Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order

Hannes Gross, Stefan Mangard, Thomas Korak

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
{firstname.lastname}@iaik.tugraz.at

Abstract. Passive physical attacks, like power analysis, pose a serious threat to the security of embedded systems and corresponding countermeasures need to be implemented. In this work, we demonstrate how the costs for protecting digital circuits against passive physical attacks can be lowered significantly. We introduce a novel masking approach called *domain-oriented masking* (DOM). Our approach provides the same level of security as threshold implementations (TI), while it requires less chip area and less randomness. DOM can also be scaled easily to arbitrary protection orders for any circuit.
To demonstrate the flexibility of our scheme, we apply DOM to a hardware design of the Advanced Encryption Standard (AES). The presented AES implementation is built in a way that it can be synthesized for any protection order. Although the design is scalable, it leads to the smallest (~7.1 kGE), fastest, and least randomness demanding (18 bits) first-order secure AES implementation. The gap between DOM and TI increases with the protection order. Our second-order secure AES S-box implementation, for example, has a hardware footprint that is half the size of the smallest existing second-order TI of the S-box. This paper includes synthesis results of our AES implementation up to the 15^{th} protection order.

Keywords: masking, domain-oriented masking, threshold implementations, private circuits, side-channel analysis, DPA, hardware security, AES

1 Introduction

The increasing number of interconnected embedded devices demand security not only on a cryptographic level but also on a physical level. Without countermeasures against physical attacks embedded devices are defenseless against attackers which have physical access. An attacker can easily extract device internal secrets by measuring the power consumption [16] or the electromagnetic emanation [22] of the device during security critical operations.

The most promising approach to achieve resistance against passive physical attacks is to make sensitive computations independent from the processed data by using so-called masking schemes. The first masking scheme—hereafter referred to as classical boolean masking scheme—was introduced by Goubin et al. [13]. Besides the classical

boolean masking scheme, there exist many other masking schemes, like Ishai et al.'s private circuits [15] and the Trichina gate [26]. However, the aforementioned schemes have been shown to be vulnerable against glitches and thus rigorous care has to be taken during the implementation to avoid leakage caused by glitches.

A masking scheme that is inherently immune against glitches is the threshold implementation (TI) masking scheme introduced by Nikova et al. [21]. It has been extensively researched and extended by Bilgin et al. [2, 4, 6] during the last years. There exist many protected hardware implementations that are based on TI, like Moradi's and Poschmann's AES implementation [20], which was improved by Bilgin et al. [3, 5].

However, the first approach to extend TI to higher protection orders [4] has shown to be vulnerable against glitches by Reparaz [23] which lead to the Generalized Masking Scheme [24] (GMS) as the secure extension of TI to higher orders. The GMS is also the basis for the second-order TI of the AES S-box of De Cnudde et al. [10].

Implementing higher-order protection based on the TI scheme, has been shown to be very costly. In particular the costs resulting from a high number of shares, like the required chip area and the amount of fresh random bits are significant [10]. Also the design effort is high as there exists no generalization to efficiently transform, for example, a first-order TI into a higher-order TI. As a result, the non-linear parts of a circuit need to be redesigned to fulfill the requirements for higher-order TI.

The private circuits scheme of Ishai et al. [15] on the other hand, which has a generalization to arbitrary protection orders, lacks the ability to deal with glitches. To the best of our knowledge there exists no secure and efficient implementation of the private circuit masking scheme in hardware that considers glitches. Furthermore, there has no hardware implementation been published that can be synthesized for arbitrary protection orders.

Our contribution In this work we introduce domain-oriented masking (DOM), a generic masking scheme that leads to hardware designs which can be synthesized for arbitrary protection orders. DOM thereby realizes the same theoretical bounds for fresh randomness as private circuits [15] without being vulnerable to glitches. Therefore, we introduce the concept of share domains and apply the idea of keeping each domain independent from other share domains.

Despite its generic construction, DOM provides lower implementation costs and the same level of security and glitch resistance as TI. The main differences between DOM and TI are: (1) our approach is domain oriented rather than function oriented, (2) the minimum number of shares are always used which results in a reduced gate count and a lowered number of required fresh random bits, (3) DOM protected hardware designs can be synthesized for arbitrary protection orders.

On the basis of the DOM scheme we present an AES implementation with freely scalable protection order. Nevertheless, our first-order secure AES variant requires only 7.1 kGE of chip area, 18 random bits per S-box call, and 246 clock (200 cycles for the interleaved variant) cycles per encryption. It is thus the smallest, least randomness demanding, and fastest first-order secure AES implementation. Our second-order AES S-box is with 5.3 kGE of chip area around two times smaller than existing implementations. Furthermore, the DOM AES S-box also requires less than half the number of

random bits of the TI S-box, while the AES core still achieves the same speed as our first-order implementation.

This paper includes implementation results of our AES implementation up to the 15^{th} protection order. The VHDL source code of the generic AES design is published online [14], which we hope will help future research and make comparisons easier.

This paper is organized as follows. In Section 2, we start with a brief recap of the most important properties of classical boolean masking approaches and threshold implementations on the basis of a Galois field multiplier. In Section 3, we introduce domain-oriented masking and make first comparisons with TI. In particular, we introduce two DOM multiplier designs with different scopes of application for efficiently protecting hardware designs. We then introduce two DOM-protected implementations of the AES in Section 4. A comparison of the implementation results with existing AES implementations is done in Section 5, which are based on the DOM multiplier designs. Section 6 concludes our work.

2 Masking

The core idea of masking is to make computations independent from the data that is processed. For this purpose, the data is split into a number of shares, which when recombined through addition over $GF(2^n)$ result in the original value. The sharing is done based on uniformly distributed random numbers. A sharing of a variable x can be written as shown in Equation 1, where the shares are denoted by capital letters with the shared variable in the subscript index.

$$x = A_x + B_x + C_x + \dots \tag{1}$$

The sharing does not only affect the representation of the data but also of the functions that are applied to this data. An unshared function "F" is split up into a number of component functions denoted by the original functions name with a capital letter in the index. Again the sum over the component functions must give the same result as for the unshared variables (see Equation 2).

$$\mathrm{F}(x, y) = \mathrm{F_A} + \mathrm{F_B} + \mathrm{F_C} + \dots \tag{2}$$

A basic requirement of all masking schemes is that each intermediate signal needs to be statistically independent of all unshared inputs and outputs. Often maintaining this independence requires the addition of a fresh random share to intermediate results. In this paper we always use Z shares to refer to randomly picked shares with the intention to provide statistical independence. The independence requirement is strongly related to the so-called *probing model* introduced by Ishai et al. [15] where the security against a probing attacker limited to d (= protection order) probing needles is considered. It was demonstrated by Faust et al. [11] and Rivain et al. [25] that there indeed exists a relation between the number of probed wires in the d-probing model and the attack order for a differential power analysis (DPA) attack. As it was shown by Chari et al. [9], there exist

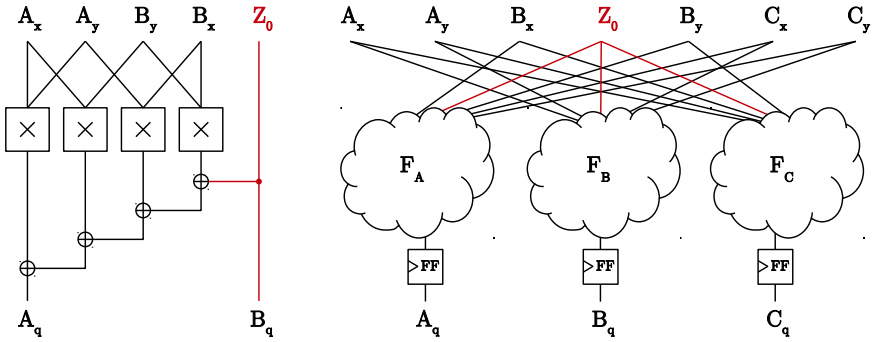

Fig. 1. Classical masked $GF(2^n)$ multiplier (left) and TI with component functions (right)

an exponential relation between the protection order and the number of leakage traces that are required for revealing the intermediate values.

