout	S30 ⊕ RndKey	-	S00	Krncker In
3	K00 ⊕ SBout	S30 ⊕ RndKey	S00	-	Krncker In
4-6	K00 ⊕ K03	S30 ⊕ RndKey	S00	-	Krncker In
7-15	K00 ⊕ K03	S30 ⊕ RndKey	S00	SBout	Krncker In
16	-	-	S00	SBout	S30
17-18	-	K03	S00	SBout	S30
19	-	K03	-	SBout	S30
20	-	K03	K13	SBout	S30
21	-	-	K13	SBout	S30
22	-	-	K13	SBout	S31
23	-	-	K13	S00	S30

The first round of encryption (loading of the inputs) starts in cycle 0 with the data and key inputs replacing respectively State 30 and the Round Key. In total, one AES encryption is obtained in $10 \times 24 + 16 = 256$ cycles. Our first-order AES implementation has the same latency in spite of the S-box requiring only two cycles. Given the AES design, it is difficult to exploit an S-box latency below four cycles.

5 Security Evaluation

In this section, we elaborate on the security of the first- and second-order AES constructions against a probing adversary in the presence of glitches. Neither formal proofs in a particular security model nor empirical leakage detecting tools can in their own capacity provide full evidence for security. A security evaluation is incomplete without complementary analyses following both methodologies. Therefore, our approach consists of three stages: first in § 5.1, we address the security of the S-box under the ideal circuit assumption using the notion of strong non-interference [BBD+16, BBP+16]. Next in § 5.2, we evaluate the security of the S-box in the presence of glitches, using leakage detection tools available in literature. Finally in § 5.3 we complete the evaluation by analyzing our whole circuit on a physical device.

5.1 Security of the S-box in a theoretical framework

We now use the concept of Strong Non-Interference (SNI) [BBD+16] to prove that the S-box construction is theoretically secure. We use the same methodology as the proof of [BBD+16, Fig. 4]. Recall the definition of SNI:

Definition 1 (Strong Non-Interference (SNI) [BBP+16])**.** An algorithm is d-strong non-interferent (or d-SNI) if and only if for every set $\mathcal{I}$ of t_1 probes on intermediate variables (*i.e.* no output wires or shares) and every set $\mathcal{O}$ of t_2 probes on output shares such that $t_1 + t_2 \leq d$, the set $\mathcal{I} \cup \mathcal{O}$ can be simulated by only t_1 shares of each input.

Now, consider our S-box in Figure 7, consisting of six parts: $\mathcal{A}_1$, $\mathcal{A}_3$ and $\mathcal{A}_5$ are affine (only computing share wise) and $\mathcal{A}_2$, $\mathcal{A}_4$ and $\mathcal{A}_6$ are d-SNI as proven in Appendices D

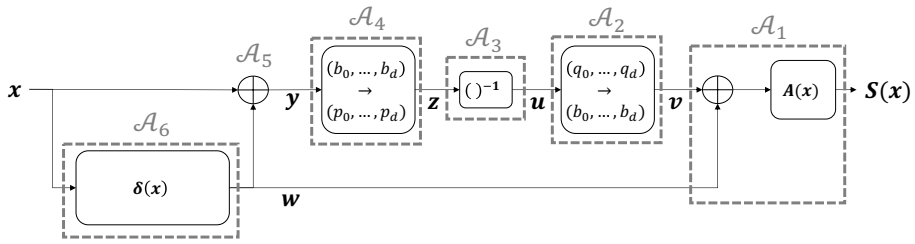

Figure 7: AES S-box

and E. The proof starts from the output and backtracks to the input. We denote by $\mathcal{I}_i$ the set of intermediate probes in gadget $\mathcal{A}_i$ and by $\mathcal{O}$ the set of output probes on $\boldsymbol{S(x)}$. The sets are constrained by $|\mathcal{O}| + \sum_{i=1}^{6} |\mathcal{I}_i| \leq d$. We further define $\mathcal{S}_i$ as the set of shares that are required at the input of block $\mathcal{A}_i$ in order to be able to simulate the probes in the remainder of the circuit, *i.e.* $\bigcup_{j=1}^{i} \mathcal{I}_i \cup \mathcal{O}$. We subsequently treat this set as a set of probes that needs to be simulated using input shares from a previous block $\mathcal{A}_{i-1}$. This way, we gradually move towards the input and try to show that the number of input shares of x required to simulate all probes $\bigcup_{i=1}^{6} \mathcal{I}_i \cup \mathcal{O}$ is at most $\sum_{i=1}^{6} |\mathcal{I}_i|$.

Consider for example block $\mathcal{A}_4$ in Table 3. This block has output z and input y. The set of shares of z, $\mathcal{S}_3$ is constrained by $|\mathcal{S}_3| \leq |\mathcal{S}_2| + |\mathcal{I}_3|$. Since $\mathcal{A}_4$ is d-SNI and since $|\mathcal{S}_3 \cup \mathcal{I}_4| \leq d$, we have that the number of shares of y required to simulate $\mathcal{S}_3 \cup \mathcal{I}_4$ is at most $|\mathcal{I}_4|$. We call this set of shares $\mathcal{S}_4$. Now, since we are able to simulate $\mathcal{S}_3$ using $\mathcal{S}_4$ and since $\mathcal{S}_3$ is able to simulate the remaining probes $\bigcup_{i=1}^{3} \mathcal{I}_i \cup \mathcal{O}$, we know that the set of shares $\mathcal{S}_4$ is sufficient to simulate $\bigcup_{i=1}^{4} \mathcal{I}_i \cup \mathcal{O}$.

Table 3: Proof that the S-box in Figure 7 is d-SNI for $d = 1, 2$

	Probes	**Constraints**	**Details**
	$S(x) : \mathcal{O}$	$\lvert\mathcal{O}\rvert + \sum_{i=1}^{6} \lvert\mathcal{I}_i\rvert \leq d$	
$\mathcal{A}_1$	$v : \mathcal{S}_{1,1}; w : \mathcal{S}_{1,2}$	$\lvert\mathcal{S}_{1,k}\rvert \leq \lvert\mathcal{I}_1\rvert + \lvert\mathcal{O}\rvert$	Affine
$\mathcal{A}_2$	$u : \mathcal{S}_2; w : \mathcal{S}_{1,2}$	$\lvert\mathcal{S}_2\rvert \leq \lvert\mathcal{I}_2\rvert$	d-SNI
$\mathcal{A}_3$	$z : \mathcal{S}_3; w : \mathcal{S}_{1,2}$	$\lvert\mathcal{S}_3\rvert \leq \lvert\mathcal{I}_3\rvert + \lvert\mathcal{S}_2\rvert$	Affine
$\mathcal{A}_4$	$y : \mathcal{S}_4; w : \mathcal{S}_{1,2}$	$\lvert\mathcal{S}_4\rvert \leq \lvert\mathcal{I}_4\rvert$	d-SNI
$\mathcal{A}_5$	$x : \mathcal{S}_{5,1}; w : \mathcal{S}_{5,2}$	$\lvert\mathcal{S}_{5,1}\rvert \leq \lvert\mathcal{I}_5\rvert + \lvert\mathcal{S}_4\rvert$ $\lvert\mathcal{S}_{5,2}\rvert \leq \lvert\mathcal{I}_5\rvert + \lvert\mathcal{S}_{1,2}\rvert$	Affine
$\mathcal{A}_6$	$x : \mathcal{S}_{5,1} \cup \mathcal{S}_6$	$\lvert\mathcal{S}_6\rvert \leq \lvert\mathcal{I}_6\rvert$	d-SNI

Table 3 shows that we need $|\mathcal{S}_{5,1} \cup \mathcal{S}_6| < |\mathcal{S}_4| + |\mathcal{I}_5| + |\mathcal{I}_6| < |\mathcal{I}_4| + |\mathcal{I}_5| + |\mathcal{I}_6|$ shares of the input to simulate all d-tuples of probes in the circuit. This proves that the S-box is d-SNI.

5.2 Practical Evaluation of Glitch Security of the S-box

A useful property for the synthesis of secure circuits in the presence of glitches is non-completeness [NRS11]. We use the VerMI tool described in [ANR17] to verify the security of the gadgets that create the S-box, *i.e.* the conversions and the Kronecker delta. This tool was designed specifically for masked hardware implementations. In particular, it can verify if a circuit satisfies the non-completeness property from register to register. By applying this tool directly to the RTL HDL descriptions of our gadgets, we confirm that each stage is non-complete and therefore secure in the univariate setting in the presence of glitches if the shared input does not have a secret dependent bias. We verify this condition on the input sharing independently (Appendix C).

We note that it has been implied that verifying glitch security and strong non-interference separately does not guarantee composability in a glitchy environment [FGP+17]. In section 5.1, we have given security proofs for the S-box as best as we could with the tools at our disposal. In this section, we consider glitches. The combined theoretical verification of "glitchy" SNI is an interesting direction for future research. However, note that SNI is not a *necessary* condition for the S-box to be secure. We further evaluate the security of the entire S-box using state-of-the-art tools.

We use the simulation tool of [Rep16], in which we exhaustively probe the S-box and create power traces using an identity leakage model. These traces do not only contain explicit intermediates (stabilized values on wires) but also values that could be observed in a glitch (transient values on wires). We *exhaustively* probe the S-box in this way in a completely *noiseless* setting and create up to 100 million simulated traces. For more details, we refer to [Rep16]. We detect no univariate leakage with up to 100 million traces nor bivariate in the case of our second-order gadgets. We draw the same conclusions when using the tool described in [DBR18]. This tool essentially exhausts every possible glitch in the computation by verifying that there is no mutual information between the secret and all possible (pairs of) glitch-extended probes.

While the theoretical possibility of a very weak bias still exists we would need more than 100 million traces to detect it and thus the practical implications of this are thin: if the leak is not even *detected* with 100 million traces in a *noiseless* scenario, it would take even considerably more traces to *exploit* it (perform key-recovery) in a *realistic noisy* scenario.

