Affine Equivalence and its Application to Tightening Threshold Implementations

Pascal Sasdrich, Amir Moradi, Tim Güneysu

Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
{pascal.sasdrich, amir.moradi, tim.gueneysu}@rub.de

Abstract. Motivated by the development of Side-Channel Analysis (SCA) countermeasures which can provide security up to a certain order, defeating higher-order attacks has become amongst the most challenging issues. For instance, Threshold Implementation (TI) which nicely solves the problem of glitches in masked hardware designs is able to avoid first-order leakages. Hence, its extension to higher orders aims at counteracting SCA attacks at higher orders, that might be limited to univariate scenarios. Although with respect to the number of traces as well as sensitivity to noise the higher the order, the harder it is to mount the attack, a d-order TI design is vulnerable to an attack at order $d + 1$.
In this work we look at the feasibility of higher-order attacks on first-order TI from another perspective. Instead of increasing the order of resistance by employing higher-order TIs, we go toward introducing structured randomness into the implementation. Our construction, which is a combination of masking and hiding, is dedicated to TI designs and deals with the concept of "affine equivalence" of Boolean functions. Such a combination hardens a design practically against higher-order attacks so that these attacks cannot be successfully mounted. We show that the area overhead of our construction is paid off by its ability to avoid higher-order leakages to be practically exploitable.

1 Introduction

Side-channel analysis (SCA) attacks exploit information leakage related to cryptographic device internals e.g., by analyzing the power consumption [11]. Hence, integration of dedicated countermeasures to SCA attacks into security-sensitive applications is essential particularly in case of pervasive applications (see [9, 17, 20]). Amongst the known countermeasures, *masking* as a form of secret sharing scheme has been extensively studied by the academic communities [8, 12]. Based on Boolean masking and multi-party computation concept, Threshold Implementation (TI) has been developed particularly for hardware platforms [15]. Since the TI concept is initially bases on counteracting only first-order attacks, trivially higher-order attacks, which make use of higher-order statistical moments to exploit the leakages, can still recover the secrets. Hence, the TI has been extended to higher orders [3] which might be limited to univariate settings [18]. In addition to its area and time overheads, which increase with the desired security order, the minimum number of shares also naturally increases, e.g., 3 shares for the first-order, 5 shares for the second-order, and at least 7 shares for the third-order security.

Contribution: In this work we look at the feasibility of higher-order attacks on first-order secure TI designs from another perspective. Instead of increasing the resistance against higher-order attacks by employing higher-order TIs, we intend to introduce structured randomness into a first-order secure TI. Our goal is to practically harden designs against higher-order attacks that are known to be sensitive to noise.

Concretely, we investigate the PRESENT [7] S-box under first-order secure TI settings that is decomposed into two quadratic functions thereby allowing the minimum number of three shares. By changing the decompositions during the operation of the device we can introduce (extra) randomness to the implementation. In particular we present different approaches to find and generate these decompositions on an FPGA platform and compare them in terms of area and time overheads. More importantly, we examine and compare the practical evaluation results of our constructions using a state-of-the-art leakage assessment methodology [10] at higher orders.

Our proposed approach which can be considered as a *hiding* technique is combined with first-order TI which provides provably secure first-order resistance. Therefore, although such a combination leads to higher area overhead, it brings its own advantage, i.e., practically avoiding the feasibility of higher-order attacks.

Outline: The remainder of this article is organized as follows: Section 2 recapitulates the concept of TI. We also briefly introduce the S-box decomposition for TI and affine equivalence in case of the PRESENT S-box. In Section 3 different approaches to find and exchange affine equivalent functions are presented and compared. Practical evaluation of our construction is given in Section 4. Finally, we conclude our research in Section 5.

2 Background

2.1 Threshold Implementation

We use lower-case letters for single-bit random variables, bold ones for vectors, raising indices for shares, and lowering indices for elements within a vector. We represent functions with sans serif fonts, and sets with calligraphic ones.

Let us denote an intermediate value of a cipher by $\boldsymbol{x}$ made of s single-bit signals $\langle x_1, \ldots, x_s \rangle$. The underlying concept of Threshold Implementation (TI) is to use Boolean masking to represent $\boldsymbol{x}$ in a shared form $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n)$, where $\boldsymbol{x} = \bigoplus_{i=1}^{n} \boldsymbol{x}^i$ and each $\boldsymbol{x}^i$ similarly denotes a vector of s single-bit signals $\langle x_1^i, \ldots, x_s^i \rangle$. A linear function $\mathsf{L}(.)$ can be trivially applied over the shares of $\boldsymbol{x}$ as $\mathsf{L}(\boldsymbol{x}) =$

$\bigoplus_{i=1}^{n} \mathsf{L}(\boldsymbol{x}^i)$. However, the realization of non-linear functions, e.g., an S-box, over Boolean masked data is challenging. Following the concept of TI, if the algebraic degree of the underlying S-box is denoted by t, the minimum number of shares to realize the S-box under the first-order TI settings is $n = t + 1$. Further, such a TI S-box provides the output $\boldsymbol{y} = \mathsf{S}(\boldsymbol{x})$ in a shared form $(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^m)$ with $m \geq n$ shares (usually $m = n$) in case of Bijective S-boxes. In case of a bijective S-box (e.g., of PRESENT) the bit length of $\boldsymbol{x}$ and $\boldsymbol{y}$ (respectively of their shared forms) are the same.

