Constructing TI-Friendly Substitution Boxes using Shift-Invariant Permutations*

Si Gao, Arnab Roy, and Elisabeth Oswald

University of Bristol

Abstract. The threat posed by side channels requires ciphers that can be efficiently protected in both software and hardware against such attacks. In this paper, we proposed a novel Sbox construction based on iterations of shift-invariant quadratic permutations and linear diffusions. Owing to the selected quadratic permutations, all of our Sboxes enable uniform 3-share threshold implementations, which provide first order SCA protections without any fresh randomness. More importantly, because of the "shift-invariant" property, there are ample implementation trade-offs available, in software as well as hardware. We provide implementation results (software and hardware) for a four-bit and an eight-bit Sbox, which confirm that our constructions are competitive and can be easily adapted to various platforms as claimed. We have successfully verified their resistance to first order attacks based on real acquisitions. Because there are very few studies focusing on software-based threshold implementations, our software implementations might be of independent interest in this regard.

Keywords: Shift-invariant · Threshold implementation · Sbox

1 Introduction

In the past decade, side channel analysis (SCA) has become a serious threat to various cryptographic devices. In this adversarial model, an attacker may observe information leakage from a device operating some key-related information. For cryptographic engineers, efficiently implementing a good cipher is then no longer enough. They must also mitigate against the threat of such leakage and integrate a proper countermeasure, which often is a non-trivial task.

Since they were proposed, Threshold Implementations (TI) [1, 2] have become a recognised countermeasure for power analysis [3–7] when hardware implementations are considered. Unlike Boolean masking schemes [8, 9], TI requires more shares, but the "non-completeness" property of TI ensures that in each computation logic gate, at least one of the (input) shares is missing. As a consequence, even in the presence of hardware glitches, this missing share guarantees that the observed leakage will not give out information about any secret intermediate value [2] and thus robustly protects against so-called first-order attacks. In

* This is the full version of the paper accepted at the CT-RSA 2019 with the same title.

this paper, we only consider threshold implementations that provide first order protections.

One obstacle in threshold implementations is that there is no trivial efficient constructions for arbitrary cryptographic components. Take 3-share TI schemes for instance: in theory, any arbitrary quadratic function can be re-written in a TI-shared form with 3 shares. In practice, however, considering the requirement of "uniformity", a uniform 3-share TI scheme may not exist [2]. For smaller components (eg. Sboxes), this issue has been extensively studied up to affine equivalence [3, 10–12]. For larger components, there is no generic construction available. On the other hand, solutions for uniform TIs may exist with higher implementation costs, such as increasing the number of shares or adding fresh randomness. Recently, De Meyer, Moradi and Wegener proposed a bit-serialized implementation of the Sbox of AES [13]: although their implementation with AES can be easily deployed in many applications, it comes with the price of adding fresh randomness. Joan Daemen proposed a technique called "Changing of the Guards", which significantly eases the dilemma between uniformity and fresh randomness [14]. As the "Changing of the Guards" technique borrows randomness from the shares of other concurrent components, engineers no longer need to ensure uniformity for their TI schemes, as long as there are a few extra random bits available in the beginning of the encryption. Considering that the overhead of TI is already high, it is imperative to keep any extra cost as low as possible. Therefore, in this paper, we would like to avoid any fresh randomness and minimize the number of shares.

Instead of searching for efficient TI representations for existing Sboxes, we can also construct new Sboxes that are intrinsically suitable for TI protections. The TI forms of all 4×4 Sboxes were described in [3]. Boss el al. constructed several 8-bit Sboxes with round-based balanced Feistel, MISTY, SPNs structures, where the core building blocks are 4-bit Sboxes with easier TI protections [15]. The main focus of their paper was in finding Sboxes with efficient hardware TI implementations. However, the authors claimed their approach also "enables an efficient and low-cost implementation in software"(their "software implementation" refers to masked bitslice implementations, rather than TI-based software implementations). De Meyer and Varici further extended this approach to several new constructions (such as Generalized Feistel, Lai-Massey, Asymmetric SPN etc.) and provided implementation costs in terms of ASIC logic area [16].