5.3 Physical Evaluation

After evaluating the S-box both theoretically and empirically in simulation, we finally put our entire AES design to the test in a physical environment.

Setup. We program a Xilinx Spartan6 FPGA with both our first- and second-order design on a SAKURA-G board, specifically designed for side-channel evaluation. For the synthesis, we use the Xilinx ISE option `KEEP_HIERARCHY` to prevent optimization across modules (and in particular across shares). To minimize platform noise, we split the implementation over a crypto FPGA, which handles the AES encryption and a control FPGA, which communicates with the host computer and supplies masked data to the crypto FPGA. The FPGA's are clocked at 3.072 MHz and sampled at 1GS/s.

The crypto FGPA is also equipped with a PRNG to generate the randomness required in every clock cycle. This PRNG is loaded with a fresh seed for every encryption. In contrast with other state-of-the-art masked implementations, we have to be able to generate one or two non-zero bytes for the multiplicative masks. We refer to Appendix F for a description of how we achieve this in practice, without stalling the pipeline.

Univariate. We perform a non-specific leakage detection test [BCD+13] using the methodology from [RGV17]. This means we gather power traces in two sets: the first corresponding to encryptions of a fixed plaintext and the other to encryptions of random plaintexts. We choose the fixed plaintext equal to the key in order to test the special case of zero inputs to the S-box in the first round. Nonzero S-box inputs then occur in encryption round two and are thus naturally also tested. The two sets of measurements are compared using the t-test statistic. When the t-statistic at order d crosses the threshold $T = \pm 4.5$, the null hypothesis "The design has no d^{th}-order leakage" is rejected with confidence $> 99.999\%$. On the other hand, when the t-statistic remains below this threshold, we corroborate that side-channel information is not distinguishable at order d.

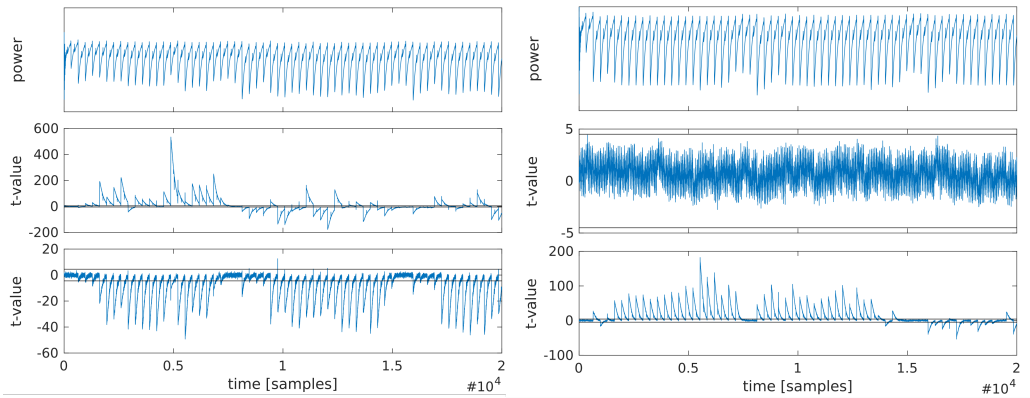

Figure 8: Non-specific leakage detection test on 2.5 rounds of encryption of a first-order protected AES. Left: PRNG off; 12 000 traces. Right: PRNG on; 50 million traces. Rows(top to bottom): exemplary power trace, first-order, second-order t-value.

The results for our first-order design are shown in Figure 8. Each trace consists of 64 clock cycles, comprising about two and a half rounds of encryption. An example of such a trace is shown in Figure 8, top. To verify the soundness of our setup, we first perform the leakage detection test with the PRNG turned off (*i.e.* unmasked implementation). This is shown in the left column of the figure and as expected, the design presents severe leakage at only 12 000 traces. On the right side, we do the leakage detection test with the PRNG turned on. We do not observe evidence for first-order leakage with up to 50 million power traces. The design does leak in the second order, as anticipated.

Similarly, we show the test results for our second-order design in Figure 9. The leakage when the PRNG is turned off (left column) is clear. The masked implementation (right

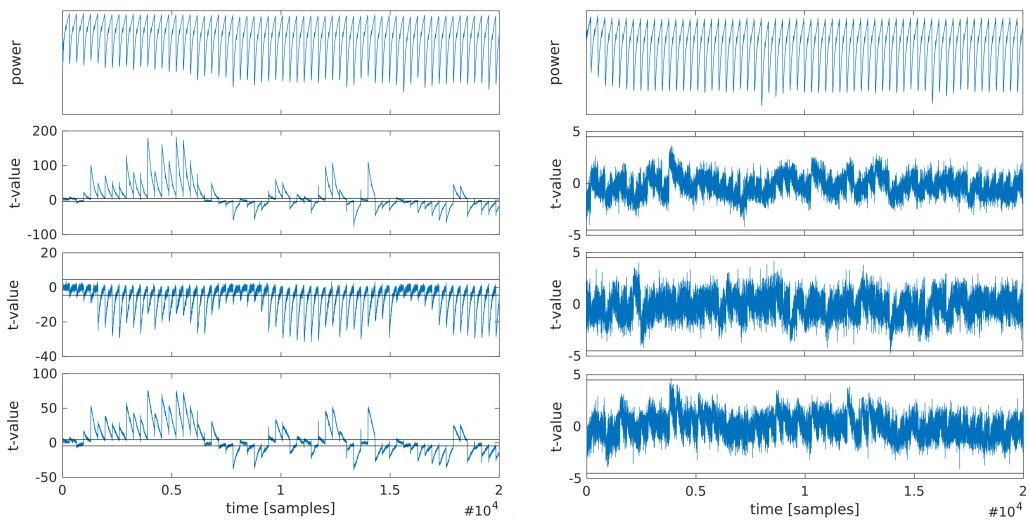

Figure 9: Non-specific leakage detection test on 2.5 rounds of encryption of a second-order protected AES. Left: PRNG off; 12 000 traces. Right: PRNG on; 50 million traces. Rows(top to bottom): exemplary power trace, first-order, second-order, third-order t-value.

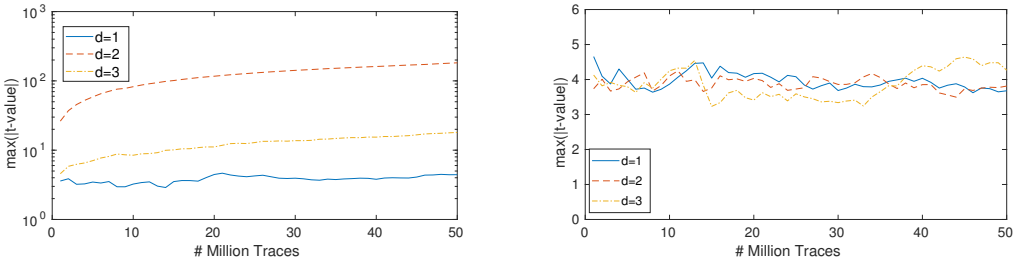

Figure 10: Evolution of the maximum absolute t-value across the measurements. Left: First order. Right: Second order.

column) does not present evidence for first- nor second-order leakage with up to 50 million power traces. While we would expect the third-order t-statistic to surpass the threshold, this is not yet the case due to platform noise.

We also track the evolution of the maximum absolute t-test value as a function of the number of traces taken. This is shown in Figure 10 for the first-order (left) and second-order (right) protected AES implementations. On the left, we clearly see an increase in the absolute t-value of the second- and third-order moment, while the statistic for first order is stable. For our second-order implementation, the noise of the platform prevents us from seeing evidence for third-order leakage.

Bivariate. In order to do a bivariate leakage detection test, we reduce the length of the power traces to 15 clock cycles and the sample rate of the oscilloscope to 200MS/s. Each trace then consists of 1 000 time samples. In order to reduce the signal-to-noise ratio, we make the traces DC free. We then combine the measurements at different time samples by doing an outer product of the centered traces with themselves. The resulting symmetric matrices are the samples for our t-test.

We first perform this experiment on the first-order protected AES implementation to verify if we can indeed detect bivariate leakage. The resulting t-statistic after 1 and 45 million traces is shown in Figure 11 and confirms that our method is sound.

Next, we do the same for the second-order masked AES implementation. We collect 50 million traces and show the resulting t-statistic in Figure 12. The result shows clearly that no bivariate leakage can be detected with 50 million traces.

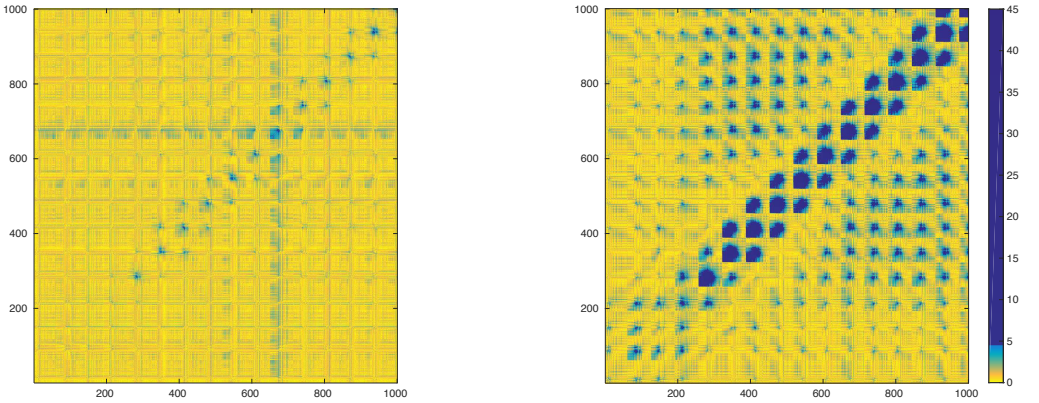

Figure 11: Non-specific bivariate leakage detection test on 15 clock cycles of a first-order protected AES. Left: 1 million traces. Right: 45 million traces.