Each output share $\boldsymbol{y}^{j\in\{1,\ldots,m\}}$ is given by a component function $\mathsf{f}^j(.)$ over a subset of the input shares. To achieve the first-order security, each component functions $\mathsf{f}^{j\in\{1,\ldots,m\}}(.)$ must be independent of at least one input share.

Since the security of masking schemes is based on the uniform distribution of the masks, the output of a TI S-box must be also uniform as it is used as input in further parts of the implementation (e.g., the SLayer output of one PRESENT cipher round which is given to the next SLayer round after being processed by the linear PLayer and key addition). To express the *uniformity* under the TI concept suppose that for a certain input $\mathbf{x}$ all possible sharings $\mathcal{X} = \left\{(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n) | \mathbf{x} = \bigoplus_{i=1}^{n} \boldsymbol{x}^i\right\}$ are given to a TI S-box. The set made by the output shares, i.e., $\left\{\left(\mathsf{f}^1(.), \ldots, \mathsf{f}^m(.)\right) | (\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n) \in \mathcal{X}\right\}$, should be drawn uniformly from the set $\mathcal{Y} = \left\{(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^m) | \mathbf{y} = \bigoplus_{i=1}^{m} \boldsymbol{y}^i\right\}$ as all possible sharings of $\mathbf{y} = \mathsf{S}(\mathbf{x})$.

This process so-called uniformity check should be individually performed for $\forall\ \mathbf{x} \in \{0, 1\}^s$. We should note that if an S-box is a bijection and $m = n$, each $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n)$ should be mapped to a unique $(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^n)$. In other words, in this case it is enough to check whether the TI S-box forms also a bijection with $s \cdot n$ input (and output) bit length. For more detailed information we refer the interested reader to the original article [15].

2.2 S-Box Decomposition

Since the nonlinear part of most block ciphers, i.e., the S-box, has algebraic degree of $t > 2$, the number of input and output shares $n, m > 3$, which directly affects the circuit complexity and its area overhead. Therefore, it is preferable to decompose the S-box $\mathsf{S}(.)$ into smaller functions, e.g., $\mathsf{g} \circ \mathsf{f}(.)$, each of them with maximum algebraic degree of 2. It is noteworthy that if $\mathsf{S}(.)$ is a bijection, each of the smaller functions (here in this case $\mathsf{g}(.)$ and $\mathsf{f}(.)$) must also be a bijection. Such a trick helps keeping the number of shares for input and output at minimum, i.e., $n = m = 3$. However, it comes with the disadvantage of the necessity to place a register between each two consecutive TI smaller functions to avoid the glitches being propagated. Although such a composition is feasible in case of small S-boxes (let say up to 6-bit permutations [5]), it is still challenging to find such decompositions for 8×8 S-boxes. As stated before, the target of this work is an implementation of PRESENT cipher, which involves a 4×4 invertible cubic S-box (i.e., with the algebraic degree of 3) with Truth Table `C56B90AD3EF84712`. Therefore, all the representations below are coordinated based on 4-bit bijections.

In [16], where the first TI of PRESENT is presented, the authors gave a decomposition of the PRESENT S-box by two quadratic functions, i.e., each of which with the algebraic degree of 2. Later the authors of [4] and [5] presented a systematic approach which allows deriving the TI of all 4-bit bijections. In their seminal work they provided 302 classes of 4-bit bijections, with the application that every 4-bit bijection is affine equivalent to only one of such 302 classes. Based on their classification, the PRESENT S-box belongs to the cubic class $\mathcal{C}^4_{266}$ with Truth Table `0123468A5BCFED97`. It other words, it is possible to write the PRESENT S-box as $\mathsf{S} : \mathsf{A}' \circ \mathcal{C}^4_{266} \circ \mathsf{A}$, where $\mathsf{A}'(.)$ and $\mathsf{A}(.)$ are 4-bit bijective affine functions. Therefore, given the uniform TI representation of $\mathcal{C}^4_{266}$ one can easily apply $\mathsf{A}(.)$ on all input shares and $\mathsf{A}'(.)$ on all output shares to obtain a uniform TI of the PRESENT S-box.

As stated in [5] $\mathcal{C}^4_{266}$ can be decomposed into two 4-bit quadratic bijections belonging to the following combinations of classes: $(\mathcal{Q}_{12} \circ \mathcal{Q}_{12})$, $(\mathcal{Q}_{293} \circ \mathcal{Q}_{300})$, $(\mathcal{Q}_{294} \circ \mathcal{Q}_{299})$, $(\mathcal{Q}_{299} \circ \mathcal{Q}_{294})$, $(\mathcal{Q}_{299} \circ \mathcal{Q}_{299})$, $(\mathcal{Q}_{300} \circ \mathcal{Q}_{293})$, and $(\mathcal{Q}_{300} \circ \mathcal{Q}_{300})$. However, the uniform TI of the quadratic class $\mathcal{Q}_{300}$ <u>with 3 shares</u> can only be achieved if it is again decomposed in two parts. Therefore, the above decompositions in which $\mathcal{Q}_{300}$ is involved need to be implemented in 3 stages if the minimum number of 3 shares is desired. Excluding such decompositions we have four options to decompose the PRESENT S-box in two stages with 3-share uniform TI since the PRESENT S-box is affine equivalent to $\mathcal{C}^4_{266}$.