While linear functions over $GF(2^n)$ can be implemented trivially, the implementation of nonlinear parts is quite challenging. In the past, Galois field (GF) multipliers have shown to be a good reference point to compare masking schemes. Multipliers are also of particular interest because on the basis of a simple one-bit multiplier, which corresponds to an AND gate, every boolean logic gate can be realized and in consequence every possible circuit. In the following, we explain the classical masking scheme and the threshold implementations scheme on the basis of a Galois field multiplier that is protected against first-order attacks.

2.1 Classical Boolean Masking

First-order masking requires only two shares. Accordingly, a shared multiplication of two inputs x and y over $GF(2^n)$ can be written as the multiplication of two shared finite field elements as demonstrated in Equation 3, where $x = A_x + B_x$ and $y = A_y + B_y$.

$$\begin{aligned} q = x \times y &= (A_x + B_x)(A_y + B_y) \\ &= A_x A_y + A_x B_y + B_x A_y + B_x B_y \end{aligned} \tag{3}$$

While all partial products are independent of x and y, the resulting sum is not. Therefore, a fresh random share Z_0 needs to be added to the first multiplication result of Equation 3 to break the dependency between the intermediates calculated by the multipliers. Figure 1 (left) shows a classical masked $GF(2^n)$ multiplier.

Even tough this multiplier seems to be secure in the d-probing model, this implementation is still not free from first-order leakages. Consider the results of the two the multipliers on the left, for example, that calculate $A_x A_y$ and $A_x B_y$. If this intermediate signals reach the exclusive-OR gate before Z_0 then the resulting signal is no longer independent to the value of y. As a result, the security of the masked multiplier depends on signal transition times caused by wire lengths, transistor speeds, et cetera, which

is hard to control for digital designers. For this reason, the classical boolean masking scheme is considered to be flawed [18]. As it was shown by Mangard et al. [19] the problems are not caused by the four multipliers itself or their additive leakage. The only reason this sharing produces first-order leakages is due to glitches caused by the addition of the multiplier results.

Researchers have tried to repair the masked multiplier in Figure 1 (left) [1, 12, 17]. These works mainly focused on balancing and reordering the signals and gates in such a way that no glitches can occur any longer. These approaches, however, require an enormous effort in the backend of the hardware design flow in order to guarantee the correctness of the signal timings.

2.2 Threshold Implementations

Threshold implementations focus on component functions and on the properties these component functions have to fulfill to guarantee data independence even in the presence of glitches. These properties are correctness, non-completeness, and uniformity.

The *correctness* property of TI simply requires that the sum over the component functions must give the same result as for the unshared variables. For first-order secure implementations, the *non-completeness* property requires for each component function to be independent of at least one share per variable. The non-completeness rule can also be generalized [4] to any order of security d requiring for up to d component functions to be independent of at least one input share.

As a consequence of the non-completeness property, each component function output needs to be fed into a register before it can be safely used in the next function. The property that is usually the hardest to achieve is the *uniformity* which demands all share inputs and outputs of component functions to be uniformly distributed regardless of which unshared values they represent. For first-order TI the uniformity of the component functions can be often achieved without performing a complete resharing of the outputs by using more shares, or correction terms, or using fresh random shares in more than one component function. However, the GMS of Reparaz et al. [24], as the secure extension of TI to higher orders, requires a resharing of all output shares with fresh random bits to avoid glitches.

While functions that are linear over $GF(2^n)$ can be implemented in a first-order secure manner with only two shares, Nikova et al. [21] states the lower bound for non-linear functions with two variables to be at least three shares. In general, the number of input shares required for higher-order security is given by $s_{in} \geq d \times t + 1$ where s is the number of shares, d is the protection order, and t the degree of the function. The number of output shares for TI is given with $s_{out} \geq \binom{s_{in}}{t}$. In [24], Reparaz et al. mentioned that given an independent input sharing and carefully designed component functions with more calculation steps and registers, the number of shares can be lowered to $d+1$. However, they state that they avoid giving a generic construction for the $d + 1$ case because extreme care has to be taken to not unmask any intermediate value.

As an example for a first-order secure TI, the $GF(2)$ multiplier in Figure 1 (right) uses three shares per variable and one fresh random share for achieving uniformity of all output shares as shown by Bilgin [7] (Equation 4).

$$
\begin{aligned}
\mathrm{F_A} &= B_x B_y + B_x C_y + C_x B_y + Z_0 \\
\mathrm{F_B} &= C_x C_y + A_x C_y + C_x A_y + A_x Z_0 + A_y Z_0 \\
\mathrm{F_C} &= A_x A_y + A_x B_y + B_x A_y + A_x Z_0 + A_y Z_0 + Z_0
\end{aligned} \tag{4}
$$

For the sake of completeness we note that the TI multiplier could also be implemented by using two fresh random shares and less logic gates, or with more shares and an increased gate count.

First-order leakage caused by glitches is avoided in any TI design through the non-completeness rule. The first component function $\mathrm{F_A}$, e.g., is independent of all A shares, the second component function $\mathrm{F_B}$ of all the B shares, and so forth. The glitch resistance however comes at high costs in terms of gate count. While the classical multiplier requires only four AND gates and four XOR gates, the TI variant in Equation 4 consumes 13 AND gates, 12 XOR gates, and three registers. In the next section we show that our DOM schemes achieves the same security as TI by requiring only four AND gates, four XOR gates, and two registers for the multiplier.

3 Domain-Oriented Masking (DOM)

In contrast to threshold implementations, which consider properties at function level, our approach is based on the concept of share domains. In DOM implementations, each share of a variable is associated with one share domain. This is also reflected in the notation that is used in this paper. The shares A_x and B_x of a variable x, for example, are associated with the domains A and B, respectively.

A DOM implementation uses $d+1$ shares per variable in order to achieve d^{th}-order security. There are $d+1$ domains in this case. The basic idea of the DOM approach is to keep the shares of all domains independent from shares of other domains. This independence ensures d^{th}-order security according to the d-probing model. If, for example, in a first-order security setting, a component function takes the inputs A_x and A_y from domain A, all intermediate values calculated by this function are independent of the corresponding unshared inputs $x = A_x + B_x$ and $y = A_y + B_y$. This is a consequence of the fact that B_x and B_y are not part of this function and are combined by another component function working on domain B.

In case of linear functions, the independence of the domains is trivial to achieve because linear functions only require to combine shares within one share domain. The critical part, like in all masking schemes, are the non-linear functions. In the case of non-linear functions, domain borders need to be crossed and dedicated measures need to be taken in order to maintain the independence of the shares in the different domains. The basic idea of DOM is to secure domain crossings by adding a fresh random share Z and by using a register in order to prevent that glitches propagate from one domain to the other.

In the following, we detail the concept of two-input GF DOM multipliers, which can serve as a basis to protect arbitrary circuits, and which are also the most critical part of masked AES implementations. In particular, we introduce two variants of DOM

multipliers named DOM-*indep* and DOM-*dep*. The advantage of the DOM-*indep* multiplier is its lower demand for fresh randomness and also the gate count is significantly smaller compared to the DOM-*dep* multiplier especially for low protection orders. The difference between these multipliers, however, is that the DOM-*indep* variant requires that the inputs are shared independently.