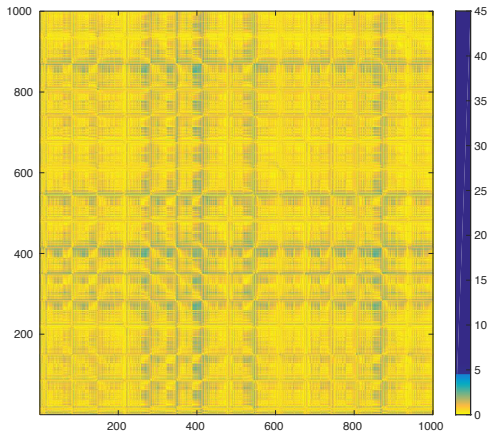

Figure 12: Non-specific bivariate leakage detection test on 15 clock cycles of a second-order protected AES with 50 million traces.

6 Implementation Cost

We presented first- and second-order secure constructions for AES and evaluated their security. In this section we investigate the implementation cost and compare it to the state-of-the-art AES designs of [CRB+16] and [GMK17]. All area measures were obtained with the Synopsis Design Compiler v.2013.12, using the Open Cell Nangate 45nm library [NAN] and are expressed in 2-input NAND gate equivalents[1]. We use compile option `-exact_map` to prevent optimization across modules. For a fair comparison, we also synthesize the implementations of [CRB+16] and [GMK17] with the same library and toolchain. From the latter, we picked the options for smallest area, *i.e.* not perfectly-interleaved and the eight-stage S-box. Both these works create a shared implementation from Canright's compact AES S-box [Can05] using the tower-field method. Our approach is thus radically different. We cannot compare easily with [UHA17] because of different synthesis libraries, though they seem to have a similar area footprint for larger randomness requirement (64 bits per S-box). Also, they only provide a first-order implementation. We first detail the cost of the S-box only in § 6.1 and then look at the entire AES encryption in § 6.2.

6.1 The S-box

Table 4: Implementation results for the AES S-box with Nangate 45nm Library

	First-order secure			Second-order secure		
Variant	Area	Randomness	Latency	Area	Randomness	Latency
Module	[GE]	[bits/S-box]	[cc]	[GE]	[bits/S-box]	[cc]
This work	**1 685**	**19**	**2** (+3)	**3 891**	**53**	**4** (+3)
Kronecker delta	259	3	(3)	629	13	(3)
Bool to Mult.	538	8	1	1 434	20	2
Inversion	226	-	-	226	-	-
Mult. to Bool	538	8	1	1 388	20	2
Others	124	-	-	214	-	-
[**CRB+16**]	2 348	54	6	4 744	162	6
[**GMK17**]	2 432	18	8	4 759	54	8

[1] One NAND gate is $0.798 \mu m^2$

Table 4 shows our implementation results for the S-box. Our S-box implementations are the smallest to date among state-of-the-art schemes with similar randomness and latency with an area reduction of 29% for first order and 18% for second order.

6.2 AES

Table 5 shows the implementation results of our entire AES implementations in comparison with those of De Cnudde *et al.* [CRB+16] and Gross *et al.* [GMK17]. Our S-box area reduction results in an overall improvement of around 10% over the state-of-the-art with comparable or even better randomness consumption and latency.

Table 5: Implementation results for AES-128 with Nangate 45nm Library

	First-order secure			Second-order secure		
Variant	Area	Randomness	Latency	Area	Randomness	Latency
Module	[GE]	[bits/S-box]	[cc]	[GE]	[bits/S-box]	[cc]
This work	**6 557**	**19**	**256**	**10 931**	**53**	**256**
S-box	1 685	-	-	3 891	-	-
State Array	2 509	-	-	3 728	-	-
Key Array	1 579	-	-	2 368	-	-
Control	208	-	-	199	-	-
Others	576	-	-	745	-	-
[**CRB+16**]	7 682	54	276	12 640	162	276
[**GMK17**]	7 337	18	246	12 024	54	246

7 Conclusion

We have ported the well-known concept of adaptively masking ciphers such as AES to hardware. The idea has been extensively studied in software, but had not yet been applied in hardware up till now. We show that this methodology is a very competitive alternative to state-of-the-art masked AES designs. Our approach is conceptually simple, yet incorporates modern countermeasures to mitigate the effect of glitches in hardware.

Specifically, we present secure circuits for converting between Boolean and multiplicative masking and for circumventing the well-known zero problem of multiplicative masking. We apply the methodology to the AES cipher for first- and second-order security and show with experiments that our implementations do not exhibit univariate or multivariate leakage with up to 50 million traces. Our AES S-box implementations require comparable randomness and latency to state-of-the-art implementations and yet achieve an 18 to 29% smaller chip area. We believe this is an interesting addition to the hardware designer's toolbox.

Acknowledgements

This work was supported in part by the NIST Research Grant 60NANB15D346. Oscar Reparaz and Begül Bilgin are postdoctoral fellows of the Fund for Scientific Research - Flanders (FWO) and Lauren De Meyer is funded by a PhD fellowship of the FWO. The authors would like to thank François-Xavier Standaert and Vincent Rijmen for helpful discussions.

References

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES, secure against some attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. 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 191–219, 2016.

[ANR17] Victor Arribas, Svetla Nikova, and Vincent Rijmen. VerMI: Verification tool for masked implementations. Cryptology ePrint Archive, Report 2017/1227, 2017.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.

[BCD+13] G. Becker, J. Cooper, E. De Mulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*, volume 1001, page 13, 2013.

[BGN+14a] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2014.

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

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

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

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

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.

[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.

[Cor17] Jean-Sébastien Coron. Checkmasks: Formal verification of side-channel countermeasures. Publicly available at https://github.com/coron/checkmasks, 2017.

[Cor18] Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2018.

[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.

[DBR18] Lauren De Meyer, Begül Bilgin, and Oscar Reparaz. Consolidating security notions in hardware masking. *IACR Cryptology ePrint Archive*, 2018:597, 2018.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014.

[DK10] Ivan Damgård and Marcel Keller. Secure multiparty AES. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected*

Papers, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2010.

[FGP+17] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults and the robust probing model. Cryptology ePrint Archive, Report 2017/711, 2017.

[FPS17] Sebastian Faust, Clara Paglialonga, and Tobias Schneider. Amortizing randomness complexity in private circuits. 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 781–810. Springer, 2017.

[FRR+10] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 135–156. Springer, 2010.

[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.

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

[GPQ10] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Secure multiplicative masking of power functions. In Jianying Zhou and Moti Yung, editors, *Applied Cryptography and Network Security, 8th International Conference, ACNS 2010, Beijing, China, June 22-25, 2010. Proceedings*, volume 6123 of *Lecture Notes in Computer Science*, pages 200–217, 2010.

[GPQ11a] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Montgomery's trick and fast implementation of masked AES. In Abderrahmane Nitaj and David Pointcheval, editors, *Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings*, volume 6737 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2011.

[GPQ11b] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Thwarting higher-order side channel analysis with additive and multiplicative maskings. In Preneel and Takagi [PT11], pages 240–255.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of KECCAK. In Hana Kubátová, Martin Novotný, and Amund Skavhaug, editors, *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, pages 205–212. IEEE, 2017.

[GT02] Jovan Dj. Golic and Christophe Tymen. Multiplicative masking and power analysis of AES. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2002.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006.

[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.

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

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

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Rao and Sunar [RS05], pages 157–171.

[NAN] NANGATE. The NanGate 45nm Open Cell Library. Available at http://www.nangate.com.

[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.

[PR11] Emmanuel Prouff and Thomas Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In Preneel and Takagi [PT11], pages 63–78.

[PT11] Bart Preneel and Tsuyoshi Takagi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.

[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.

[Rep16] Oscar Reparaz. Detecting flawed masking schemes with leakage detection tests. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2016.

[RGV17] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. 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 387–399. Springer, 2017.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. 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 413–427. Springer, 2010.

[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.

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

[UHA17] 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.

[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.

A On optimized second-order conversions

When adopting the conversion procedures described in § 3.1 for $d = 2$, an additional Boolean refreshing u is required to obtain second-order security (see Figure 3). Genelle *et al.* propose mask conversion procedures tailored for software implementations that aim at providing higher-order security [GPQ11b]. The conversions require a number of additive refreshing masks: $\frac{(d-1)d}{2}$ units for Boolean to Multiplicative and $\frac{d(d+1)}{2}$ for Multiplicative to Boolean. The authors suggest that one can ommit these extra refreshings when $d = 2$ and still maintain second-order security [GPQ11b, p. 246], both for Boolean to Multiplicative and vice-versa. Here we will see that the "optimized" variants exhibit second-order leaks and thus additional randomness is needed to achieve second-order security.

A.1 Boolean to Multiplicative

Following the basic recipe for converting three Boolean shares to multiplicative shares results in the circuit in Figure 13. The same conversion is initially proposed by Genelle *et al.*.

Consider the pair of intermediates (V_1, V_2) where $V_1 = (b_0 r_0) \oplus (b_1 r_0)$ and $V_2 = b_2$ (indicated by the red stars in Figure 13). We will see that the pair (V_1, V_2) jointly leak information on the sensitive input value x in the second statistical order.

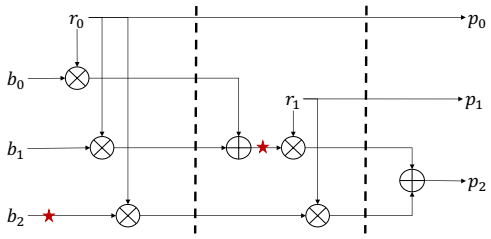

Figure 13: Conversion from Boolean to multiplicative masking with second-order leakage

To see this, consider the case when $V_1 = 0$. (This occurs with probability $\frac{1}{|\mathbb{F}_q|}$.) Then $b_0 \oplus b_1 = 0$ since $r_0 \neq 0$ by construction. This implies that the second intermediate $V_2 = b_2 = b_0 \oplus b_1 \oplus b_2$ leaks the sensitive value x.