For the sake of simplicity – as an example – we consider the first decomposition, i.e., $\mathcal{Q}_{12} \circ \mathcal{Q}_{12}$, which indicates that it is possible to write the PRESENT S-box as $\mathsf{S} : \mathsf{A}'' \circ \mathcal{Q}_{12} \circ \mathsf{A}' \circ \mathcal{Q}_{12} \circ \mathsf{A}$, where all three $\mathsf{A}''(.)$, $\mathsf{A}'(.)$, and $\mathsf{A}(.)$ are 4-bit affine bijections. Thanks to the classifications given in [5] a uniform first-order TI of $\mathcal{Q}_{12}$ can be achieved by *direct sharing*. For $\mathcal{Q}_{12}$:`0123456789CDEFAB` we can write

$$e = a, \qquad f = b + bd + cd, \qquad g = c + bd, \qquad h = d, \tag{1}$$

with $\langle a, b, c, d\rangle$ the 4-bit input, $\langle e, f, g, h\rangle$ the 4-bit output, and a and e the least significant bits.

The component functions of the uniform first-order TI of $\mathcal{Q}_{12}$ can be derived by $f^{i,j}_{\mathcal{Q}_{12}}(\langle a^i, b^i, c^i, d^i\rangle, \langle a^j, b^j, c^j, d^j\rangle) = \langle e, f, g, h\rangle$ as

$$\begin{aligned} e &= a^i, & f &= b^i + b^j d^j + c^j d^j + d^j b^i + d^j c^i + b^j d^i + c^j d^i, \\ g &= c^i + b^j d^j + d^j b^i + b^j d^i, & h &= d^i. \end{aligned} \tag{2}$$

The three 4-bit output shares provided by $f^{2,3}_{\mathcal{Q}_{12}}(.,.)$, $f^{3,1}_{\mathcal{Q}_{12}}(.,.)$ and $f^{1,2}_{\mathcal{Q}_{12}}(.,.)$ make a uniform first-order TI of $\mathcal{Q}_{12}$. Since the affine transformations $(\mathsf{A}, \mathsf{A}', \mathsf{A}'')$ do not change the uniformity, by applying them on each 4-bit share separately we can construct a 3-share uniform first-order TI of the PRESENT S-box. Figure 1

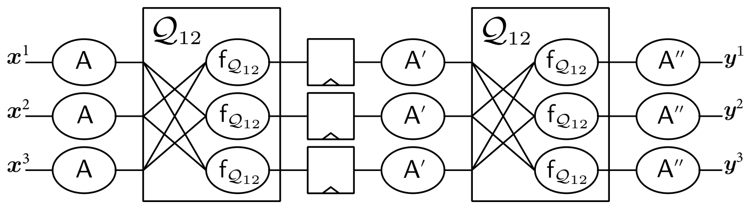


Fig. 1. A first-order TI of the PRESENT S-box

shows the graphical view of such a construction, and the detailed formulas of the component functions are given in Appendix A.

2.3 Affine Equivalence

In order to find such affine functions we give a pseudo code in Algorithm 1 which is mainly formed following [6]. The algorithm is based on precomputation of all 4×4 linear functions, i.e. 20 160 cases, each of which is represented by a 4×4 binary matrix with columns $(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3)$. Hence, each affine function $\mathsf{A}(.)$ is considered as a matrix multiplication followed by a constant addition $\mathsf{A}(\boldsymbol{x}) = [\boldsymbol{c}_0\ \boldsymbol{c}_1\ \boldsymbol{c}_2\ \boldsymbol{c}_3] \cdot \boldsymbol{x} \oplus \boldsymbol{c}$.

Algorithm 1: Find affine equivalent triples

```
Input  : $\mathcal{L}^4$: all $4 \times 4$ linear permutations,
         S: targeted S-box,
         F, G: targeted functions
Output: $\mathcal{A}$: all $(\mathsf{A}, \mathsf{A}', \mathsf{A}'')$ as $\mathsf{S} : \mathsf{A}'' \circ \mathsf{G} \circ \mathsf{A}' \circ \mathsf{F} \circ \mathsf{A}$
$\mathcal{A} \leftarrow \emptyset$
for $\forall \boldsymbol{L} \in \mathcal{L}^4$, $\forall \boldsymbol{c} \in \{0,1\}^4$ do
    form affine A by $\boldsymbol{L}$ and constant $\boldsymbol{c}$
    for $\forall \boldsymbol{L}' \in \mathcal{L}^4$, $\forall \boldsymbol{c}' \in \{0,1\}^4$ do
        form affine A′ by $\boldsymbol{L}'$ and constant $\boldsymbol{c}'$
        $\boldsymbol{c}'' \leftarrow \mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(0)\right)\right)\right)\right)$
        $\boldsymbol{c}''_1 \leftarrow \mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(1)\right)\right)\right)\right) \oplus \boldsymbol{c}''$
        $\boldsymbol{c}''_2 \leftarrow \mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(2)\right)\right)\right)\right) \oplus \boldsymbol{c}''$
        $\boldsymbol{c}''_3 \leftarrow \mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(4)\right)\right)\right)\right) \oplus \boldsymbol{c}''$
        $\boldsymbol{c}''_4 \leftarrow \mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(8)\right)\right)\right)\right) \oplus \boldsymbol{c}''$
        form affine $\mathsf{A}''^{-1}$ by columns $(\boldsymbol{c}''_1, \boldsymbol{c}''_2, \boldsymbol{c}''_3, \boldsymbol{c}''_4)$ and constant $\boldsymbol{c}''$
        if $\forall \boldsymbol{y} \in \{0,1\}^4 \setminus \{0,1,2,4,8\}$, $\mathsf{G}\left(\mathsf{A}'\left(\mathsf{F}\left(\mathsf{A}\left(\mathsf{S}^{-1}(\boldsymbol{y})\right)\right)\right)\right) \overset{?}{=} \mathsf{A}''^{-1}(\boldsymbol{y})$ then
            derive affine A″ as the inverse of $\mathsf{A}''^{-1}$
            $\mathcal{A} \leftarrow \mathcal{A} \cup \{(\mathsf{A}, \mathsf{A}', \mathsf{A}'')\}$
        end
    end
end
```