As an example for a violation of share independence consider the classical masked GF multiplier in Figure 1 (left). If this multiplier calculates $x \times x$ for the same sharing of both inputs x, then this would result in the multiplication terms $A_x A_x$, $A_x B_x$, $B_x A_x$, and $B_x B_x$. The terms $A_x A_x$ and $B_x B_x$ use only one share of x and are thus uncritical. The terms $A_x B_x$ and $B_x A_x$, on the other hand, violate the share independence by bringing shares from different domains of one sharing together. Even without the XOR gates at the outputs of the classical masked GF multiplier, the intermediate results depend on x. The share independence of course only requires the shares of the inputs to be pairwise independent not the variables themselves. It is therefore possible to calculate, for example, $x \times x$ with the DOM-*indep* multiplier, as long as both inputs are shared independently.

In practice, dependencies between shares can be less obvious and can even be just temporary. As an example consider a 2-bit transformation that is defined as follows: The first output bit q_0 of this transformation is the linear combination of two of the input bits $x_0 + x_1$ and the second output bit is just the first input bit $q_1 = x_0$. Due to signal delays it is possible that both output bits are temporarily formed by the same input bit x_0 only. If those bits are the shared inputs of a non-linear function, then this again results in a temporary violation of the share independence in form of glitches.

We start to introduce DOM by means of the first-order secure DOM-*indep* multiplier in Section 3.1 and extend it in Section 3.2 to arbitrary protection orders. The independence requirement for the input shares of the DOM-*indep* multiplier allows us to successively introduce the basic concept of DOM and to show its security in the d-probing model. This multiplier design is similar to the design of Ishai et al. [15] but without the vulnerability to glitches, and it has a balanced arrangement of the multiplication terms which shortens the signal delay paths.

On the basis of the DOM-*indep* multiplier, we then construct the DOM-*dep* multiplier for inputs with related sharing and discuss its security. In Section 3.4 a comparison of the DOM multiplier variants is given.

3.1 1st-Order Secure DOM-*indep* Multiplier

A first-order secure DOM-*indep* multiplier (see Figure 2) consists of two share domains. The inputs x and y are provided to the multiplier by the shares A_x and B_x, and A_y and B_y, respectively. The sharings for x and y are required to be uniformly random and independent from each other. The multiplier returns the shares A_q and B_q of the output q. A DOM multiplier performs three steps in order to map the input shares to the output shares. We refer to these steps as *calculation*, *resharing* and *integration*.

Calculation: In the calculation step, the actual multiplication is performed and the product terms $A_x A_y$, $A_x B_y$, $B_x A_y$ and $B_x B_y$ are calculated. In DOM, we distinguish between inner-domain terms ($A_x A_y$, $B_x B_y$) and cross-domain terms ($A_x B_y$, $B_x A_y$). The calculation of inner-domain terms only combines shares within one domain. These

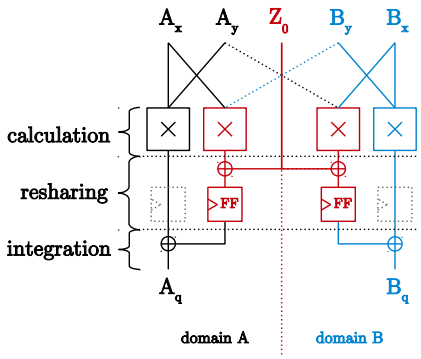

Fig. 2. First-order DOM-*indep* $GF(2^n)$ multiplier

terms are not critical from a security point of view. Any function that is computed based on shares that are independent from the shares in other domains only lead to outputs that are also independent of the shares of the other domains.

In case of cross-domain calculations, there is less freedom. In fact, in a DOM scheme, cross-domain calculations can only be done for independently shared variables. If shares of the same variable would be combined for example, the scheme would be trivially broken. For example, the product $A_x B_x$ would leak information about x. However, shares from different domains that correspond to different variables can be combined without violating the requirement for d^{th}-order security. In fact, there is no leakage about x or y when calculating any function of A_x and B_y. This results from the requirement that x and y are independently shared. There is also no leakage caused by any function of B_x and A_y for an independent sharing of x and y. Circuit parts that operate on inputs from multiple domains are plotted red in Figure 2. These parts are not assigned to a specific domain and contain the cross-domain terms.

Resharing: In DOM, the integration of cross-domain terms into a domain is prepared in the resharing step. By adding a fresh random Z share to a cross-domain term, it becomes statistically independent from all other values and can therefore be added to any arbitrary domain in a next step. However, using a new share for each cross-domain term would lead to a high overhead. In DOM, the goal is to minimize the number of fresh shares. In case of the 1^{st}-order secure multiplier, the same fresh share Z_0 is used for the resharing of the product terms $A_x B_y$ and $B_x A_y$. This does not lead to a first-order leakage and at the same time allows to build a very efficient design.

In order to prevent that any glitch propagates through the resharing step, in DOM always a register is included as last part of the resharing step. The two registers in grey dotted lines are optional registers for the inner-domain terms and are only required in the case pipelining is used. At first sight, the registers in Figure 2 seem to add an additional delay compared to the TI variant of the multiplier. However, the TI scheme also requires registers at the output of each component function. Otherwise no cascading of functions is possible. In case of a DOM multiplier, the output can be directly plugged into the next

non-linear function. The number of register stages is thus the same for both schemes (cf. Figure 1, right).

Integration: During the integration phase, the reshared cross-domain terms are added to the domains, which concludes the GF multiplication. This addition leads to glitches at the XOR gate at the output of the domain. However, as the resharing finishes with a register no glitches can occur that depend on x or y. In terms of correctness of the scheme, it is important to point out that the fresh share Z_0 becomes part of both domains of the multiplier. Hence, it holds that $q = A_q + B_q$ and there is no need for any additional share.

In summary, the security against a first-order probing attacker is given because each domain contains only inner-domain terms and cross-domain terms that are reshared with a fresh random share which is only used once in each domain. An attacker thus always needs to combine two or more intermediate signals to get one signal that depends on one of the independently shared inputs x or y. Problems caused by different signal propagation times as for the classical masked multiplier in Section 2.1 are prevented through registered outputs in the resharing phase.

3.2 Higher-Order Secure DOM-*indep* Multiplier

The first-order DOM multiplier can be extended to arbitrary protection orders. The generalization requires to first extend the *calculation* phase to produce a correct sharing with $d + 1$ shares for any given protection order d. In the *resharing* phase it needs to be ensured that the fresh random Z shares are distributed over the domains in a way that (1) each cross-domain term is reshared with a Z share that is unique inside the targeted domain, and (2) none of the signal combinations created in the *integration* phase reveals more than the inner-domain terms or shares.

Calculation: The same rules as for the first-order DOM apply for the higher-order generalization. Again, any combination of inner-domain shares can be used for the multiplication terms inside their associated domain without any restrictions. Cross-domain multiplication terms are restricted to be originated from independently shared variables to prevent the case that two shares of the same sharing are combined.

Given this restrictions, the $GF(2^n)$ multiplication formula in Equation 3 can be easily generalized for $d + 1$ input shares per variable as shown in Equation 5. Each row of this formula stands for one component function calculating one share of the output q. The multiplications in the diagonal (bold) are the inner-domain multiplication terms containing only shares from one specific domain and hence only leak about shares of this domain. The cross-domain products do not leak more information on the inputs x and y then when probing one share of x and one share of y directly. Hence, with this formula the sharing for the *calculation* step for the GF multiplier is secure in the d-probing model and can be realized for arbitrary numbers of shares. An example for a second-order DOM multiplier is given in Figure 3.