As a result, the value $\mathbf{E}[L_1(V_1) \cdot L_2(V_2)|X = x]$ depends on the secret input x for any device leakage behavior functions L_1, L_2, including the Hamming weight leakage behavior functions. This can be verified with the following MATLAB script.

```
% including the value 0 in secret is not fair
% since the conversion Bool to Mult never sees
% masked 0 at the input
for secret = 1:255
  b0 = floor(field_size.*rand(1,number_traces));
  b1 = floor(field_size.*rand(1,number_traces));
  b2 = bitxor(bitxor(b0,b1),secret);
  r0 = zeros(1,number_traces);
  for i=1:number_traces
      while r0(i)==0
          r0(i) = floor(field_size.*rand(1,1));
      end
  end
  r0b0 = arrayfun(F,r0,b0);
  r0b1 = arrayfun(F,r0,b1);
```

```
  r0b2 = arrayfun(F,r0,b2);
  r0b0_p_r0b1 = bitxor(r0b0,r0b1);
  leak1 = (r0b0_p_r0b1==0); % ZV easier but not really needed
  leak2 = hw(1+b2)';
  second_order = mean((leak1 - mean(leak1)) .* (leak2 - mean(leak2)));
  fprintf('encoding secret %3d, cov(leak1,leak2)=% 2.5f\n', secret, second_order);
end
```

A.2 Multiplicative to Boolean

Consider the conversion from multiplicative to Boolean masking in Figure 14 without extra refreshing as proposed in [GPQ11b].

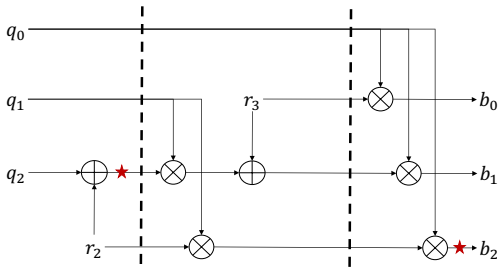

Figure 14: Conversion from multiplicative to Boolean masking with second-order leakage

The pair of intermediates (V_1, V_2) with $V_1 = q_2 \oplus r_2$ and $V_2 = q_0 q_1 r_2$ (as indicated by the red stars in Figure 14) leaks information in the second statistical order. For instance, whenever $V_1 = 0$, then $V_2 = q_0 q_1 q_2 = x$ reveals information on the sensitive variable x. The following MATLAB script shows that the second-order statistic $\mathbf{E}[L_1(V_1) \cdot L_2(V_2)|X = x]$ varies as a function of the secret x.

```
%%
% Second-order leak in Mult -> Bool
%
fprintf(' -- init\n');

% including the value 0 in secret is not fair
for secret = [1 2 95]%1:255 %1:255 % 1:255
    for rep=1:1
        clear leak1 leak2 second_order
        clear a b c r1 ab abr1 c_p_r1

        % non-zero a, b
        a = zeros(1,number_traces);
        b = zeros(1,number_traces);
        c = zeros(1,number_traces);
        r1 = zeros(1,number_traces);
        for i=1:number_traces
            while a(i)==0
                a(i) = floor(field_size.*rand(1,1));
            end
            while b(i)==0
                b(i) = floor(field_size.*rand(1,1));
            end
```

```
            while r1(i)==0
                r1(i) = floor(field_size.*rand(1,1));
            end
        end

        r1=uint8(r1);
        secret_times_a = arrayfun(F,uint8(secret*ones(1,number_traces)),inv_table(1+a)');
        c = arrayfun(F,secret_times_a,inv_table(1+b)');

        c_p_r1 = bitxor(c,r1);
        ab = arrayfun(F,a,b);
        abr1 = arrayfun(F,ab,r1);

        leak1 = hw(1+c_p_r1);
        leak2 = hw(1+abr1);



        second_order = mean((leak1 - mean(leak1)) .* (leak2 - mean(leak2)));

        fprintf('encoding secret %3d, cov(leak1,leak2)=% 2.5f\n', secret, second_order);
    end
end
```

B Inversion circuit

The AES S-box circuit from Boyar, Matthews and Peralta [BMP13] is the smallest to date, even beating Canright's tower-field one. The circuit consists of three parts: $S = B \cdot F \cdot U \oplus$ `0x63` with U, B linear and F non-linear. As we are only interested in the inversion part of the S-box, we adopt only F and U and add our own linear layer to obtain the inversion output $x_0^{-1}, x_1^{-1}, \ldots, x_7^{-1}$. We provide only the linear equations of the new block here. For F and U we refer to [BMP13, Fig. 10 and 11].

$$
\begin{aligned}
x_0^{-1} &= z_9 \oplus z_{11} \oplus z_{15} \oplus z_{17} \\
x_1^{-1} &= z_3 \oplus z_4 \oplus z_6 \oplus z_7 \oplus z_{12} \oplus z_{13} \oplus z_{15} \oplus z_{16} \\
x_2^{-1} &= z_0 \oplus z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_8 \oplus z_{12} \oplus z_{13} \oplus z_{15} \oplus z_{16} \\
x_3^{-1} &= z_0 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \oplus z_{10} \oplus z_{11} \oplus z_{12} \oplus z_{13} \oplus z_{15} \oplus z_{17} \\
x_4^{-1} &= z_0 \oplus z_2 \oplus z_6 \oplus z_8 \oplus z_{12} \oplus z_{13} \oplus z_{15} \oplus z_{16} \\
x_5^{-1} &= z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_6 \oplus z_8 \oplus z_{10} \oplus z_{11} \oplus z_{12} \oplus z_{14} \oplus z_{15} \oplus z_{16} \\
x_6^{-1} &= z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_6 \oplus z_8 \oplus z_9 \oplus z_{10} \oplus z_{13} \oplus z_{14} \oplus z_{15} \oplus z_{17} \\
x_7^{-1} &= z_3 \oplus z_5 \oplus z_6 \oplus z_8 \oplus z_{12} \oplus z_{13} \oplus z_{15} \oplus z_{16}
\end{aligned}
$$

C Probability Distributions of Probes

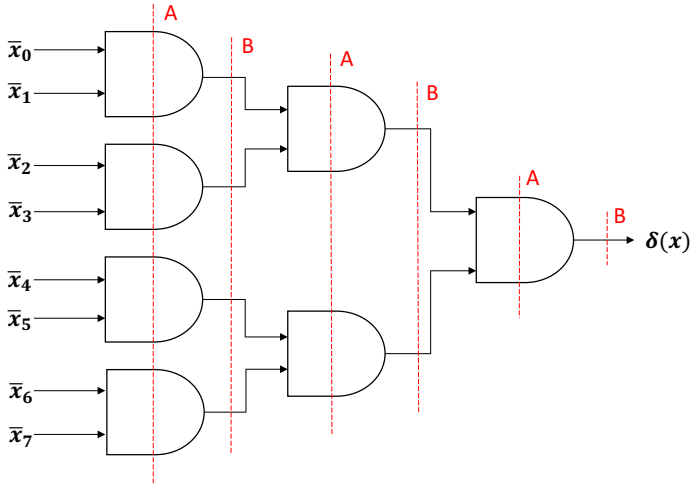

Figure 15: Circuit for the shared Kronecker delta function $\delta(x)$ for $n = 8$

In Figure 15 we show again the AND tree that implements the shared Kronecker Delta function with randomness optimizations from § 3.3 and we indicate with red dotted lines the stages where we place our probes. At each probe, we compute the probability distribution of the wire for each possible value of the secret x and verify that the distribution does not vary with the secret. We do the same for each pair of probes in the case of the second-order implementation. We distinguish A stages, in which we target the cross products t_{ij} of the DOM multipliers and B stages, which contain the multiplication results. Note that the A-stage probes are the cross products *before* any randomness is added.

One Probe. If we look only at individual probes (first-order) in either the first- or second-order implementation, we find that all B-stage wires are uniformly distributed for each secret. For each of the cross products in the A stages, we find a non-uniform distribution $[\frac{3}{4}\ \frac{1}{4}]$. However, this distribution does not change if we vary the secret.

Two Probes. In the second-order implementation, pairs of probes in the B stages also result in uniform distributions $[\frac{1}{4}\ \frac{1}{4}\ \frac{1}{4}\ \frac{1}{4}]$. In A stages we see the distribution $[\frac{9}{16}\ \frac{3}{16}\ \frac{3}{16}\ \frac{1}{16}]$ for most pairs. Since this is the outer product of $[\frac{3}{4}\ \frac{1}{4}]$ with itself, it means such a pair of probes is statistically independent. In contrast, let $i \neq j$, $j \neq k$ and $i \neq k$; then when we probe two cross products (t_{ij}, t_{ik}) or (t_{ij}, t_{kj}) in the same multiplier, we obtain the probability distribution $[\frac{5}{8}\ \frac{1}{8}\ \frac{1}{8}\ \frac{1}{8}]$.

The multivariate probe of a B-stage wire and a wire in the next A stage results in distributions $[\frac{3}{8}\ \frac{3}{8}\ \frac{1}{8}\ \frac{1}{8}]$ (the outer product of $[\frac{3}{4}\ \frac{1}{4}]$ and $[\frac{1}{2}\ \frac{1}{2}]$), except when we combine a cross product t_{ij} with share i or j of one of the multiplication inputs. In those cases, we see probability distribution $[\frac{1}{2}\ \frac{1}{4}\ 0\ \frac{1}{4}]$. Again, these distributions are not uniform but they are independent of the secret.