Table 1. The number of existing affine triples for different compositions

Decomposition	***No. of Triples***	$\#(\mathsf{A})$	$\#(\mathsf{A}')$	$\#(\mathsf{A}'')$	$\#(\boldsymbol{L})$	$\#(\boldsymbol{L}')$	$\#(\boldsymbol{L}'')$
$\mathcal{Q}_{12} \circ \mathcal{Q}_{12}$	147 456	384	36 864	384	48	2 304	48
$\mathcal{Q}_{294} \circ \mathcal{Q}_{299}$	229 376	512	57 344	448	56	3 584	64
$\mathcal{Q}_{299} \circ \mathcal{Q}_{294}$	229 376	448	57 344	512	64	3 584	56
$\mathcal{Q}_{299} \circ \mathcal{Q}_{299}$	200 704	448	50 176	448	56	3 136	56

Given the PRESENT S-box and $\mathsf{f} = \mathsf{g} = \mathcal{Q}_{12}$ the algorithm finds 147 456 such 3-tuple affine bijections $(\mathsf{A}, \mathsf{A}', \mathsf{A}'')$. Table 1 lists the number of found affine triples for each of the aforementioned decompositions.

3 Design Considerations

This section briefly demonstrates the architecture the PRESENT TI which we have implemented. Afterwards, different approaches for generating and exchanging affine triples are presented and compared.

3.1 Threshold Implementation of PRESENT cipher

PRESENT is a lightweight symmetric block cipher with a block size of 64 bits and either 80-bit or 128-bit security level (i.e., key size). The encryption of a plaintext is based on a Substitution-Permutation (S/P) network always taking 31 rounds and 32 sub-keys to compute the ciphertext (independently of the security level). The only difference between PRESENT-80 and PRESENT-128 is in the key schedule function to derive the sub-keys from the initial 80-bit or 128-bit key. Figure 2 gives an overview of our hardware architecture implemented on an Xilinx Spartan-6 FPGA. We opted to implement the PRESENT encryption scheme in a round-based manner along with the 128-bit key schedule variant. The sub-keys are derived on-the-fly. The substitution layer uses the first-order TI of the PRESENT S-box shown in Figure 1 and implements 16 S-boxes in parallel before the permutation is applied bitwise to all 64-bit states. Due to the additional register stage within the TI S-box each round requires two clock cycles.

As stated in Section 2.3, given a certain decomposition there exist many triple affine functions to realize a uniform first-order TI of the PRESENT S-box. Our goal is to randomly change such affine functions on the fly, that it first does not affect the correct functionality of the S-box, and second randomizes the intermediate values – particularly the shared $\mathcal{Q}_{12}$ inputs – with the aim of hardening higher-order attacks. As shown in Figure 2 all S-boxes share the same affine triple. In other words, at the start of each encryption an affine triple is randomly selected, and all S-boxes are configured accordingly. Although it is possible to change the affines more frequently, we kept the selected affines for an entire encryption process. To this end, we need an architecture to derive the

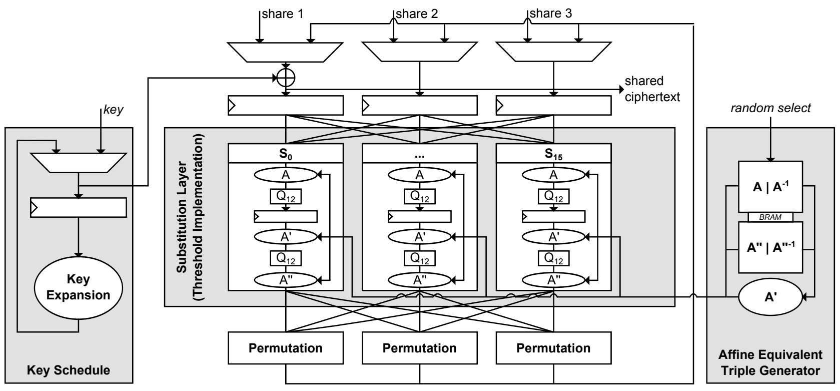


Fig. 2. Architecture of the PRESENT encryption design

affine triples randomly. Below we discuss about different ways to realize such a part of the design.

3.2 Searching for the Affine Triples

At a first step, we decided to implement Algorithm 1 as a hardware circuit which searches for the affine triples in parallel to the encryption. The found affine triples are stored into a "First In, First Out" (FIFO) memory, and prior to each encryption one affine triple is taken from the FIFO with which the corresponding part of the TI S-boxes are configured. If the FIFO is empty, the previous affine triple is used again. Due to the fact that the search is not time-invariant, i.e., new affine triples are not found periodically, some affines are used multiple times in a row while others are only used once. Since the efficiency of SCA countermeasures depends on the uniformity of the used randomness, such an implementation may not achieve the desired goal (i.e., hardening the higher-order attacks) if certain affines are used more often that the others. One solution to find affine triples more often is to run the search circuit with a higher clock frequency compared to that of the encryption circuit. Although this measure is limited, it at least alleviates the problem of changing S-boxes not periodically. On the other hand, if affine triples are found too fast this may cause a FIFO overflow. In this case either some search results should be ignored or the search circuit should be stopped requiring some additional control logic.