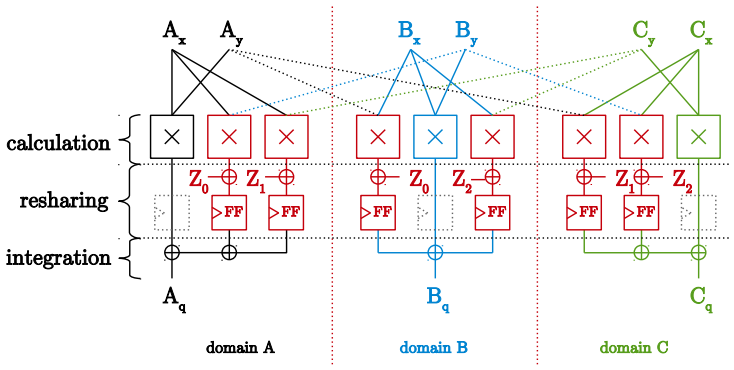

Fig. 3. Second-order secure DOM-*indep* $GF(2^n)$ multiplier

$$
\begin{aligned}
q = x \times y =& (A_x + B_x + C_x + D_x + E_x + \dots)(A_y + B_y + C_y + D_y + E_y + \dots) \\
=& \underbrace{\boldsymbol{A_x A_y} + A_x B_y + A_x C_y + A_x D_y + A_x E_y + \dots}_{A_q} \\
& \underbrace{B_x A_y + \boldsymbol{B_x B_y} + B_x C_y + B_x D_y + B_x E_y + \dots}_{B_q} \\
& \underbrace{C_x A_y + C_x B_y + \boldsymbol{C_x C_y} + C_x D_y + C_x E_y + \dots}_{C_q} \\
& \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots
\end{aligned}
\tag{5}
$$

Resharing: A core property for the generalization of the DOM scheme is how the required fresh random Z shares can be efficiently distributed among the cross-domain terms in a correct manner. From Equation 5 it can be seen that there are exactly $d(d+1)$ cross-domain terms which need to be reshared. It is also important to note that there are exactly two product terms that combine shares from two given domains. For example shares from domain A and B are only combined in the terms $A_x B_y$ and $B_x A_y$. In DOM, we use the same fresh share for product terms that combine shares from the same domain, see Equation 6. Hence, we use $d(d+1)/2$ fresh shares for a d$^{\text{th}}$-order DOM implementation of the multiplier (like Ishai et al. [15]).

Since no probing of any intermediate value created in the *calculation* phase contains more than one share of each input variable x or y, and in the *resharing* phase we only add fresh random shares to the cross-domain terms, no advantage to a d-probing attacker is given during these phases.

Integration: In the integration phase, the multiplication terms of each component function are added up at the output of the domain. Because a digital designer has no influ-

ence on the sequence in which these terms are added up, the higher-order DOM multiplier needs to provide security for each possible partial sum of these terms. In particular, it has to be taken care of that each of these possible partial sums an attacker could probe reveals only the shares of the domain she is probing in. This is ensured by the resharing shown in Equation 6, where each Z share is only reused for cross-domain multiplication terms with the same domain association. Please note that the matrix in Equation 6 appears similar to the one used for the ISW algorithm in private circuits [15]. The difference, however, is in the distribution of the multiplications terms which allows a more efficient implementation in hardware.

In order to exploit the reuse, it would be necessary to probe the two component functions that use the terms with the reused share. However, the two component functions that use the two terms are associated to the same domains as the terms in the cross-domain products. Hence, there is no advantage for the attacker due the reuse.

For example, the share Z_0 in Figure 3 is used on the terms A_xB_y and B_xA_y and these two terms only occur in the component functions for A_q and B_q. An attacker that probes any partial sum of the terms of A_q learns only about shares in domain A. When probing any partial sum of the terms B_q, there is only information about shares of the domain B. A second-order attacker that learns about partial sums of A_q and B_q learns about shares from the domains A and B in any case. The fact that partial products A_xB_y and B_xA_y reuse Z_0 does not provide any advantage to an attacker.

Based on Equation 5, the fact that a DOM multiplier fulfills d^{th}-order security can be verified visually. In this multiplication matrix the diagonal terms are formed by the inner-domain product terms. These inner-domain terms also divide the multiplication matrix into an upper and lower triangular matrix in which each of the fresh random Z shares is used exactly once. The triangle formed by the Z shares is mirrored along the diagonal. The mirroring of the Z shares ensures that each possible combination of partial sums from any two component function removes at most one fresh random share, and reveals only the inner-domain shares of both domains. Because this applies for all combinations of partial sums of all different domains, an attacker restricted to d probes obtains at most d shares per variable. The higher order multiplier is thus secure in the d-probing model.

$$
\begin{aligned}
\mathrm{A_q} &= \boldsymbol{A_xA_y} + (A_xB_y + Z_0) + (A_xC_y + Z_1) + (A_xD_y + Z_3) + (A_xE_y + Z_6) + \dots \\
\mathrm{B_q} &= (B_xA_y + Z_0) + \boldsymbol{B_xB_y} + (B_xC_y + Z_2) + (B_xD_y + Z_4) + (B_xE_y + Z_7) + \dots \\
\mathrm{C_q} &= (C_xA_y + Z_1) + (C_xB_y + Z_2) + \boldsymbol{C_xC_y} + (C_xD_y + Z_5) + (C_xE_y + Z_8) + \dots \\
\mathrm{D_q} &= (D_xA_y + Z_3) + (D_xB_y + Z_4) + (D_xC_y + Z_5) + \boldsymbol{D_xD_y} + (D_xE_y + Z_9) + \dots \\
\mathrm{E_q} &= (E_xA_y + Z_6) + (E_xB_y + Z_7) + (E_xC_y + Z_8) + (E_xD_y + Z_9) + \boldsymbol{E_xE_y} + \dots \\
&\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots
\end{aligned}
\tag{6}
$$

The component functions of the multiplication matrix can also be written in closed form as shown in Equation 7.

$$
\mathrm{F_i} = t_{i,i} + \sum_{j>i}^{d} \left(t_{i,j} + Z_{(i+j*(j-1)/2)}\right) + \sum_{j<i}^{d} \left(t_{i,j} + Z_{(j+i*(i-1)/2)}\right) \tag{7}
$$

This equation is the basis for the scalable AES designs in Section 4.

3.3 Higher-Order Secure DOM-*dep* Multiplier

For the DOM-*dep* multiplier we do not require that its inputs are shared independently. The DOM-*dep* multiplier construction is secure, even if the shares of the inputs x and y are identical ($A_x = A_y$, $B_x = B_y$, ...). A straightforward way to implement the DOM-*dep* multiplier would be to use the DOM-*indep* multiplier and to reshare one of its inputs. The additional cost in this case are d fresh random shares and $d + 1$ registers to ensure the resharing is done before the multiplication is performed. This construction has a high negative impact on the performance of the multiplier because the multiplication requires two cycles instead of one. To overcome the need for a two register stages, we can trade one register stage against more computational overhead as follows.