D Strong Non-Interference of Conversions

In this section, we prove the strong non-interference of the conversions between Boolean and multiplicative masking. We cannot use the tool of [Cor17] since it is incompatible with the use of our multiplicative operations. An important substitution rule from [Cor18] is that an XOR with a random $r_i \xleftarrow{\$} \mathbb{F}_q$ serves as a one-time pad when r_i is not used in another part of the probe:

$$r_i \oplus x \to r_i$$

However, extending this substitution rule to field multiplication is not straight-forward. In general, the multiplication of a secret field element $x \in \mathbb{F}_q$ with a random variable r_i cannot be simulated by r_i because of the non-uniform mapping of zeroes in a multiplication. However, if at least one of the multiplicants is nonzero, the random value *does* play the role of a one-time pad. Therefore, we define and use the following substitution rule:

$$r_i \otimes x \to r_i \qquad \text{iff } x \in \mathbb{F}_q^*$$

This rule is valid whether $r_i \xleftarrow{\$} \mathbb{F}_q^*$ or $r_i \xleftarrow{\$} \mathbb{F}_q$. In what follows, we show how to simulate all d-probes in the conversion circuits using only $|\mathcal{I}|$ input shares, where $\mathcal{I}$ is the set of intermediate probes. It can be seen that for any field multiplication, at least one of the operands is nonzero in our setting. We thus show that the conversions are d-SNI for $d \in \{1, 2\}$. Table 6 shows the proof for $d = 1$ (Figure 16) and Tables 7 and 8 for $d = 2$ (Figure 17). For readability, we do not attempt to simulate when the probe(s) themselves already depend on only $|\mathcal{I}|$ input shares.

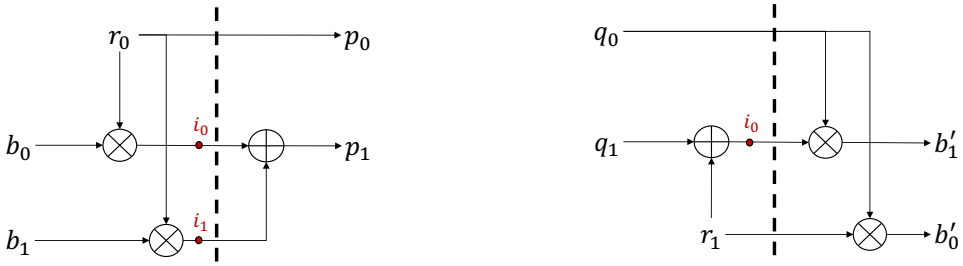

Figure 16: First-order Boolean to multiplicative (left) and multiplicative to Boolean (right) conversion circuits with intermediate probes

Table 6: Simulation of intermediate probes $\mathcal{I}$ and output probes $\mathcal{O}$ such that $|\mathcal{I}| + |\mathcal{O}| \leq d = 1$ using $|\mathcal{I}|$ input shares for the first-order conversions.

$\lvert\mathcal{I}\rvert$	**Probes**	**Simulation**	**using**
Multiplicative to Boolean: $q_0, q_1 \in \mathbb{F}_q^*, r_1 \xleftarrow{\$} \mathbb{F}_q$			
0	$b'_0 = r_1 q_0$	$\sim r_1$	
	$b'_1 = (q_1 \oplus r_1) q_0$	$\sim r_1 q_0 \sim r_1$	
1	$i_0 = q_1 \oplus r_1$		q_1
Boolean to multiplicative: $b \in \mathbb{F}_q^*, r_0 \xleftarrow{\$} \mathbb{F}_q^*$			
0	$p_0 = r_0$		
	$p_1 = r_0 b$	$\sim r_0$	
1	$i_0 = r_0 b_0$		b_0
	$i_1 = r_0 b_1$		b_1

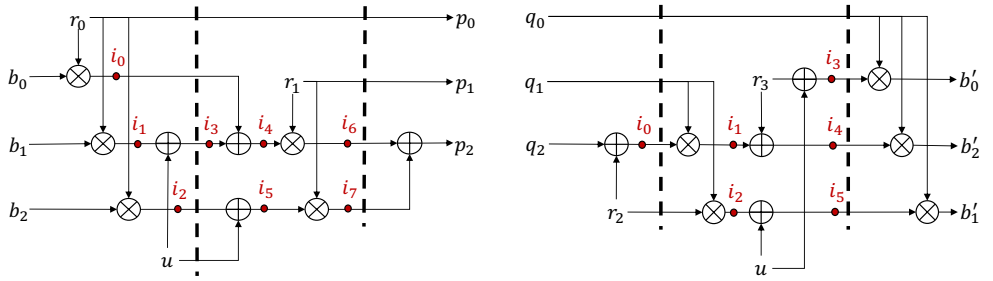

Figure 17: Second-order Boolean to multiplicative (left) and multiplicative to Boolean (right) conversion circuits with intermediate probes

Table 7: Simulation of intermediate probes $\mathcal{I}$ and output probes $\mathcal{O}$ such that $|\mathcal{I}| + |\mathcal{O}| \leq d = 2$ using $|\mathcal{I}|$ input shares for the second-order multiplicative to Boolean conversion.

$\lvert\mathcal{I}\rvert$	**Probes**	**Simulation using**	
	Multiplicative to Boolean: $q_0, q_1, q_2 \in \mathbb{F}_q^*, r_2, r_3, u \xleftarrow{\$} \mathbb{F}_q$		
0	$(b'_0, b'_1) = ((r_3 \oplus u)q_0, (r_2q_1 \oplus u)q_0)$	$\sim (r_3q_0, (r_2q_1 \oplus u)q_0) \sim (r_3q_0, uq_0) \sim (r_3, u)$	
	$(b'_0, b'_2) = ((r_3 \oplus u)q_0, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (uq_0, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	
		$\sim (u, r_3q_0) \sim (u, r_3)$	
	$(b'_1, b'_2) = ((r_2q_1 \oplus u)q_0, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (uq_0, r_3q_0) \sim (u, r_3)$	
1	$(i_0, b'_0) = (q_2 \oplus r_2, (r_3 \oplus u)q_0)$	$\sim (r_2, r_3)$	
	$(i_1, b'_0) = ((q_2 \oplus r_2)q_1, (r_3 \oplus u)q_0)$	$\sim (r_2q_1, r_3q_0) \sim (r_2, r_3)$	
	$(i_2, b'_0) = (r_2q_1, (r_3 \oplus u)q_0)$	$\sim (r_2, r_3)$	
	$(i_3, b'_0) = (r_3 \oplus u, (r_3 \oplus u)q_0)$		q_0
	$(i_4, b'_0) = ((q_2 \oplus r_2)q_1 \oplus r_3, (r_3 \oplus u)q_0)$	$\sim ((q_2 \oplus r_2)q_1 \oplus r_3, uq_0) \sim (r_3, u)$	
	$(i_5, b'_0) = (r_2q_1 \oplus u, (r_3 \oplus u)q_0)$	$\sim (r_2q_1 \oplus u, r_3q_0) \sim (u, r_3)$	
	$(i_0, b'_1) = (q_2 \oplus r_2, (r_2q_1 \oplus u)q_0)$	$\sim (q_2 \oplus r_2, uq_0) \sim (r_2, u)$	
	$(i_1, b'_1) = ((q_2 \oplus r_2)q_1, (r_2q_1 \oplus u)q_0)$	$\sim ((q_2 \oplus r_2)q_1, uq_0) \sim (r_2q_1, uq_0) \sim (r_2, u)$	
	$(i_2, b'_1) = (r_2q_1, (r_2q_1 \oplus u)q_0)$	$\sim (r_2q_1, uq_0) \sim (r_2, u)$	
	$(i_3, b'_1) = (r_3 \oplus u, (r_2q_1 \oplus u)q_0)$	$\sim (r_3, (r_2q_1 \oplus u)q_0) \sim (r_3, uq_0) \sim (r_3, u)$	
	$(i_4, b'_1) = ((q_2 \oplus r_2)q_1 \oplus r_3, (r_2q_1 \oplus u)q_0)$	$\sim (r_3, (r_2q_1 \oplus u)q_0) \sim (r_3, uq_0) \sim (r_3, u)$	
	$(i_5, b'_1) = (r_2q_1 \oplus u, (r_2q_1 \oplus u)q_0)$	$\sim (u, uq_0)$	q_0
	$(i_0, b'_2) = (q_2 \oplus r_2, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (q_2 \oplus r_2, r_3q_0) \sim (r_2, r_3)$	
	$(i_1, b'_2) = ((q_2 \oplus r_2)q_1, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim ((q_2 \oplus r_2)q_1, r_3q_0) \sim (r_2q_1, r_3q_0) \sim (r_2, r_3)$	
	$(i_2, b'_2) = (r_2q_1, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (r_2q_1, r_3q_0) \sim (r_2, r_3)$	
	$(i_3, b'_2) = (r_3 \oplus u, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (u, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	
		$\sim (u, r_3q_0) \sim (u, r_3)$	
	$(i_4, b'_2) = ((q_2 \oplus r_2)q_1 \oplus r_3, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (r_3, r_3q_0)$	q_0
	$(i_5, b'_2) = (r_2q_1 \oplus u, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	$\sim (u, ((q_2 \oplus r_2)q_1 \oplus r_3)q_0)$	
		$\sim (u, r_3q_0) \sim (u, r_3)$	
2	$(i_0, i_1) = (q_2 \oplus r_2, (q_2 \oplus r_2)q_1)$		q_1, q_2
	$(i_0, i_2) = (q_2 \oplus r_2, r_2q_1)$		q_1, q_2
	$(i_0, i_3) = (q_2 \oplus r_2, r_3 \oplus u)$		q_2
	$(i_0, i_4) = (q_2 \oplus r_2, (q_2 \oplus r_2)q_1 \oplus r_3)$		q_1, q_2
	$(i_0, i_5) = (q_2 \oplus r_2, r_2q_1 \oplus u)$		q_1, q_2
	$(i_1, i_2) = ((q_2 \oplus r_2)q_1, r_2q_1)$		q_1, q_2
	$(i_1, i_3) = ((q_2 \oplus r_2)q_1, r_3 \oplus u)$		q_1, q_2
	$(i_1, i_4) = ((q_2 \oplus r_2)q_1, (q_2 \oplus r_2)q_1 \oplus r_3)$		q_1, q_2
	$(i_1, i_5) = ((q_2 \oplus r_2)q_1, r_2q_1 \oplus u)$		q_1, q_2
	$(i_2, i_3) = (r_2q_1, r_3 \oplus u)$		q_1
	$(i_2, i_4) = (r_2q_1, (q_2 \oplus r_2)q_1 \oplus r_3)$		q_1, q_2
	$(i_2, i_5) = (r_2q_1, r_2q_1 \oplus u)$		q_1
	$(i_3, i_4) = (r_3 \oplus u, (q_2 \oplus r_2)q_1 \oplus r_3)$		q_1, q_2
	$(i_3, i_5) = (r_3 \oplus u, r_2q_1 \oplus u)$		q_1
	$(i_4, i_5) = ((q_2 \oplus r_2)q_1 \oplus r_3, r_2q_1 \oplus u)$		q_1, q_2

Table 8: Simulation of intermediate probes $\mathcal{I}$ and output probes $\mathcal{O}$ such that $|\mathcal{I}| + |\mathcal{O}| \leq d = 2$ using $|\mathcal{I}|$ input shares for the second-order Boolean to multiplicative conversion.