It is not surprising that very few papers actually consider using TI-based software implementations. To the best of our knowledge, the only available TI constructions on software are TI-based PRESENT on an 8-bit micro-controller [17] and TI-based ARX ciphers [18]. The reason behind this is straightforward: the main concern that TI solves — glitches — do not exist in software[1]. The overhead of using TI-based countermeasures is usually much higher than using (bitsliced) masking. Thus in theory, there is little point in applying TI to software im-

[1] Technically, glitches still exist within one instruction. However, the threat that glitches may bring—mixing between different data shares—does not usually show up.

plementations. In practice however, it has been observed that d-order bitsliced maskings sometimes fail to provide d-order SCA protections. This is because the internal architecture of micro-processors is not publicly accessible. Now even if a cryptographic engineer carefully writes his/her code in assembly, some implicit operations/registers may still mix different shares and produce exploitable leakage such as demonstrated in [19, 20]. In the worst case, as Balasch el al. suggested, a d-order masking may only achieve $\lfloor \frac{d}{2} \rfloor$-order security in practice [21].

Our contribution. In this paper, we aim to find several Sboxes that come with easier first order TI protections, in both software and hardware platforms. In contrast to Boss el al.'s work [15] we use shift-invariant quadratic permutations instead of smaller Sboxes [22]. Similar to the χ^2 function [23], any coefficient Boolean function of these permutations is simply a "rotated" version of another. In other words, the bit-width of the elementary computation logic —which we called "granularity" in this paper — can be 1. Combined with the idea of serial threshold implementations [24, 25], the granularity of first order TI implementation can then be 1. Finer granularity brings more flexibility for cryptographic engineers, giving them more fine-grained trade-off options between executing time, logic area as well as power consumption. Specifically, the benefits of such protected Sboxes include:

- No fresh randomness.
- Easier software implementations. Since the shared version of our TI function preserves the "shift-invariant" property to some extent, bit-slicing such protected Sbox becomes easier.
- Flexible hardware implementations. As the granularity of such TI Sboxes is 1, in hardware, it is possible to implement only 1 computation unit, then get all other shared bits by shifting. Such strategy can lead to a very compact footprint, in the price of taking more cycles to execute.
- Full implementations/security evaluations. Despite the fact that all the implementations in this paper follow exactly the same rules as standard TI-s, we have verified these implementations with real-world acquisitions.

Outline. In Section 2 we explain a few essential concepts, including the cryptographic properties for Sboxes, the principle of threshold implementations and our Sbox searching strategies. Section 3 first introduces the concept of shift-invariance, then presents a search for quadratic TI-uniform shift-invariant permutations. Based on the results of this search, we further construct Sboxes with an SPN network. Section 4 and 5 discuss the possible implementation tradeoffs on software/hardware platforms, respectively. Section 6 presents TVLA-based security evaluation results on both an ARM M0 core and a Kintex 7 FPGA.

2 Preliminaries

2.1 Cryptanalytic properties for Sboxes

In a block cipher the Sboxes provide the desired non-linear properties. A newly constructed Sbox must be evaluated for cryptographic properties e.g. differential

uniformity, linearity, to thwart the differential and linear attacks. Let : $\mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ be a function.

Definition 1. *(**Differential uniformity** [26]) For any pair $(a, b) \in \mathbb{F}_{2^n}$, define the set*

$$\mathsf{D}^F(a \to b) = \{x \in \mathbb{F}_{2^n} | F(x \oplus a) \oplus F(x) = b\}.$$

The differential uniformity of F is defined as $\delta(F) := \max_{a \neq 0, b} |\mathsf{D}^F(a \to b)|$ where the $|\mathsf{D}^F(a \to b)|$ denotes the cardinality of the set $\mathsf{D}^F(a \to b)$ and is determined by the entry at the position (a, b) in the difference distribution table of F.