DOM-dep Multipliers with One Register Stage Instead of calculating $x \times y$ directly, a random blinding value z is used to calculate it as shown in Equation 8.

$$x \times y = x \times \underbrace{(y + z)}_{\text{blinding}} + \underbrace{(x \times z)}_{\text{correction}} \tag{8}$$

On first sight this appears quite similar to the resharing approach with just more computational overhead. However, z does not represent a sharing of the zero bit vector as it is used for resharing, but a random blinding value. The blinding of y allows for calculation of the multiplication $x \times (y+z)$ in an efficient manner. At first, one blinding share is added to each share of y resulting in Equation 9.

$$y + z = (A_y + A_z) + (B_y + B_z) + (C_y + C_z) \ldots \tag{9}$$

Again, registers are needed to ensure that the blinding of each share is performed before the next calculation step is taken. Because the dependency between the blinded shares and y is no longer given, the blinded shares can now be summed up without demasking y but only $y + z$ resulting in a single value b. The first multiplication in Equation 8 can now be performed by multiplying each share of x with b by using normal GF multipliers instead of DOM multipliers (Equation 10).

$$x \times (y + z) = x \times b = bA_x + bB_x + bC_x \ldots \tag{10}$$

The d-probing security is given because each possible linear combination of blinded y shares ($(A_y + A_z)$, $(B_y + B_z) \ldots$), that could be formed during the calculation of b, result again in a uniformly random value unrelated to the shares of y. Multiplying each share of x with a uniformly random value also does not compromise the security of x. Please note that we strictly keep all x shares in their specific domains so that d probing security for x is straightforwardly given at all times.

Furthermore, for each share of y that is used for the calculation of b one fresh random mask is added. As a result, at each time y is protected by $d+1$ shares, either by the y shares itself or by the z shares. Vice versa, z is protected during the calculation of b

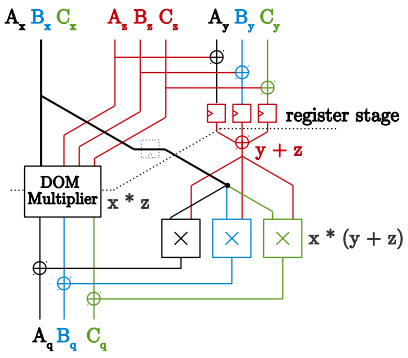

Fig. 4. Second-order DOM-*dep* multiplier

because each shares of z is masked by a share of y. Consequently, in order to protect y, the unshared value of z needs to be protected against probing otherwise y can be easily recovered from b.

For the second multiplier of Equation 8 calculating the *correction* $x \times z$, we thus need to ensure that at no time two shares of z come together. This can be easily achieved by using the DOM multipliers from Figure 2 or Figure 3, respectively. Since, x and z are independent because z is always randomly picked, calculating $x \times z$ is safe regarding the independence requirement.

Adding the result of both multiplications together with respect to the domains is then straightforward. Equation 11 shows the full mathematical description of the multiplier implementation.

$$
\begin{aligned}
\mathrm{A_q} &= A_x(\underbrace{(A_y + A_z) + (B_y + B_z) + \ldots}_{b}) + A_x A_z + (A_x B_z + Z_0) + (A_x C_z + Z_1) + \ldots \\
\mathrm{B_q} &= B_x((A_y + A_z) + (B_y + B_z) + \ldots) + (B_x A_z + Z_0) + B_x B_z + (B_x C_z + Z_2) + \ldots \\
\mathrm{C_q} &= \underbrace{C_x((A_y + A_z) + (B_y + B_z) + \ldots)}_{x \times b} + \underbrace{(C_x A_z + Z_1) + (C_x B_z + Z_2) + C_x C_z + \ldots}_{\text{DOM-}indep\text{ multiplier: } x \times z} \\
&\vdots
\end{aligned}
\tag{11}
$$

Figure 4 shows the resulting design of a second-order DOM-*dep* multiplier. The shares of x are illustrated in a bundled manner for clarity reasons. As one can see, the design contains only one register stage going through the DOM multiplier and the calculation of b. In addition to the requirements of the DOM-*indep* multiplier, a linear number of fresh random shares for z are required, as well as a linear amount of registers and GF multipliers. The overhead for transforming a DOM-*indep* multiplier into a DOM-*dep* multiplier thus grows linearly in terms of randomness as well as chip area.

Increasing the efficiency for first-order multipliers Even though the overhead of a DOM-*dep* multiplier compared to a DOM-*indep* multiplier grows just linearly, the insertion of the blinding value z triples the number of required random shares for the first-order multiplier. However, a closer investigation of Equation 11 reveals that the calculation of b actually uses more randomness than required. For example for the $x \times (y + z)$ part, $A_y + A_z$ is calculated before it is multiplied with A_x just to remove $A_x A_z$ again in the $x \times z$ part. This simply results from the simplification performed for higher-order DOM multipliers to compute b only once for all domains. If instead b is calculated for each domain individually, one z share can be saved. Equation 12 shows the updated multiplication formulas for a first-order multiplier with two fresh random shares only. Z_0 is used instead the shares of z for blinding y, and Z_1 is used for the secure integration of $A_x Z_0$ and $B_x Z_0$.

$$\begin{aligned} \mathrm{A_q} &= A_x(A_y + (B_y + Z_0)) + (A_x Z_0 + Z_1) \\ \mathrm{B_q} &= \underbrace{B_x(B_y + (A_y + Z_0))}_{x\times(y+z)} + \underbrace{(B_x Z_0 + Z_1)}_{x+z} \end{aligned} \tag{12}$$

We do not consider this approach to be suitable for orders above one because the area overhead introduced by calculating each b per domain increases quadratically with the protection order, and also the randomness overhead introduced by the blinding with one additional random share relativizes for higher orders.

3.4 Comparison of DOM Multiplier Variants

Table 1 summarizes the DOM multiplier constructions discussed in this section. The basis for the comparison is the DOM-*indep* which has the smallest requirement on fresh randomness, and chip area (number of GF multipliers, XOR gates and registers). Randomness and area numbers for the other variants are to be read as in addition to the results stated in the columns left.

Table 1. Summary of DOM multiplier variants

	DOM-*indep*	**DOM-*indep* + resharing**	**DOM-*dep***
Related input sharing	no	yes	yes
Register stages	1	2	1
Fresh random shares	$d(d+1)/2$	$\ldots + d$	$\ldots + 1$[a]
GF multipliers	$(d+1)^2$	$\ldots$	$\ldots + (d+1)$
XOR's	$d(d+1)$	$\ldots + (d+1)$	$\ldots + (d+1)$
Registers	$d(d+1)$	$\ldots + (d+1)$	

[a] Not required for first-order implementations.

Using the resharing approach to make the DOM-*indep* suitable for inputs with related sharing introduces a second register stage, and requires additionally d random

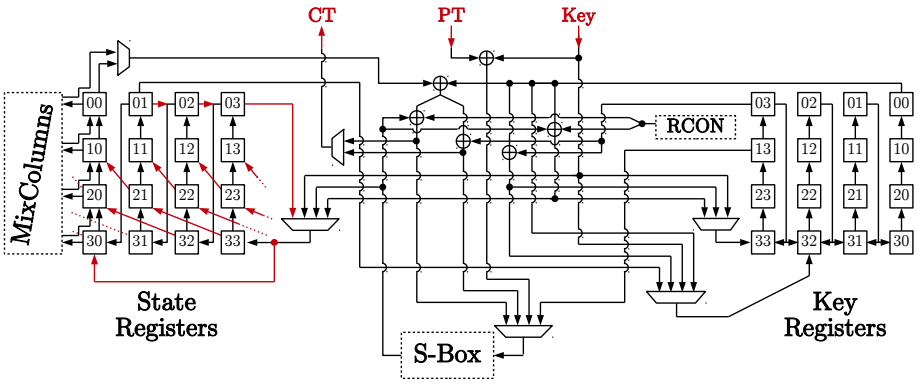

Fig. 5. AES-*interleaved* variant datapath (all data signals are 8 bits wide)

shares and $d + 1$ registers and XOR gates. The costs for avoiding the register stage for the DOM-*dep* multiplier is another random share (except for first-order implementations), $d + 1$ additional finite-field multipliers, and $(d + 1)$ XOR's for adding the result of $x \times z$ and $x \times b$ together. The total number of registers stays the same compared to the DOM-*indep* + resharing solution.
In the next section, the different variants of DOM multipliers are investigated in practice for a protected AES hardware implementation.