$\lvert\mathcal{I}\rvert$	**Probes**	**Simulation**	**using**
	Boolean to multiplicative: $b \in \mathbb{F}_q^*, r_0, r_1 \xleftarrow{\$} \mathbb{F}_q^*, u \xleftarrow{\$} \mathbb{F}_q$		
0	$(p_0, p_1) = (r_0, r_1)$		
	$(p_0, p_2) = (r_0, r_1 r_0 b)$	$\sim (r_0, r_1)$	
	$(p_1, p_2) = (r_1, r_1 r_0 b)$	$\sim (r_1, r_0)$	
1	$(i_0, p_0) = (r_0 b_0, r_0)$		b_0
	$(i_1, p_0) = (r_0 b_1, r_0)$		b_1
	$(i_2, p_0) = (r_0 b_2, r_0)$		b_2
	$(i_3, p_0) = (r_0 b_1 \oplus u, r_0)$		b_1
	$(i_4, p_0) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_0)$	$\sim (u, r_0)$	
	$(i_5, p_0) = (r_0 b_2 \oplus u, r_0)$		b_2
	$(i_6, p_0) = (r_1(r_0 b_0 \oplus r_0 b_1 \oplus u), r_0)$	$\sim (r_1 u, r_0)$	
	$(i_7, p_0) = (r_1(r_0 b_2 \oplus u), r_0)$		b_2
	$(i_0, p_1) = (r_0 b_0, r_1)$		b_0
	$(i_1, p_1) = (r_0 b_1, r_1)$		b_1
	$(i_2, p_1) = (r_0 b_2, r_1)$		b_2
	$(i_3, p_1) = (r_0 b_1 \oplus u, r_1)$		b_1
	$(i_4, p_1) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_1)$	$\sim (u, r_1)$	
	$(i_5, p_1) = (r_0 b_2 \oplus u, r_1)$		b_2
	$(i_6, p_1) = (r_1(r_0 b_0 \oplus r_0 b_1 \oplus u), r_1)$	$\sim (r_1 u, r_1)$	
	$(i_7, p_1) = (r_1(r_0 b_2 \oplus u), r_1)$		b_2
	$(i_0, p_2) = (r_0 b_0, r_1 r_0 b)$	$\sim (r_0 b_0, r_1)$	b_0
	$(i_1, p_2) = (r_0 b_1, r_1 r_0 b)$	$\sim (r_0 b_1, r_1)$	b_1
	$(i_2, p_2) = (r_0 b_2, r_1 r_0 b)$	$\sim (r_0 b_2, r_1)$	b_2
	$(i_3, p_2) = (r_0 b_1 \oplus u, r_1 r_0 b)$	$\sim (u, r_1)$	
	$(i_4, p_2) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_1 r_0 b)$	$\sim (u, r_1)$	
	$(i_5, p_2) = (r_0 b_2 \oplus u, r_1 r_0 b)$	$\sim (u, r_1)$	
	$(i_6, p_2) = (r_1(r_0 b_0 \oplus r_0 b_1 \oplus u), r_1 r_0 b)$	$\sim (r_1 u, r_1 r_0 b) \sim (r_1 u, r_0)$	
	$(i_7, p_2) = (r_1(r_0 b_2 \oplus u), r_1 r_0 b)$	$\sim (r_1 u, r_1 r_0 b) \sim (r_1 u, r_0)$	
2	$(i_0, i_1) = (r_0 b_0, r_0 b_1)$		b_0, b_1
	$(i_0, i_2) = (r_0 b_0, r_0 b_2)$		b_0, b_2
	$(i_0, i_3) = (r_0 b_0, r_0 b_1 \oplus u)$		b_0, b_1
	$(i_0, i_4) = (r_0 b_0, r_0 b_0 \oplus r_0 b_1 \oplus u)$		b_0, b_1
	$(i_0, i_5) = (r_0 b_0, r_0 b_2 \oplus u)$		b_0, b_2
	$(i_0, i_6) = (r_0 b_0, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$		b_0, b_1
	$(i_0, i_7) = (r_0 b_0, r_1(r_0 b_2 \oplus u))$		b_0, b_2
	$(i_1, i_2) = (r_0 b_1, r_0 b_2)$		b_1, b_2
	$(i_1, i_3) = (r_0 b_1, r_0 b_1 \oplus u)$		b_1
	$(i_1, i_4) = (r_0 b_1, r_0 b_0 \oplus r_0 b_1 \oplus u)$		b_0, b_1
	$(i_1, i_5) = (r_0 b_1, r_0 b_2 \oplus u)$		b_1, b_2
	$(i_1, i_6) = (r_0 b_1, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$		b_0, b_1
	$(i_1, i_7) = (r_0 b_1, r_1(r_0 b_2 \oplus u))$		b_1, b_2
	$(i_2, i_3) = (r_0 b_2, r_0 b_1 \oplus u)$		b_1, b_2
	$(i_2, i_4) = (r_0 b_2, r_0 b_0 \oplus r_0 b_1 \oplus u)$	$\sim (r_0 b_2, u)$	b_2
	$(i_2, i_5) = (r_0 b_2, r_0 b_2 \oplus u)$		b_2
	$(i_2, i_6) = (r_0 b_2, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$	$\sim (r_0 b_2, r_1 u)$	b_2
	$(i_2, i_7) = (r_0 b_2, r_1(r_0 b_2 \oplus u))$		b_2
	$(i_3, i_4) = (r_0 b_1 \oplus u, r_0 b_0 \oplus r_0 b_1 \oplus u)$		b_0, b_1
	$(i_3, i_5) = (r_0 b_1 \oplus u, r_0 b_2 \oplus u)$		b_1, b_2
	$(i_3, i_6) = (r_0 b_1 \oplus u, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$		b_0, b_1
	$(i_3, i_7) = (r_0 b_1 \oplus u, r_1(r_0 b_2 \oplus u))$		b_1, b_2
	$(i_4, i_5) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_0 b_2 \oplus u)$	$\sim (r_0 \oplus u, u)$	
	$(i_4, i_6) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$		b_0, b_1
	$(i_4, i_7) = (r_0 b_0 \oplus r_0 b_1 \oplus u, r_1(r_0 b_2 \oplus u))$	$\sim (r_0 \oplus u, r_1 u)$	
	$(i_5, i_6) = (r_0 b_2 \oplus u, r_1(r_0 b_0 \oplus r_0 b_1 \oplus u))$	$\sim (r_0 \oplus u, r_1 u)$	
	$(i_5, i_7) = (r_0 b_2 \oplus u, r_1(r_0 b_2 \oplus u))$		b_2
	$(i_6, i_7) = (r_1(r_0 b_0 \oplus r_0 b_1 \oplus u), r_1(r_0 b_2 \oplus u))$	$\sim (r_1(r_0 \oplus u), r_1 u)$	

E Strong Non-Interference of Kronecker Delta

The Kronecker Delta circuit (Figure 4) is d-SNI for $d \in \{1, 2\}$ because the used multiplier gates are SNI. This can be verified with the tool of Coron [Cor17]. However, the recycling of randomness throughout the circuit might break the SNI property. In this section, we show that the first- and second-order Kronecker delta circuits (detailed in Figure 18) are 1-SNI resp. 2-SNI even with optimized randomness. We note that we have a pen-and-paper proof instead of using an existing tool. That is because existing tools, including the recent tool of Coron [Cor17] which uses the aforementioned one-time pad rules, are not able to reduce the probes to the point of simulatibility. More specifically, we are able to successfully simulate any d-probes provided to us by the tool [Cor17] as breaking the d-SNI condition. We observe two aspects that lead to false negatives and should be investigated further to improve [Cor17]. On the one hand, the randomness recycling limits the substitutions that the tool thinks it is able to make. On the other hand, the tool does not simplify the equations of specific probes and therefore does not see that some of the used random bits disappear.

We demonstrate that all (pairs of) probes can be simulated using at most $|\mathcal{I}|$ input shares. We proceed by reducing all (pairs of) probes using the one-time pad method of [Cor18] to a point, where only randomness remains and thus simulation is trivial. We recall that any variable that is the XOR of a value with a uniformly random r_i can be replaced by r_i if r_i does not appear anywhere else in the current probes.