3.3 Selecting Precomputed Affine Triples

As stated in Table 1, considering the decomposition $\mathcal{Q}_{12} \circ \mathcal{Q}_{12}$, there exist 147 456 triple affines $(\mathsf{A}, \mathsf{A}', \mathsf{A}'')$. Each single affine transformation is a 4-bit permutation, and it can be represented as a look-up table containing sixteen 4-bit entries which requires 64 bits of memory. This results in 27 Mbit memory in order to store all the affine triples. However, the employed Xilinx Spartan-6 FPGA (LX75) offers only 3 Mbit storage in terms of general purpose block memory (BRAM). Therefore, alternative approaches to generate the affine equivalent triples are necessary.

Instead of storing the affines in a look-up table, in the second option we represent an exemplary affine $\mathsf{A}(\boldsymbol{x}) = \boldsymbol{L} \cdot \boldsymbol{x} \oplus \boldsymbol{c}$, with $\boldsymbol{x}$ as a 4-bit vector, $\boldsymbol{L}$ a 4×4 binary matrix and $\boldsymbol{c}$ a 4-bit constant. In this case, only the binary matrix and the constant need to be stored which reduces the memory requirements to 20 bits per affine. However, still more than 8 Mbit memory are necessary to store all affine triples. Therefore, we could store only a fraction of all possible affine triples. As an example, 16 384 affine triples occupy 60 BRAMs of the Spartan-6 (LX75) FPGA.

3.4 Generating Affine Triples On-the-fly

A detailed analysis of the affine triples led to interesting observations. First, the number of affine triples depends on the components in the underlying decomposition. For instance, in case of $\mathcal{Q}_{299} \circ \mathcal{Q}_{299}$ 448×448 and in case of $\mathcal{Q}_{299} \circ \mathcal{Q}_{294}$ 448×512 affine triples exist (see Table 1). Second, the total number of affine triples is limited by the number of unique input affines A and the number of output affines A'' such that $|\mathsf{A}| \times |\mathsf{A}''|$ gives the number of corresponding affine triples. This means that all affine triples of a decomposition can be generated by combining all A with all A''. Furthermore, we have observed that all affines A (for each decomposition) consist of a few linear matrices combined with certain constants. In particular, in case of the decomposition $\mathcal{Q}_{12} \circ \mathcal{Q}_{12}$ the 384 input affines A are formed by 48 binary matrices $\boldsymbol{L}$ each of which combined with 8 different constants $\boldsymbol{c} \in \{0, \ldots, 7\}$ or $\boldsymbol{c} \in \{8, \ldots, 15\}$. Indeed the same holds for the 384 output affines A'' which are made of 48 binary matrices $\boldsymbol{L}''$ by constants $\boldsymbol{c} \in \{0, 1, 4, 5, 10, 11, 14, 15\}$ or $\boldsymbol{c} \in \{2, 3, 6, 7, 8, 9, 12, 13\}$. Therefore, it is sufficient to store only all relevant binary matrices $\boldsymbol{L}$ and $\boldsymbol{L}''$ in addition to a single bit indicating to which group their constants belong to. Hence, in total $48 \times 2 \times (16+1) = 1632$ bits of memory (fitting into a single BRAM) are required to store all necessary data. Even better, by arranging the binary matrices in the memory smartly the group of the corresponding constants can be derived from the address where the binary matrix is stored.

Given two input and output affines A and A'', we need to derive the middle affine A'. To this end, an approach similar to Algorithm 1 can be used. If we represent the middle affine as $\mathsf{A}'(\boldsymbol{x}) = \boldsymbol{L}' \cdot \boldsymbol{x} \oplus \boldsymbol{c}'$, the constant $\boldsymbol{c}$ and the columns $(\boldsymbol{c}'_1, \boldsymbol{c}'_2, \boldsymbol{c}'_3, \boldsymbol{c}'_4)$ of the binary matrix $\boldsymbol{L}$ can be derived as

$$\boldsymbol{c}' = {\mathcal{Q}_{12}}^{-1}\left(\mathsf{A}''^{-1}\left(\mathsf{S}\left(\mathsf{A}^{-1}\left({\mathcal{Q}_{12}}^{-1}(0)\right)\right)\right)\right) \tag{3}$$

$$\boldsymbol{c}'_1 = {\mathcal{Q}_{12}}^{-1}\left(\mathsf{A}''^{-1}\left(\mathsf{S}\left(\mathsf{A}^{-1}\left({\mathcal{Q}_{12}}^{-1}(1)\right)\right)\right)\right) \oplus \boldsymbol{c}' \tag{4}$$

$$\boldsymbol{c}'_2 = {\mathcal{Q}_{12}}^{-1}\left(\mathsf{A}''^{-1}\left(\mathsf{S}\left(\mathsf{A}^{-1}\left({\mathcal{Q}_{12}}^{-1}(2)\right)\right)\right)\right) \oplus \boldsymbol{c}' \tag{5}$$

$$\boldsymbol{c}'_3 = {\mathcal{Q}_{12}}^{-1}\left(\mathsf{A}''^{-1}\left(\mathsf{S}\left(\mathsf{A}^{-1}\left({\mathcal{Q}_{12}}^{-1}(4)\right)\right)\right)\right) \oplus \boldsymbol{c}' \tag{6}$$

$$\boldsymbol{c}'_4 = {\mathcal{Q}_{12}}^{-1}\left(\mathsf{A}''^{-1}\left(\mathsf{S}\left(\mathsf{A}^{-1}\left({\mathcal{Q}_{12}}^{-1}(8)\right)\right)\right)\right) \oplus \boldsymbol{c}' \tag{7}$$

Obviously, this requires the inverse of both A and A''. Since it is not efficient to derive such inverse affines on the fly, we need to store all binary matrices $\boldsymbol{L}^{-1}$ and $\boldsymbol{L}''^{-1}$ in addition to all $\boldsymbol{L}$ and $\boldsymbol{L}''$. Fortunately, all such binary matrices (requiring 3 kbits) still fit into a single 16-kbit BRAM of Spartan-6 FPGA. It is noteworthy that the constant of each inverse affine can be computed by $\boldsymbol{L}^{-1} \cdot \boldsymbol{c}$.