The Walsh transformation of the function F is defined as $W : \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \to \mathbb{Z}$ and is given as

$$W^F(a, b) = \sum_{x \in \mathbb{F}_{2^n}} (-1)^{a \cdot x + b \cdot F(x)}.$$

The linearity of an Sbox gives a measure of its best linear approximation. The linearity of F is defined as follows,

Definition 2. *The linearity of F is defined as $\mathsf{L}(F) = \max_{a,b \neq 0} W^F(a, b)$.*

Besides, an Sbox should not have any algebraic properties e.g low degree of the polynomial, which may be exploited by an adversary to mount an attack. It is known that the maximum algebraic degree of an m-bit permutation Sbox will be $m - 1$.

2.2 Threshold Implementation

In side channel research,threshold implementation (TI) usually refers to a countermeasure that based on secret sharing. For an $m \times n$ vectorial Boolean function f where each input x is shared as an s-length vector $\mathbf{x} = \left(x^{(1)}, .., x^{(s)}\right)$, TI implements a few shared functions $f^{(j)}$ that satisfy:

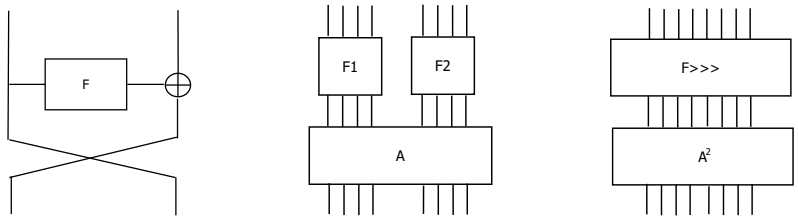
- Correctness. The sum of all shared functions is equal to the original unshared function f (i.e. $\sum_{j=1}^{s} f^{(j)} = f$).
- Non-completeness. Every shared function $f^{(j)}$ is independent of at least one share of x. Specifically, for a d-order TI scheme, the combination of d $f^{(j)}$ functions is still independent of at least one share.
- Uniformity. For any unshared input value $x = x^{(1)} \oplus x^{(2)} \oplus ... \oplus x^{(s)}$, the corresponding output shares $\mathbf{y} = \left(y^{(1)}, .., y^{(s)}\right)$ are uniformly distributed on all $\mathbf{y}$-s that satisfy $f(x) = y^{(1)} \oplus y^{(2)} \oplus ... \oplus y^{(s)}$.

To ensure uniformity for permutations ($m = n$), we can simply check if the shared version of f is an $m \times s$-bit permutation [3] (or prove it is invertible [14]).

2.3 Constructing TI Sboxes

To ensure non-completeness, threshold implementations need more shares for Boolean functions with higher degrees. As the implementation cost increases with the number of shares, the cheapest protected non-linear functions are quadratic ($deg = 2$) Boolean functions. For Sbox constructions, it is favourable to use permutations rather than arbitrary quadratic vectorial Boolean functions. Previous studies have sucessfully found uniform TI schemes for many quadratic permutations, including 3×3 and 4×4 Sboxes [3], 5-bit permutations [27] as well as a few observations on 6-bit quadratic permutations [28].

All the results above serve as a perfect building block for larger Sboxes: although directly applying TI is difficult, we can always use smaller Sboxes/quadratic permutations with known TIs to build large Sboxes. Boss et al. started searching for 8-bit Sboxes with Feistel (Figure 1(a)), SPN (Figure 1(b)), and MISTY structures, using 4-bit TI Sboxes as building blocks [15]. De Meyer and Varici extended this search to other constructions, such as Double Misty, Asymmetric SPN and Generalized Feistel structures [16].