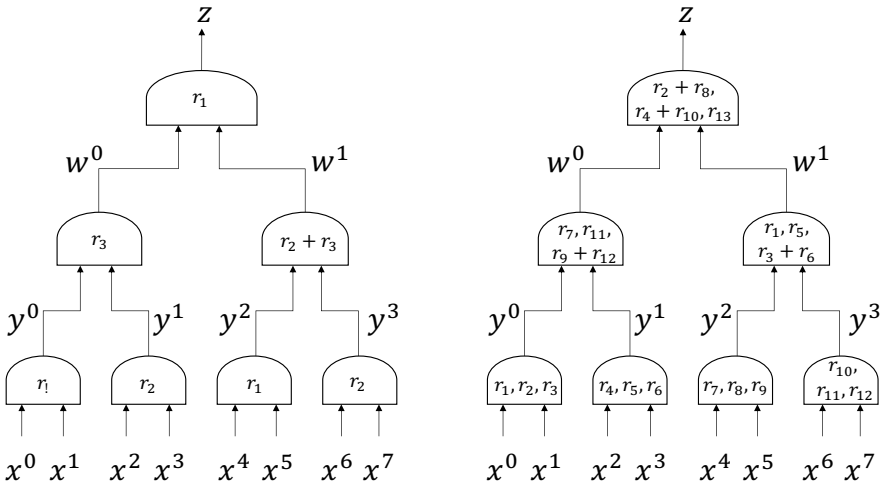

Figure 18: First- (left) and second-order (right) Kronecker Delta. Notation is adapted for clarity.

For the first-order circuit, we can easily exhaust all intermediates as follows. We skip the crossproducts in the first layer, since they can obviously be simulated using one input share. The equations demonstrate clearly the special property of the DOM multiplier that was explained in § 3.3, resulting in the independence of for example w_i^0 of r_2.

For $i \in \{0, 1\}$:

$$\begin{aligned} y_i^0 &= x_i^0 x^1 + r_1 && \sim r_1 \\ y_i^1 &= x_i^2 x^3 + r_2 && \sim r_2 \\ y_i^2 &= x_i^4 x^5 + r_1 && \sim r_1 \\ y_i^3 &= x_i^6 x^7 + r_2 && \sim r_2 \end{aligned}$$

For $i,j \in \{0,1\}$:

$$y_i^0 y_j^1 = (x_i^0 x^1 + r_1)(x_j^2 x^3 + r_2) \qquad \sim r_1 r_2$$
$$y_i^2 y_j^3 = (x_i^4 x^5 + r_1)(x_j^6 x^7 + r_2) \qquad \sim r_1 r_2$$

For $i,j \in \{0,1\}, i \neq j$:

$$y_i^0 y_j^1 + r_3 = (x_i^0 x^1 + r_1)(x_j^2 x^3 + r_2) + r_3 \qquad \sim r_3$$
$$y_i^2 y_j^3 + (r_2 + r_3) = (x_i^4 x^5 + r_1)(x_j^6 x^7 + r_2) + r_2 + r_3 \qquad \sim r_3$$

For $i \in \{0,1\}$:

$$w_i^0 = y_i^0 y^1 + r_3 = (x_i^0 x^1 + r_1) y^1 + r_3 \qquad \sim r_3$$
$$w_i^1 = y_i^2 y^3 + r_2 + r_3 = (x_i^4 x^5 + r_1) y^3 + r_2 + r_3 \qquad \sim r_2$$

For $i,j \in \{0,1\}$:

$$w_i^0 w_j^1 = ((x_i^0 x^1 + r_1) y^1 + r_3)((x_j^4 x^5 + r_1) y^3 + r_2 + r_3) \qquad \sim r_3 r_2$$

For $i,j \in \{0,1\}, i \neq j$:

$$w_i^0 w_j^1 + r_1 = ((x_i^0 x^1 + r_1) y^1 + r_3)((x_j^4 x^5 + r_1) y^3 + r_2 + r_3) + r_1 \sim r_3 r_2 + r_1 \qquad \sim r_1$$

For $i \in \{0,1\}$:

$$z_i^0 = w_i^0 w^1 + r_1 = ((x_i^0 x^1 + r_1) y^1 + r_3) w^1 + r_1 \sim r_3 w^1 + r_1 \qquad \sim r_1$$

We now prove that the second-order circuit is 2-SNI. Here also, some of the used random variables disappear. We first note all intermediates of the circuit below to clarify which intermediate depends on which r_i. We skip the crossproducts in the first layer, since their equations and dependencies are obvious.

$$y_0^0 = x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}$$
$$y_1^0 = x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}$$
$$y_2^0 = x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}$$
$$y_0^1 = x_0^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_5}$$
$$y_1^1 = x_1^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_6}$$
$$y_2^1 = x_2^2 x^3 + \boldsymbol{r_5} + \boldsymbol{r_6}$$
$$y_0^2 = x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8}$$
$$y_1^2 = x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9}$$
$$y_2^2 = x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9}$$
$$y_0^3 = x_0^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{11}}$$
$$y_1^3 = x_1^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{12}}$$
$$y_2^3 = x_2^6 x^7 + \boldsymbol{r_{11}} + \boldsymbol{r_{12}}$$

$$y_0^0 y_0^1 = (x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2})(x_0^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_5})$$
$$y_0^0 y_1^1 + r_7 = (x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2})(x_1^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_6}) + \boldsymbol{r_7}$$
$$y_0^0 y_2^1 + r_{11} = (x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2})(x_2^2 x^3 + \boldsymbol{r_5} + \boldsymbol{r_6}) + \boldsymbol{r_{11}}$$
$$y_1^0 y_0^1 + r_7 = (x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3})(x_0^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_5}) + \boldsymbol{r_7}$$
$$y_1^0 y_1^1 = (x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3})(x_1^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_6})$$
$$y_1^0 y_2^1 + (r_9 + r_{12}) = (x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3})(x_2^2 x^3 + \boldsymbol{r_5} + \boldsymbol{r_6}) + \boldsymbol{r_9} + \boldsymbol{r_{12}}$$
$$y_2^0 y_0^1 + r_{11} = (x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3})(x_0^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_5}) + \boldsymbol{r_{11}}$$
$$y_2^0 y_1^1 + (r_9 + r_{12}) = (x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3})(x_1^2 x^3 + \boldsymbol{r_4} + \boldsymbol{r_6}) + \boldsymbol{r_9} + \boldsymbol{r_{12}}$$
$$y_2^0 y_2^1 = (x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3})(x_2^2 x^3 + \boldsymbol{r_5} + \boldsymbol{r_6})$$

$$y_0^2 y_0^3 = (x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8})(x_0^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{11}})$$
$$y_0^2 y_1^3 + r_1 = (x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8})(x_1^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{12}}) + \boldsymbol{r_1}$$
$$y_0^2 y_2^3 + r_5 = (x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8})(x_2^6 x^7 + \boldsymbol{r_{11}} + \boldsymbol{r_{12}}) + \boldsymbol{r_5}$$

$$
\begin{aligned}
y_1^2 y_0^3 + r_1 &= (x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9})(x_0^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{11}}) + \boldsymbol{r_1} \\
y_1^2 y_1^3 &= (x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9})(x_1^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{12}}) \\
y_1^2 y_2^3 + (r_3 + r_6) &= (x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9})(x_2^6 x^7 + \boldsymbol{r_{11}} + \boldsymbol{r_{12}}) + \boldsymbol{r_3} + \boldsymbol{r_6} \\
y_2^2 y_0^3 + r_5 &= (x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9})(x_0^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{11}}) + \boldsymbol{r_5} \\
y_2^2 y_1^3 + (r_3 + r_6) &= (x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9})(x_1^6 x^7 + \boldsymbol{r_{10}} + \boldsymbol{r_{12}}) + \boldsymbol{r_3} + \boldsymbol{r_6} \\
y_2^2 y_2^3 &= (x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9})(x_2^6 x^7 + \boldsymbol{r_{11}} + \boldsymbol{r_{12}})
\end{aligned}
$$

$$
\begin{aligned}
w_0^0 &= (x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_{11}} \\
w_1^0 &= (x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_9} + \boldsymbol{r_{12}} \\
w_2^0 &= (x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_{11}} + \boldsymbol{r_9} + \boldsymbol{r_{12}} \\
w_0^1 &= (x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_5} \\
w_1^1 &= (x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_3} + \boldsymbol{r_6} \\
w_2^1 &= (x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_5} + \boldsymbol{r_3} + \boldsymbol{r_6}
\end{aligned}
$$

$$
\begin{aligned}
w_0^0 w_0^1 &= ((x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_{11}})((x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_5}) \\
w_0^0 w_1^1 + (r_2 + r_8) &= ((x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_{11}})((x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_3} + \boldsymbol{r_6}) + \boldsymbol{r_2} + \boldsymbol{r_8} \\
w_0^0 w_2^1 + (r_4 + r_{10}) &= ((x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_{11}})((x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_5} + \boldsymbol{r_3} + \boldsymbol{r_6}) + \boldsymbol{r_4} + \boldsymbol{r_{10}} \\
w_1^0 w_0^1 + (r_2 + r_8) &= ((x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_5}) + \boldsymbol{r_2} + \boldsymbol{r_8} \\
w_1^0 w_1^1 &= ((x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_3} + \boldsymbol{r_6}) \\
w_1^0 w_2^1 + r_{13} &= ((x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_5} + \boldsymbol{r_3} + \boldsymbol{r_6}) + \boldsymbol{r_{13}} \\
w_2^0 w_0^1 + (r_4 + r_{10}) &= ((x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_{11}} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_0^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_8}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_5}) + \boldsymbol{r_4} + \boldsymbol{r_{10}} \\
w_2^0 w_1^1 + r_{13} &= ((x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_{11}} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_1^4 x^5 + \boldsymbol{r_7} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_1} + \boldsymbol{r_3} + \boldsymbol{r_6}) + \boldsymbol{r_{13}} \\
w_2^0 w_2^1 &= ((x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_{11}} + \boldsymbol{r_9} + \boldsymbol{r_{12}})((x_2^4 x^5 + \boldsymbol{r_8} + \boldsymbol{r_9}) y^3 + \boldsymbol{r_5} + \boldsymbol{r_3} + \boldsymbol{r_6})
\end{aligned}
$$

$$
\begin{aligned}
z_0 &= ((x_0^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_2}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_{11}}) w^1 + \boldsymbol{r_2} + \boldsymbol{r_8} + \boldsymbol{r_4} + \boldsymbol{r_{10}} \\
z_1 &= ((x_1^0 x^1 + \boldsymbol{r_1} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_7} + \boldsymbol{r_9} + \boldsymbol{r_{12}}) w^1 + \boldsymbol{r_2} + \boldsymbol{r_8} + \boldsymbol{r_{13}} \\
z_2 &= ((x_2^0 x^1 + \boldsymbol{r_2} + \boldsymbol{r_3}) y^1 + \boldsymbol{r_{11}} + \boldsymbol{r_9} + \boldsymbol{r_{12}}) w^1 + \boldsymbol{r_4} + \boldsymbol{r_{10}} + \boldsymbol{r_{13}}
\end{aligned}
$$

E.1 $|\mathcal{I}| = 0$

Simulating two outputprobes is trivial as each outputshare has at least one pad (r_i) that does not appear in the other outputshares:

$$
\begin{aligned}
(z_0, z_1) &\sim (r_{10}, r_{13}) \\
(z_0, z_2) &\sim (r_8, r_{13}) \\
(z_1, z_2) &\sim (r_8, r_{10})
\end{aligned}
$$

E.2 $|\mathcal{I}| = 1$

We combine each outputprobe with an intermediate probe.