In summary, at the start of each encryption two $\boldsymbol{L}$ and $\boldsymbol{L}''$ (each of which from a set of 48 cases) are randomly selected, that needs $6 + 6$ bits of randomness[1]. In addition, $3 + 3$ random bits are also required to form constants $\boldsymbol{c}$ and $\boldsymbol{c}''$. As exampled before, one bit of each constant should be additionally saved or derived from the address of the binary matrix. Therefore – excluding the masks required to represent the plaintext in a 3-share form for the TI design – in total 18 bits randomness is required for each encryption.

For ASIC platforms, where block memories are not easily available, an alternative is to derive the content of binary matrices $\boldsymbol{L}$ and $\boldsymbol{L}''$ as Boolean functions over the given random bits. Hence, a fully combinatorial circuit can provide the input and output affines followed (as before) by a module which retrieves the middle affine.

3.5 Comparison

Table 2 gives an overview of the design of the three above-mentioned approaches to derive the affine triples. The table reports the area overhead, reconfiguration time, and coverage of the affines' space. Comparing the first naive approach (of searching the affine triples in parallel to the encryption) to the approach of precomputing affine triples, the logic requirements could be dramatically decreased at cost of additional memory. In addition, the amount of affine triples that are covered is limited potentially reducing the security gain. We should note that the 20 BRAMs used in the "Search" approach are due to the space required to store all 4×4 linear permutations $\mathcal{L}^4$ required to run Algorithm 1 (excluding those required for the FIFO). The last approach where the affine triples are generated on-the-fly seems to be the best choice. It not only leads to the least area overhead (both logic and memory requirements) but also covers the whole number of possible affine triples.

We should note that our design needs a single clock cycle to derive the middle affine A'. Indeed the 114 LUTs (reported in Table 2) are mainly due to realization of the Equations (3)-(7) in a fully combinatorial fashion.

Further, with respect to the design architecture of the encryption function (Figure 2) the quadratic component functions of $\mathcal{Q}_{12}$ are implemented by look-up tables (LUTs), and the affine functions by fully combinatorial circuits realizing the binary matrix multiplication (AND operations) and XOR with the constant. Therefore, given (16+4) bits as the content of the binary matrix and the constant, the circuit does not need any extra clock cycles for configuration. Table 2 also gives an overview of the area and speed overhead of our design compared to a similar designs. For the first reference, the TI S-box is implemented by the

[1] For each selection $\in \{1, \ldots 48\}$ reject sampling with 6-bit random should be used.

Table 2. Area and time overhead of different design approaches

Section/Method/ Module	**Resource Utilization** *Logic* *(LUT)*	*Memory* *(FF)*	*(BRAM16)*	**Reconfig. Time** *(Cycles)*	**Affine Coverage** *(Percent)*	**Max. Freq.** *(MHz)*	**Order of TI**
3.2/Search	562	250	20	16	100.0	-	-
3.3/Precompute	204	0	60	0	11.1	-	-
3.4/Generate	114	20	1	1	100.0	-	-
Encryption [this work]	1720	722	0	-	-	112	1st
Encryption [16]	641	384	0	-	-	218	1st
Encryption [14]	808	384	0	-	-	207	1st
Encryption [14]	2245	1680	0	-	-	204	2nd

design of [16] (i.e., without any random affine). The second reference implements both a first-order and a second-order TI S-box for PRESENT in a similar fashion (using $\mathcal{Q}_{294}$ and $\mathcal{Q}_{299}$ instead of $\mathcal{Q}_{12}$) but with fixed affine transformations. The numbers for the encryption function exclude the PRNG as well as the circuit which finds/derives the affines. Due to the extra logic to support arbitrary affines, our design is certainly larger and slower.

4 Evaluation

We employed a SAKURA-G platform [1] equipped with a Spartan-6 FPGA for practical side-channel evaluations using the power consumption of the device. The power consumption traces have been measured and recorded by means of a digital oscilloscope with a $1\,\Omega$ resistor in the V_{dd} path and capturing at the embedded amplifier of the SAKURA-G board. We sampled the voltage drop at a rate of 500 MS/s and a bandwidth limit of 20 MHz while the design was running at a low clock frequency of 3 MHz to reduce the noise caused by overlapping of the power traces.

4.1 Non-Specific Statistical *t*-test

In order to evaluate the resistance or vulnerabilities of our designs against higher-order side-channel attacks we applied the well-known state-of-the-art leakage assessment metric called *Test Vector Leakage Assessment* (TVLA) methodology. This evaluation scheme is based on the Welch's (two-tailed) *t*-test and also known as *fix vs. random* or *non-specific* *t*-test. For further details, particularly how to apply this assessment tool for higher-order leakages as well as how to implement it efficiently in particular for large-scale investigations, we refer the reader to [19] giving detailed practical instructions. In short, we should note that such an assessment scheme examines the existence of leakage at a certain order without giving any reference to whether the detected leakage is exploitable by an attack. However, if the test reports no detectable leakage, it can be concluded that –

with a high level of confidence – the device under test does not exhibit any exploitable leakage.