(a) Feistel in [15] (b) SPN in [15] (c) this paper

Fig. 1. Structure overview

Since the building blocks are smaller Sboxes/permutations, such constructions give much more compact 8-bit Sboxes in hardware [15, 16]. Generally speaking, for an n-bit Sbox, its 3-share TI form would be a $3n$-bit permutation. Although each share can be computed with only 2 input shares ($2n$-bit), in hardware, increasing inputs usually boosts the area cost. Using smaller TI-Sboxes as building blocks significantly reduces the overall implementation cost, but it is unclear whether such constructions can provide flexibility when considering other platforms. Neither of these papers discusses the possibilities of serial TI—an extra trade-off proposed back in 2013 [24]. Boss et al.'s work did mention software implementations, yet their argument is that fewer AND gates lead to more efficient bit-sliced masking in software, rather than any TI protection [15]. None of these papers present security evaluations of their final implementations.

2.4 The notion of granularity

Irrespective of considering hardware or software implementations, constructions that feature multiple identical computation tasks usually give the cryptographic engineer more flexibility for the speed/cost trade-off. Taking hardware implementations for instance, all 4 bits in a PRESENT Sbox must be implemented with combinational logic, because all 4 bits are based on different Boolean functions [29]. Meanwhile, for the Keccak 5-bit χ^2 function, it is possible to implement only the circuit to do a 1 bit computation, as other 4 output bits can be computed through rotating the inputs [23] using the same circuit.

In this paper, we denote the output size of the smallest "gadget" to compute an Sbox as the "granularity". Clearly, the granularity for an unprotected PRESENT Sbox is 4, whereas for an unprotected 5-bit χ^2 function is 1. A finer granularity gives crypto engineers more opportunities for trade-offs: for instance, they can opt for a serial (slower) implementation, or a parallel (faster) implementation in hardware. Granularity also plays a critical role in software implementations. As most processors have intrinsic bit-widths (8,32 or 64), when performing bitwise operations, most of the bit-width will be wasted unless all the bits require the same operation. In order to take full advantage of the bit-width, a bit-slice implementation usually "slices" the same bits from multiple Sboxes to one register. As the CPU processes multiple Sboxes simultaneously, the overall throughput increases. Implementations with finer granularity provide intrinsic parallelism, which may take the most of the bit-width of our processors without manually "slicing" from a lot of concurrent data blocks (eg. Sboxes).

3 Constructing TI-Sboxes with better granularity

In this section, we present our TI-Sboxes search strategy. To achieve better implementation flexibility, we choose a different type of building blocks: instead of using 4 bit Sboxes with known TIs, our search utilizes the "Shift-invariant" [22] permutations. Such constructions usually lead to finer granularity (for each elemental operation) and give better implementation trade-offs for not only the Sbox itself, but also its TI-protection.

3.1 Shift-invariant: concept and previous works

Technically, an $n \times n$ vectorial Boolean function F is shift-invariant if for any rotated shift τ and any state x, $F(\tau(x))) = \tau(F(x))$ [22]. As stated in Daemen's thesis [22], "shift-invariant transformations can be implemented as an interconnected array of identical 1-bit output 'processors'"(granularity 1). Daemen further studied both linear and non-linear shift-invariant transformations, exploring their invertibility, local propagation and correlation properties [22]. As shift-invariance is closely linked to the concept of cellular automaton, Mariot, Picek , Leporati and Jakobovic searched up to 7×7 Sboxes from a cellular automaton perspective [30]. The most well known output of this direction is the χ^2 function in Keccak. However, it worth mentioning that without any other trick, χ^2 itself does not have a uniform 3-share TI.

3.2 Quadratic shift-invariant permutation with uniform TI

For an unprotected Sbox, shift-invariance ensures its granularity is equal to 1. However, considering the requirements of first order TI, its granularity also grows with the number of shares. Further reducing the granularity requires not only shift-invariance, but also its TI property: for any Boolean function f, if its direct shared form (i.e. Section 4.2 in [3]) is uniform, its granularity can be reduced to 1, using a serial TI implementation [24, 25]. Thus, for granularity, our best option would be using quadratic shift-invariant permutations with a uniform direct sharing threshold implementation.