4 DOM Protected AES Implementations

To compare the efficiency the DOM scheme with existing threshold implementations, we implemented different variants of the AES-128 encryption-only design suggested by Moradi and Poschmann [20]. Moradi's design was also used and modified by Bilgin et al. [3, 5] resulting in a more efficient first-order TI, and recently by De Cnudde et al. [10] for a second-order TI of the AES S-box following the GMS scheme [23].

The control path of the design of Moradi et al. consists of a linear-feedback shift register (LFSR), the round constant generation module (RCON), and some additional logic gates to generate the control signals (see Figure 5 for an overview). The LFSR of Moradi's design has a cycle length of 21 for the unprotected implementation. In each round, the first 16 cycles are spent on *AddRoundKey* and *SubBytes*. Then there is one cycle spent on *ShiftRows*, and the remaining four cycles are used for *MixColumns* and to calculate the first four bytes of the next round key. For the TI variant another four cycles are added in each round due to the pipelining delay of the S-box. The datapath of Moradi's AES design mainly consists of the S-box, the key and state registers which are implemented as shift registers, the *MixColumns* module, and some multiplexers.

We implemented two variants of the AES S-box resulting in two AES designs with different design goals. The AES-*interleaved* design uses an S-box with only five pipelining stages and was designed for a minimum number of cycles per encryption. Because for some multipliers the sharing of the inputs are possibly related due to glitches,

the DOM-*dep* multipliers are used in this case. The AES-*simple* design uses only the DOM-*indep* multipliers but requires additionally three pipeline stages in the S-box to ensure independence. The AES-*simple* design has a lower demand for fresh randomness and a lower chip area requirement compared to the AES-*interleaved* but requires more cycles per encryption. Both variants are fully scalable in terms of protection order.

4.1 Different DOM Designs of the AES S-box

The most complex and most security critical part of the AES implementation is the S-box. Figure 6 shows our design of a 1$^{\text{st}}$-order secure DOM variant of Canright's [8] AES S-box. The S-box consists of many linear operations like the linear mappings at the input and the output, the square scalers, the sub-field inverters, and the adders. The Galois field multipliers with different field order form the non-linear parts of the S-box. Canright's S-box makes repeated use of a finite field isomorphism to express $GF(2^8)$ elements as multiple elements in lower subfields—down to eight elements in $GF(2)$.

The main difference between the first-order TI S-box designs and our DOM design, is that we use only two shares throughout the whole circuit. In order to secure all GF multipliers of the AES S-box depicted in Figure 6, we have implemented them based on the DOM concept introduced in Section 3. To maximize the efficiency of the implementation, additional pipelining registers are added to the S-box. The pipelining registers are marked with circles and appear along the red dotted lines in Figure 6. The grey marked registers and stages are only required for the eight-stage variant. As a result, each GF multiplier has an additional register in the path of the inner-domain multiplication before the last XOR gate at the domain output (cf. Figure 2).

To make the S-box secure and efficient at the same time, it is necessary to pinpoint all multipliers that have related input sharings. These multipliers need to be treated more carefully than the multipliers with independent inputs where the DOM-*indep* multipliers can be used. Therefore, the S-box is subdivided into five or eight pipeline stages, respectively. The following description refers to the five stage variant.

The $GF(2^4)$ multiplier in Stage 1 receives its inputs from the linear mapping at the S-box input. The linear mapping takes the 8-bit input shares A_x and B_x and linearly combines these eight bits inside their share domain (see [8] for more details). Because of the different signal transition times and gate delays, it is therefore possible that the output of the linear mapping temporarily consists of bits with related sharing. Applying these bits directly to the DOM-*indep* multiplier design from Figure 2—while the linear mapping has not yet settled—would thus violated the independence in the cross-domain GF multipliers. To avoid these glitches, the DOM-*dep* is used (Figure 4) for the five-stage S-box variant. For the eight-stage S-box the normal DOM multipliers are used but registers are inserted after the linear maps to ensure the signals are settled before the bits are applied to the multiplier.

The situation is similar at Stage 2 and Stage 3. At these stages glitches can occur from the combination of the square scaler outputs with the outputs of the DOM multipliers. Again these glitches can be avoided either by using DOM-*dep* multipliers or by inserting pipelining stages at the marked positions in Figure 4.

For the multipliers in Stage 4, the inputs are the pipelined S-box inputs and the output of the DOM multipliers of the previous stage. The output of the DOM multipli-

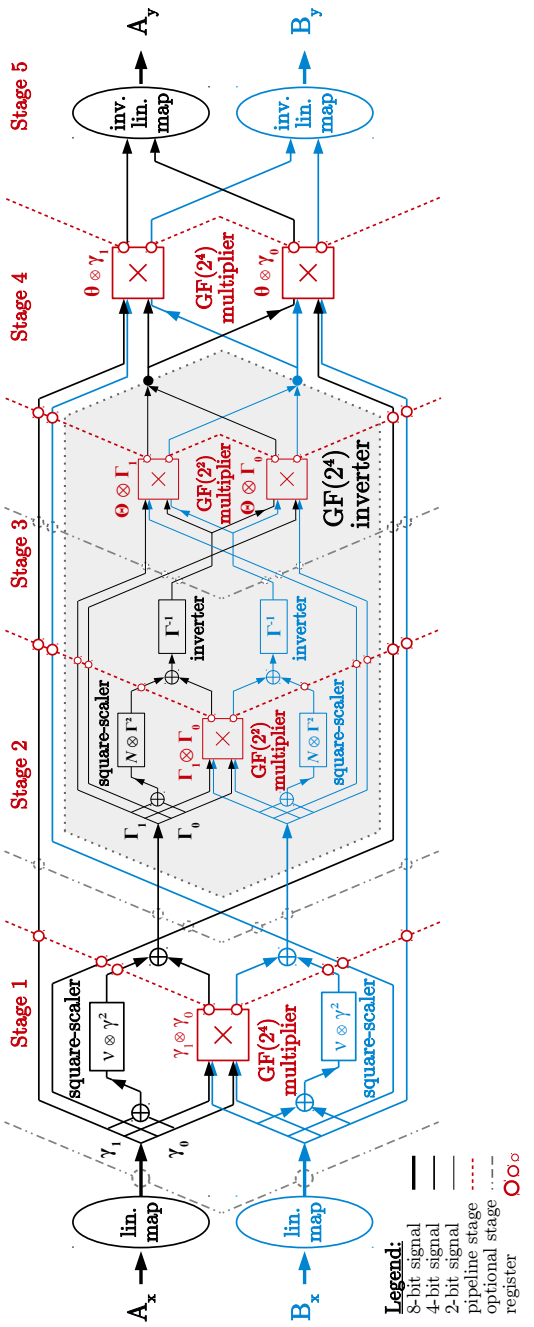

Fig. 6. First-order DOM design of the AES S-box, with five or eight register stages

ers of Stage 3 originate from the inputs of the $GF(2^4)$ inverter which is remasked in Stage 1 (the masking is effective at latest at Stage 2). Therefore, the inputs of the Stage 4 multipliers are clearly independent and so DOM-*indep* multipliers can be used here.

Stage 5 only performs the output mapping which is again a linear transformation and uncritical as long as it is not followed by a nonlinear transformation that is unprepared for related sharing of its inputs. As shown in Figure 5, the output of the S-box is either stored in the key or state registers before it is used again, or fed into the S-box which is also uncritical because the input multiplier of either S-box variant is already prepared to process related input sharings.