4.2 Results

In this section we present the result of the side-channel evaluations concerning the efficiency of our introduced approaches to avoid higher-order leakages. In order to solely evaluate the influence of randomly exchanging the affine triples we considered a single design in our evaluations. As a reference, the design is kept running with a constant affine triple[2], and its evaluation results are compared to the case where the affine triples are randomly changed prior to each encryption. Note that in both cases (constant affine and random affine) the PRNG which provides masks for the initial second-order masking (with three shares) is kept active. In other words, both designs – based on the TI concept – are expected to provide first-order resistance, and their difference should be in exhibiting higher-order leakages.

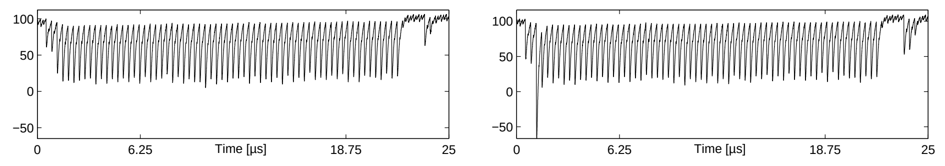
In Section 3 we introduced three different approaches to derive affine triples. Due to the issues and limitation of both first approaches, we have included the practical evaluation results of only the third option in Section 3.4, i.e., generating affine triples on-the-fly, which covers all possible affine triples.

Figure 3 shows two sample traces corresponding to the cases where the affine triple is constant or random. The main difference between these two traces can be seen by a large power peak at the beginning of the trace belonging to the random affines. Such a peak indicates the corresponding clock cycle where the random affine is selected and the middle affine is computed (as stated in Section 3.5, it is implemented by a fully combinatorial circuit). The first-order, second-order and third-order t-test results are shown in Figures 4-6 respectively for both constant and random affine. As expected, both designs do not exhibit any first-order leakage confirming the validity of our setup and designs. However, changing the affine triples randomly could avoid the second- and third-order leakage from being detectable. This can be seen in Figure 5 and Figure 6. We should highlight that the evaluations of the design with a constant affine have been performed by 50 million traces while we continued the measurements and evaluations of the design with random affines up to 200 million traces.

5 Discussions

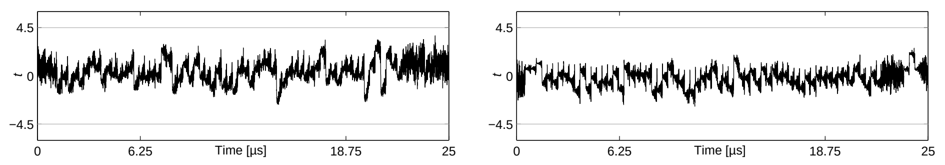
The scheme, which we have introduced here to harden higher-order attacks, at the first glance seems to just add more randomness to the design. We should stress that our approach is not the same as the concept of *remasking* applied in [2, 5, 13]. Remasking (or mask refreshing) can be done e.g., by adding two new fresh random masks $\boldsymbol{r}^1$ and $\boldsymbol{r}^2$ to the input of the TI S-box in Figure 1 as $(\boldsymbol{x}^1 \oplus \boldsymbol{r}^1, \boldsymbol{x}^2 \oplus \boldsymbol{r}^2, \boldsymbol{x}^3 \oplus \boldsymbol{r}^1 \oplus \boldsymbol{r}^2)$. Since our construction of the PRESENT TI S-box

[2] This has been easily done by fixing the corresponding 18 random bits.



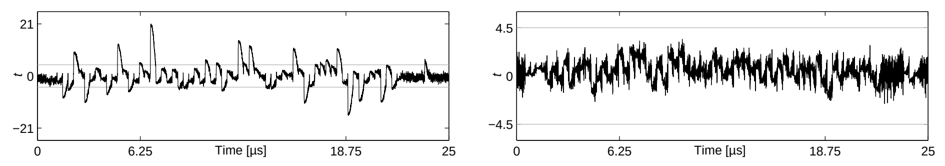
(a) Constant affine (b) Random affines

Fig. 3. Sample traces of the PRESENT encryption function



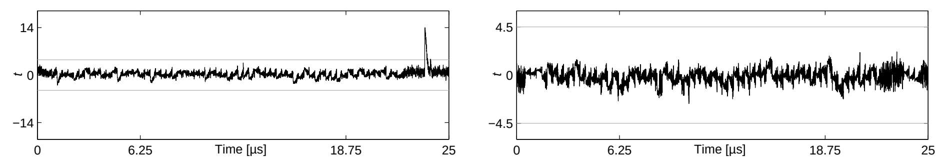
(a) Constant affine, 50 million traces (b) Random affines, 200 million traces