Therefore, our main building blocks for Sbox constructions are quadratic shift-invariant permutations with uniform 3-share TI-s. Although Daemen's thesis gave many useful results, it did not cover all possible nonlinear shift-invariant transformations. Fortunately, the search space for common Sbox sizes ($n = 4$ or $n = 8$) is small enough. For $n \times n$ shift-invariant transformations, the number of all possible quadratic transformations are equal to the number of n-bit quadratic Boolean functions $2^{\sum_{i=0}^{2}\binom{n}{i}}$. The search space for 4 bit building blocks is 2^{11}, whereas for the 8 bit case is 2^{37}. Among these transformations, we are interested in those satisfy:

- The transformation itself is an n-bit permutation.
- Its direct 3-share TI is uniform.

Both properties are easy to check: for TI uniformity we simply check whether the shared form is still a $3n \times 3n$ permutation. For early abortion in this permutation check we first examine whether the coefficient Boolean function f is balanced. If it is not balanced, the transformation it derived cannot be a permutation. Additionally, we further limit our search to functions that satisfy:

- For bit y_0, its Boolean function always contains bit x_0. If not, we can always find a shift transformation τ that ensures $F' = F \circ \tau$ (F is the shift-invariant transformation f derived)[2]. For a shift-invariant F', τ and F are commutative. This means for lower rounds (1 or 2) of SPN network, τ can be integrated into the initial/final linear transformation, which does not affect the cryptographic properties.
- f does not have a constant term. For a shift-invariant transformation, the constant term can be either all-0 or all-1. As an all-1 constant has little impact on the cryptographic property of F, we simply discard these choices.

For 4-bit quadratic functions, we found that 960 out of 2048 functions contain x_0 and 0 as their constant terms. 400 of them are balanced, whereas only 28 f lead to a 4×4 permutation F. Fortunately, all of the direct 3-shares schemes are actually 12×12 permutations (i.e. satisfy uniformity).

On the other hand, for 8 bit permutations, the search space of f is 2^{37}. Almost half of the f-s have $x_0 = c = 0$, while only a quarter of f-s are balanced. 520 128 ($\approx 2^{19}$) can generate an 8-bit shift-invariant permutation F: interestingly, all of these permutations have uniform direct 3-share TI.

[2] Note that here we only need x_0 to appear, rather than appearing as a linear term [22].

n	All f	Has $x_0 \& c = 0$	Balanced	Permutation	TI Permutation
4	2048	960	400	28	28

Table 1. Shift-invariant quadratic TI permutations: $n = 4$

n	All f	Has $x_0 \& c = 0$	Balanced	Permutation	TI Permutation
8	2^{37}	68451041152	29986581632	520128	520128

Table 2. Shift-invariant quadratic TI permutations: $n = 8$

3.3 Constructing Sboxes

In this section, we further construct cryptographically good 4/8-bit Sboxes with these quadratic permutations. The Sbox search follows exactly the same strategy as previous works [15, 16], although the granularity further complicates the situation here.

Design Architectures As shift-invariance ensures each bit can be computed in the same way, generally speaking, we would like to avoid more branches. Take two-branch balanced Feistel structure for instance: although the round function may still have granularity 1, the other branch also contributes to the granularity for the whole Sbox. To this end, we perform our Sbox search with full range Substitution-Permutation Network (SPN) (Figure 1(c)).