Outputprobe z_0. z_0 receives a one-time pad from r_8, r_4 and r_{10}. A combination of this probe with any intermediate that is independent of at least one of these is trivial to simulate since we can then replace z_0 by the pad. For example, $w_2^0 w_2^1$ does not depend on r_4, so we have

$$(z_0, w_2^0 w_2^1) \sim (r_4, w_2^0 w_2^1) \sim (r_4, r_5)$$

We therefore only consider the intermediates that depend on r_8, r_4 *and* r_{10}.

$$
\begin{aligned}
(z_0, w_2^0 w_0^1 + r_4 + r_{10}) &\sim (z_0, w_2^0 r_5 + r_4 + r_{10}) \\
&\sim (r_8, w_2^0 r_5 + r_4 + r_{10})
\end{aligned}
$$

$$\sim (r_8, r_{10})$$

In the first step, we use the fact that z_0 and w_2^0 are independent of r_5 and replace w_0^1 by its one-time pad r_5. In the second step, we use the fact that w_2^0 does not depend on r_8 and use this to replace z_0. Finally, we can replace the second probe by the pad r_{10}. A similar method can be applied to the pair of probes $(z_0, w_0^0 w_2^1 + r_4 + r_{10})$. All other intermediates are independent of either r_4, r_8 of r_{10}.

Outputprobes z_1 and z_2. Thanks to the fresh randomness r_{13}, outputshares z_1 and z_2 are trivial to combine with any intermediates. For z_1, there is no intermediate which depends on r_2, r_8 *and* r_{13} and similarly, for z_2 all intermediates have independence of either r_4, r_{10} or r_{13}.

E.3 $|\mathcal{I}| = 2$

There are a lot of pairs of intermediates in the circuit and enumerating them all would require many pages. We therefore give examples of all types of pairings. The methodology and results for the others are extremely similar. We divide this section based on the type of the first probe. We move through the circuit top-down as in Figure 18 and combine each type of probe with those on the same level and below it.

A crossproduct of w_j^i's with randomness. Consider for example $w_0^0 w_1^1 + r_2 + r_8$. We do not need to combine this probe with any intermediate that is independent of r_8 since r_8 is then again a trivial one-time pad. We note that all w_i^0 are independent of r_5 and r_6 and all w_i^1 are independent of r_{11} and r_{12}.

$$\begin{aligned}
(w_0^0 w_1^1 + r_2 + r_8, w_1^0 w_0^1 + r_2 + r_8) &\sim (r_{11} r_6 + r_2 + r_8, r_{12} r_5 + r_2 + r_8) \\
(w_0^0 w_1^1 + r_2 + r_8, w_2^0 w_2^1) &\sim (w_0^0 w_1^1 + r_2 + r_8, r_{12} r_5) \sim (r_8, r_{12} r_5) \\
(w_0^0 w_1^1 + r_2 + r_8, w_0^1) &\sim (w_0^0 w_1^1 + r_2 + r_8, r_5) \sim (r_8, r_5) \\
(w_0^0 w_1^1 + r_2 + r_8, y_2^2 y_2^3) &\sim (w_0^0 r_6 + r_2 + r_8, y_2^2 r_{12}) \sim (w_0^0 r_6 + r_2 + r_8, r_9 r_{12}) \sim (r_8, r_9 r_{12}) \\
(w_0^0 w_1^1 + r_2 + r_8, y_0^2) &\sim (r_{11} r_6 + r_2 + r_8, y_0^2) \sim (r_2, r_7) \\
(w_0^0 w_1^1 + r_2 + r_8, x_0^4 x_2^5 + r_8) &\sim (r_2, x_0^4 x_2^5 + r_8) \sim (r_2, r_8)
\end{aligned}$$

A crossproduct of w_j^i without randomness. We take for example $w_0^0 w_0^1$. This is a product of w_0^0 which contains pads r_7 and r_{11} on the one hand and w_0^1 which is padded by r_1 and r_5 on the other. We therefore ignore other probes if they are independent of either r_7 or r_{11} and independent of either r_1 or r_5.

$$\begin{aligned}
(w_0^0 w_0^1, w_2^0 w_0^1) &\sim (w_0^0 w_0^1, r_{12} w_0^1) \sim (r_{11} w_0^1, r_{12} w_0^1) \sim (r_{11} r_5, r_{12} r_5) \\
(w_0^0 w_0^1, w_2^0 w_1^1) &\sim (w_0^0 r_5, r_{12} r_6) \sim (r_{11} r_5, r_{12} r_6) \\
(w_0^0 w_0^1, w_0^1) &\sim (r_{11} w_0^1, w_0^1) \sim (r_{11} r_5, r_5) \\
(w_0^0 w_0^1, w_1^1) &\sim (r_{11} r_5, r_6) \\
(w_0^0 w_0^1, y_1^0 y_0^1 + r_7) &\sim (r_{11} w_0^1, r_3 r_4 + r_7) \sim (r_{11} r_5, r_7) \\
(w_0^0 w_0^1, y_0^2 y_2^3) &\sim (w_0^0 r_5, y_0^2 r_{12}) \sim (r_{11} r_5, r_8 r_{12})
\end{aligned}$$

There are no y_j^i which can depend on both r_7 and r_{11} or on r_1 and r_5.

Intermediate w_j^i. We consider w_0^1 as example. It has potential one-time pads r_1 and r_5 so we only consider other intermediates that depend on both.

$$(w_0^1, y_1^0 y_0^1 + r_7) \sim (w_0^1, r_3 r_4 + r_7) \sim (r_5, r_7)$$
$$(w_0^1, y_1^0 y_2^1) \sim (w_0^1, r_3 r_6) \sim (r_5, r_3 r_6)$$

There are no y_j^i depending on both r_1 and r_5.

A crossproduct of y_j^i with randomness. Take for example $y_2^0 y_0^1 + r_{11}$, which can be replaced by r_{11} if combined with another probe that is independent of r_{11}. We thus consider intermediates depending on r_{11} only.

$$(y_2^0 y_0^1 + r_{11}, y_2^2 y_0^3 + r_5) \sim (r_3 r_4 + r_{11}, r_9 r_{10} + r_5) \sim (r_{11}, r_5)$$
$$(y_2^0 y_0^1 + r_{11}, y_0^0 y_2^1 + r_{11}) \sim (r_3 r_4 + r_{11}, r_1 r_6 + r_{11})$$
$$(y_2^0 y_0^1 + r_{11}, y_0^2 y_0^3) \sim (r_3 r_4 + r_{11}, r_7 r_{10}) \sim (r_{11}, r_7 r_{10})$$
$$(y_2^0 y_0^1 + r_{11}, y_2^3) \sim (y_2^0 y_0^1 + r_{11}, r_{12}) \sim (r_{11}, r_{12})$$
$$(y_2^0 y_0^1 + r_{11}, x_0^6 x_2^7 + r_{11}) \sim (r_3 r_4 + r_{11}, x_0^6 x_2^7 + r_{11})$$

The last pair of probes can be simulated using only 1 input share (x_0^6 and x_2^7).

A crossproduct of y_j^i without randomness and below. As of the level of crossproducts of y_j^i downwards, there is no more randomness recycling and the circuit corresponds to one with fresh randomness for each gate. In this case, the shared multiplication gates are 2-SNI, which implies the ability to simulate the remaining pairs.

F Nonzero Randomness

Our first-order masked AES requires 19 bits of fresh randomness for each S-box calculation. For this purpose, we instantiate an implementation of the stream cipher Trivium [Can06], which provides 19 bits in parallel each clock cycle.[2] Of these 19 bits, one byte serves as a new multiplicative mask r_0 and must therefore be nonzero. The probability that we end up with an unusable mask is 2^{-8}. Since the S-box is used 200 times per encryption (10 rounds with each 16 state bytes and 4 key bytes), we (over)estimate this event happening roughly once per encryption. We do not want to stall the pipeline until the PRNG generates a nonzero byte. Recall from Table 2, that the S-box receives an input in only 20 out of 24 clock cycles. This means that there are four cycles in each encryption round during which we are generating but not using 19 bits of randomness. This is more than enough to create a set of backup nonzero bytes in for example a FIFO. The size of the FIFO should depend on how many zero bytes we expect to see in one encryption round. Naturally, bytes are verified to be nonzero before being put in the FIFO.

We can model the number of PRNG failures X (= # zero bytes) over $n = 20$ trials with a binomial distribution with probability $p = 2^{-8}$.

$$\Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$$

The expected number of failures is then simply $\mathbb{E}[X] = np = 0.078$. A FIFO depth of only two or three bytes should thus more than suffice.

A similar approach can be used for the second-order implementation, in which 53 bits of randomness are required each cycle, of which two bytes must be nonzero.

[2] The Trivium cipher can be implemented to generate up to 64 bits in parallel.