The rest of the S-box is implemented according to the original Canright design but without some of its optimizations which would not be beneficial for our implementation. Canright's design, for example, reuses some temporary results in other parts of the S-box. Storing temporary results would lead to many additional pipelining registers for the DOM design of the S-box and is therefore not suitable. For the generalization of the S-box to higher protection order, the black (or blue) parts in Figure 6 are duplicated and the GF multipliers are generated as described in Section 3.

4.2 AES-*interleaved*

The AES-*interleaved* variant has an asymptotic runtime of 200 cycles which is the minimum for a single S-box AES-128 implementation. This low cycle count is achieved due to a reduction of one encryption round to 20 cycles compared to Moradi's implementation. Therefore, the LFSR of the controlpath is modified to shorten the cycle length from 21 to 20. Furthermore, the first round and the last round of two successive encryptions are interleaved.

The datapath is illustrated in Figure 5. The AES-*interleaved* design uses the five-stage S-box variant. During the first round the key and plaintext are loaded byte-wise from outside into the AES core. Similar to Moradi's implementation, the first 16 cycles are spent on *AddRoundKey* and *SubBytes*. In the 16^{th} cycle the *ShiftRows* transformation is performed inside the state registers. The last four cycles are used for the key schedule. In the 10 AES rounds *MixColumns* is performed in parallel to *AddRoundKey* and *SubBytes*. After the last round, the final *AddRoundKey* produces the ciphertext which appears byte-wise at the output of the AES core. Simultaneously the next plaintext and key inputs can be loaded.

4.3 AES-*simple*

For the AES-*simple* design, the eight-stage S-box is used. The increased amount of delay cycles mainly affects the controller of the AES core, which now uses 23 cycles instead of 20 for one encryption round. The three additional cycles are inserted to finish the S-box calculations inside one round. During this time the clock is separated from the key registers by using a clock gating cell. Compared to the AES-*interleaved* design, the additional logic required for loading the next plaintext and calculating the final ciphertext for the is removed to make the AES-*simple* design more compact. As a result, one encryption takes 246 clock cycles.

Table 2. First-order secure AES-128 implementation results

Variant/Module	Chip Area [%]	Chip Area [kGE]	Randomness [Bits/S-box]	Cycles	Throughput @0.1 MHz [Kbps.]
Our Implementations					
AES-*simple*	100.0	7.1	18	246	52
S-box	37.0	2.6			
State registers	34.2	2.4			
Key registers	21.5	1.5			
Control, et cetera	7.3	0.6			
AES-*interleaved*	100.0	7.6	28	$\simeq 200$	64
S-box	37.3	2.8			
State registers	32.1	2.4			
Key registers	19.9	1.5			
Control, et cetera	10.7	0.9			
Related Implementations					
Moradi et al. [20]		11.0 / 10.8[a]	48	266	48
Bilgin et al. [3]		9.1 / 8.2[a]	44	246	52
Bilgin et al. [5]		8.1 / 7.3[a]	32	246	52

[a] This variant uses the *compile_ultra* option.

5 Implementation Results

In order to make the comparison with existing implementations as fair as possible we tried to use the same synthesis parameters as the related work if available. All results are thus collected for a UMC 180 nm generic II logic process with 1.8 V power supply and 0.1 MHz and are thus consistent with Moradi et al. [20] and Bilgin et al. [3, 5]. Even though the same standard cell library is used, the synthesis results may differ for the same RTL input because we have no access to the Synopsis tool chain—and especially to the *compile_ultra* synthesis option, which allows to highly optimize each part of the circuit individually—as used by related work. Our designs are compiled with the Cadence Encounter RTL compiler version v08.10-s28_1 and routed with Cadence NanoRoute v08.10-s155.

5.1 First-Order AES Implementations

Table 2 compares our first-order secure AES hardware implementations with existing related work. In terms of chip area requirements the AES-*simple* implementation uses only 7.1 kGE which is more than 1 kGE smaller than other designs. Comparison against the highly optimized results for the *compile_ultra* synthesis option of related work still shows a difference of 200 GE. The largest components in the design are the S-box with 37.0%, the state registers with 34.2%, and the key registers with 21.5% of the overall chip area. The remaining area is consumed by the control logic and other components (multiplexers, round constant calculation, et cetera).

The AES-*simple* variant requires only 18 bits of fresh randomness per S-box calculation while related implementations require between 32 bits and 48 bits. We assume

Table 3. Second-order secure AES-128 implementation results

Variant/Module	Chip Area *[%]*	Chip Area *[kGE]*	Randomness *[Bits/S-box]*	Cycles	Throughput @0.1 MHz *[Kbps.]*
Our Implementations					
AES-*simple*	100.0	11.9	54	246	52
S-box	44.5	5.3			
State registers	30.7	3.7			
Key registers	19.3	2.3			
Control, et cetera	5.5	0.6			
AES-*interleaved*	100.0	12.8	84	$\simeq 200$	64
S-box	45.0	5.7			
State registers	28.6	3.7			
Key registers	17.7	2.3			
Control, et cetera	8.6	1.1			
Related Implementations					
De Cnudde[a] et al. [10]		*11.2 / 7.9*[b]	126	-	-

[a] S-box implementation only

[b] This variant uses the *compile_ultra* flag which is not available in our tool chain.

that the number of required fresh random bits is even more crucial for the efficiency of an implementation because the generation of fresh random bits with high entropy requires additional hardware and involves, e.g., complex analog circuitry or pseudo random number generators based on symmetric primitives. Both options have a crucial influence on the chip area requirements, the energy budget, and on the delay or throughput.

The chip area requirements for the *interleaved* variant is about 0.5 kGE higher than for the *simple* AES variant. Due to the usage of blinding shares in the *interleaved* design, 28 random bits are required instead of 18 for this implementation. The higher amount of random bits allows to encrypt plaintext blocks with asymptotically 200 cycles instead of 246 cycles.

5.2 Second-Order AES Implementations

In the work of De Cnudde et al. [10] in 2015, the first and—to the best of our knowledge—only second-order TI of the AES S-box was introduced. In order to implement the GF multiplications, at least five shares were used to fulfill the non-completeness requirements for second-order TI. Our scheme, on the other hand, requires only three shares for a second-order secure implementation which saves two shares at minimum.

Table 3 shows that De Cnudde's S-box implementation is with 11.2 kGE almost as big as our overall chip area of the AES cores with 11.9 kGE. Furthermore, it requires 126 fresh random bits per S-box operation—not including the random bits required for share expansion at the input of the S-box—which is more than two times the number of random bits our *simple* implementation requires (54 bits).

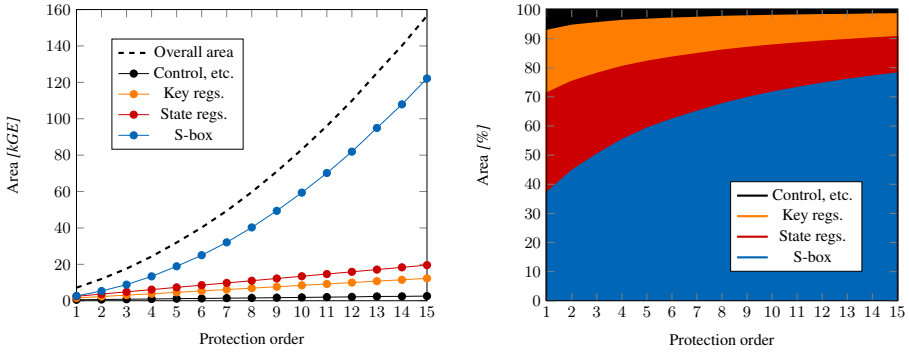

Fig. 7. AES-*simple* area requirements absolute (left) and in percent (right) per protection order