Fig. 4. Non-specific t-test: first-order evaluation results



(a) Constant affine, 50 million traces (b) Random affines, 200 million traces

Fig. 5. Non-specific t-test: second-order evaluation results



(a) Constant affine, 50 million traces (b) Random affines, 200 million traces

Fig. 6. Non-specific t-test: 3rd-order evaluation results

fulfills the uniformity, such a remasking does not have any effect on the practical security of the design as both $(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$ and $(\boldsymbol{x}^1 \oplus \boldsymbol{r}^1, \boldsymbol{x}^2 \oplus \boldsymbol{r}^2, \boldsymbol{x}^3 \oplus \boldsymbol{r}^1 \oplus \boldsymbol{r}^2)$ are 3-share representations of $\boldsymbol{x}$. In contrast, in our approach e.g., the input affine A randomly changes. Hence the input of the first $\mathcal{Q}_{12}$ function is a 3-share representation of $\mathsf{A}(\boldsymbol{x})$. Considering a certain $\boldsymbol{x}$, random selection of the input affine leads to random $\mathsf{A}(\boldsymbol{x})$ which is also represented by three Boolean shares. Therefore, the intermediate values of the S-box (at both stages) are not only randomized but also uniformly shared. As a result, hardening both second- and third-order attacks which make use of the leakage of the S-box can be justified. Note that since the S-box output stays valid as a Boolean shared representation of $\mathsf{S}(\boldsymbol{x})$ and random affine triples do not affect the PLayer (of the PRESENT cipher), the key addition and the values stored in the state register, our approach is not expected to harden third-order attacks that target the leakage of these modules. However, our construction (which is a combination of masking and hiding) allows to achieve the presented efficiencies with low number of (extra) required randomness, i.e., 18 bits per encryption. Indeed, our approach might be seen as a form of shuffling which can be applied on the order of S-box executions in a serialized architecture. However, our construction is independent of the underlying architecture (serialized versus round-based) and allows hiding the exploitable higher-order leakages in a systematic way.

References

1. Side-channel AttacK User Reference Architecture. `http://satoh.cs.uec.ac.jp/SAKURA/index.html`.
2. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A More Efficient AES Threshold Implementation. In *AFRICACRYPT 2014*, volume 8469 of *LNCS*, pages 267–284. Springer, 2014.
3. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In *ASIACRYPT 2014*, volume 8874 of *LNCS*, pages 326–343. Springer, 2014.
4. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold Implementations of All 3 ×3 and 4 ×4 S-Boxes. In *CHES 2012*, volume 7428 of *LNCS*, pages 76–91. Springer, 2012.
5. 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.
6. A. Biryukov, C. D. Cannière, A. Braeken, and B. Preneel. A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 33–50. Springer, 2003.
7. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.
8. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
9. T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme. In *CRYPTO 2008*, volume 5157 of *LNCS*, pages 203–220. Springer, 2008.
10. G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011. `http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf`.
11. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
12. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards.* Springer, 2007.

13. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.
14. A. Moradi and A. Wild. Assessment of hiding the higher-order leakages in hardware - what are the achievements versus overheads? *To appear in the proceedings of CHES 2015*, 2015.
15. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.
16. A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling. Side-Channel Resistant Crypto for Less than 2,300 GE. *J. Cryptology*, 24(2):322–345, 2011.
17. J. R. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely. Partitioning Attacks: Or How to Rapidly Clone Some GSM Cards. In *IEEE Symposium on Security and Privacy 2002*, pages 31–41. IEEE Computer Society, 2002.
18. O. Reparaz. A note on the security of Higher-Order Threshold Implementations. Cryptology ePrint Archive, Report 2015/001, 2015. `http://eprint.iacr.org/`.
19. T. Schneider and A. Moradi. Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. Cryptology ePrint Archive, Report 2015/207, 2015. `http://eprint.iacr.org/`.
20. Y. Zhou, Y. Yu, F. Standaert, and J. Quisquater. On the Need of Physical Security for Small Embedded Devices: A Case Study with COMP128-1 Implementations in SIM Cards. In *Financial Cryptography 2013*, volume 7859 of *LNCS*, pages 230–238. Springer, 2013.

A Necessary component functions for a first-order TI of PRESENT S-box

$$\begin{aligned}
\boldsymbol{y}^1 &= f^{2,3}_{\mathcal{Q}_{12}}(\langle a^2, b^2, c^2, d^2\rangle, \langle a^3, b^3, c^3, d^3\rangle) = \langle e, f, g, h\rangle\\
e &= a^2, \qquad f = b^2 + b^3d^3 + c^3d^3 + d^3b^2 + d^3c^2 + b^3d^2 + c^3d^2,\\
g &= c^2 + b^3d^3 + d^3b^2 + b^3d^2, \quad h = d^2.
\end{aligned} \tag{8}$$

$$\begin{aligned}
\boldsymbol{y}^2 &= f^{3,1}_{\mathcal{Q}_{12}}(\langle a^3, b^3, c^3, d^3\rangle, \langle a^1, b^1, c^1, d^1\rangle) = \langle e, f, g, h\rangle\\
e &= a^3, \qquad f = b^3 + b^1d^1 + c^1d^1 + d^1b^3 + d^1c^3 + b^1d^3 + c^1d^3,\\
g &= c^3 + b^1d^1 + d^1b^3 + b^1d^3, \quad h = d^3.
\end{aligned} \tag{9}$$

$$\begin{aligned}
\boldsymbol{y}^3 &= f^{1,2}_{\mathcal{Q}_{12}}(\langle a^1, b^1, c^1, d^1\rangle, \langle a^2, b^2, c^2, d^2\rangle) = \langle e, f, g, h\rangle\\
e &= a^1, \qquad f = b^1 + b^2d^2 + c^2d^2 + d^2b^1 + d^2c^1 + b^2d^1 + c^2d^1,\\
g &= c^1 + b^2d^2 + d^2b^1 + b^2d^1, \quad h = d^1.
\end{aligned} \tag{10}$$