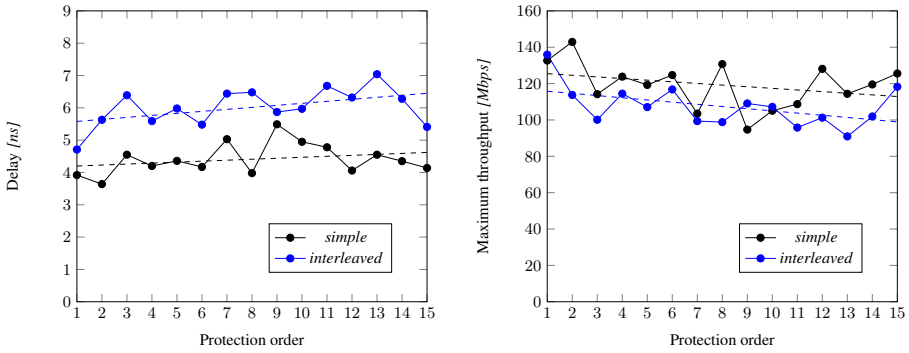

Fig. 8. Delay (left) and maximum throughput (right) versus protection order with trend lines

For our designs, the number of required cycles and the throughput is not influenced by the higher protection degree. The size of the S-box, the state and key registers, on the other hand, are increased because of the higher share count. A comparison between Table 2 and Table 3 shows that the S-box influences the area growth to a higher degree than all the other components with 44.5% of the overall area. The comparison also shows that the absolute overhead between AES-*simple* implementation and the AES-*interleaved* variant stays about the same which highly relativizes the overhead for higher order implementations.

5.3 d^{th}-Order AES Implementations

The generic construction of the AES implementation not only allows to calculate the number of required random bits of $9d(d+1)$ for the AES-*simple* variant, and $9d(d+1)+10d$ (first-order) or $9d(d+1)+10(d+1)$ for the AES-*interleaved* design, respectively. It is furthermore possible to synthesize the AES implementation for arbitrary protection orders.

Figure 7 shows the post-synthesis area requirements for the different components in relation to the protection order for the AES-*simple* variant. It can be observed that

the state key and control logic requirements grow linearly. The S-box and the contained multipliers grow quadratically with the order. For the S-box, the size increases from 37% for the first-order implementation to about 78% for the 15^{th}-order. The relative size of the state and key register decrease from 34.2% and 21.5% to around 12.5% and 8%, respectively. The smallest amount of chip area is again used for the control logic implementation which is almost constant.

Comparing Figure 7 to De Cnudde's implementation shows that our third-order DOM S-box implementation with 8.8 kGE has less area requirements than the second-order TI with 11.2 kGE without the *compile_ultra* synthesis option. Furthermore, our fourth-order DOM S-box implementation is with 13.4 kGE only about 2.2 kGE bigger than the second-order TI S-box. In terms of random bits, our third-order secure *simple* implementation requires 108 random bits and De Cnudde's second-order secure implementation 126 bits.

The influence of the increased number of shares to the longest combinatorial path is illustrated in Figure 8 (left). The maximum delay increases from 3.9 ns to about 4.1 ns for the *simple* variant which roughly corresponds to 28 MHz. The delay not only influences the maximum possible clock frequency but also the maximum throughput. As shown in Figure 8 (right), the *simple* variant achieves about 132 Mbps and the *interleaved* variant 136 Mbps for first-order protection assuming the maximum possible clock frequency. Due to the higher maximum delay of the *interleaved* variant, the maximum throughput of both implementations is quite comparable for different the protection orders. For a fixed clock frequency the throughput of the AES-*interleaved* is of course always higher.

6 Conclusions

In this work we introduced a novel masking scheme called domain-oriented masking (DOM). We have shown that our scheme reaches the same security level as TI. Nevertheless, the DOM scheme has some clear benefits over TI. At first, the number of required shares is reduced compared to existing TI implementations, especially for higher-order protection. As a consequence, the chip area requirements and the amount of fresh random bits also decrease. Furthermore, the DOM's generic structure leads to hardware implementations that can be synthesized for any given protection order without redesigning any part of the circuit.

We demonstrated the flexibility of DOM for a hardware design of the AES which can be synthesized for an arbitrary protection order. The first-order and second-order DOM AES designs are significantly smaller and less randomness demanding than the TI protected counterparts of related work. Besides the lowered hardware requirements, our AES-*interleaved* variant also requires significantly less cycles for computing one encryption. In addition we stated hardware result for our AES implementation up to the 15^{th} protection order.

References

1. M. Alam, S. Ghosh, M. Mohan, D. Mukhopadhyay, D. Chowdhury, and I. Gupta. Effect of Glitches Against Masked AES S-box Implementation and Countermeasure. *Information*

Security, IET, 3(1):34–44, March 2009.

2. B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In A. Francillon and P. Rohatgi, editors, *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 187–199. Springer International Publishing, 2014.
3. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A More Efficient AES Threshold Implementation. In D. Pointcheval and D. Vergnaud, editors, *Progress in Cryptology AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 267–284. Springer International Publishing, 2014.
4. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin Heidelberg, 2014.
5. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Trade-Offs for Threshold Implementations Illustrated on AES. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 34(7):1188–1200, July 2015.
6. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold Implementations of All 3x3 and 4x4 S-Boxes. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg, 2012.
7. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, N. Tokareva, and V. Vitkup. Threshold Implementations of Small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015.
8. D. Canright. *Cryptographic Hardware and Embedded Systems – CHES 2005: 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005. Proceedings*, chapter A Very Compact S-Box for AES, pages 441–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
9. S. Chari, C. Jutla, J. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. Wiener, editor, *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer Berlin Heidelberg, 1999.
10. T. D. Cnudde, B. Bilgin, O. Reparaz, V. Nikov, and S. Nikova. Higher-Order Threshold Implementation of the AES S-Box. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, pages 259–272, 2015.
11. S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan. Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In H. Gilbert, editor, *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 135–156. Springer Berlin Heidelberg, 2010.
12. S. Ghosh, M. Alam, K. Kumar, D. Mukhopadhyay, and D. Chowdhury. Preventing the Side-Channel Leakage of Masked AES S-Box. In *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*, pages 15–20, Dec 2007.
13. L. Goubin and J. Patarin. DES and Differential Power Analysis The Duplication Method. In *Cryptographic Hardware and Embedded Systems*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin Heidelberg, 1999.
14. H. Gross. DOM Protected Hardware Implementation of AES. `https://github.com/hgrosz/aes-dom`, 2016.
15. Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer Berlin Heidelberg, 2003.
16. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, 1999. Springer-Verlag.

17. K. Kumar, D. Mukhopadhyay, and D. RoyChowdhury. Design of a Differential Power Analysis Resistant Masked AES S-Box. In K. Srinathan, C. Rangan, and M. Yung, editors, *Progress in Cryptology INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 373–383. Springer Berlin Heidelberg, 2007.
18. S. Mangard, T. Popp, and B. Gammel. Side-Channel Leakage of Masked CMOS Gates. In A. Menezes, editor, *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer Berlin Heidelberg, 2005.
19. S. Mangard and K. Schramm. Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2006.
20. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT'11, pages 69–88, Berlin, Heidelberg, 2011. Springer-Verlag.
21. S. Nikova, C. Rechberger, and V. Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer Berlin Heidelberg, 2006.
22. J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer Berlin Heidelberg, 2001.
23. O. Reparaz. A note on the security of Higher-Order Threshold Implementations. *IACR Cryptology ePrint Archive*, 2015:1, 2015.
24. O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating Masking Schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 764–783, 2015.
25. M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer Berlin Heidelberg, 2010.
26. E. Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003:236, 2003.