Permutation Layer As the substitution layer is chosen from those quadratic TI permutations, the only decision left to make is the permutation layer. Clearly, the most efficient construction would be using shift-invariant linear permutation or nothing at all. Although shift-invariance is a good property for software/hardware implementations, considering the threat of rotational cryptanalysis [31], we prefer not to preserve it in the final Sbox. Thus, our linear transformation here needs to stop the propagation of shift-invariance. In general, the cheapest option would be using non-shift bit-permutations. However, a bit-permutation usually have a larger granularity (as each bit has to be implemented respectively), which leads to a penalty on its software performance. Instead, in this paper, we consider a linear transformation that is similar to AES's "xtime". More specifically, we search for invertible matrices that satisfy:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & 1 & 0 & \dots & 0 \\ a_{2,1} & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1,1} & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Let $\mathbf{a_1} = \{a_{1,1}, a_{1,2}, ..., a_{n-1,1}, 1\}$, if $\mathbf{A}$ is indeed invertible, in software, it can be implemented with a shift and a conditional XOR.

$$\mathbf{A}x = \begin{cases} (x << 1) \oplus \mathbf{a_1}, & \text{if } hsb(x) = 1 \\ (x << 1), & \text{otherwise} \end{cases}$$

As the conditional branch is prone to cache attack, most implementations tend to use a multiplication instruction to achieve a constant control flow

$$\mathbf{A}x = (x << 1) \oplus (\mathbf{a_1} \times hsb(x))$$

As the n-bit state x is operated as a word, the granularity is determined by this 1-bit multiplication: since this equation only holds 1 bit values, the overall granularity gets coarser. Nonetheless, from an implementation perspective, it is still much better than arbitrary binary matrix multiplication. To achieve a better diffusion property, in our Sbox search, we use two layers of $\mathbf{A}$ ($\mathbf{A}^2$) as our permutation layer.

Selection criteria. In order to achieve a balance between the implementation cost and the cryptographic properties, we have defined a selection criteria for the candidate Sboxes. Specifically, for 4-bit Sboxes,

- the differential uniformity is ≤ 4 and,
- the linearity is ≤ 8

For 8-bit Sboxes,

- the differential uniformity is ≤ 8 and,
- the linearity is ≤ 72

Besides, the algebraic degree and the degree of the interpolation polynomial should be large enough to resistent algebraic attack and interpolation attack, respectively.

3.4 Results

4-bit case. For 4-bit Sboxes, such selection criteria only accepts optimal Sboxes (differential uniformity= 4, linearity = 8) [32]. By enumerating all possible choices of A and quadratic permutations, we can find 16 such 4-bit Sboxes within 2 rounds. One such Sbox is presented as follow. The algebraic degree of this Sbox is 3, whereas the degree of the interpolation polynomial is 15.

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	9	4	A	3	7	8	C	5	B	6	D	E	F

Table 3. Shift-invariant quadratic TI permutation for S4

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	4	8	A	F	C	6	9	1	E	B	D	7	5	3	2

Table 4. Final Sbox for S4

8-bit case. For $n = 8$, the overall search space is around 2^{26}, which is quite feasible for most PCs. 6 Sboxes appear within 3 rounds: all of them have differential uniformity 8 whereas their linearity vary from 64 to 72. Due to the space limit, we present the best one (differential uniformity= 8, linearity = 64) in the Appendix. The algebraic degree of the presented Sbox is 6 and the degree of interpolation polynomial is 252.

4 Software Implementation

The major benefit of an Sbox with small granularity, is that it can be efficiently implemented in both software and hardware platforms. Although software based TIs tend to have higher overhead, in terms of security, they might have their own advantages [5]. In this section, we implement our selected Sboxes with first order TI protections in software and discuss a few possible trade-off options.

4.1 Target Platform

For software implementations, the most common platforms are smart cards or high-end processors (ARM/AMD/Intel). Although different processors may have different instruction sets, for bit-slice computations, most required bit-wise instructions can be found easily in all instruction sets. The major difference lies in the bit-width of the target processor, which determines how many bits can be computed in parallel. In this paper, our implementation chooses the most common bit-width—32. Implementations for 8-bit and 64-bit follow exactly the same rule. Because our target chip is an NXP ARM M0 core, we wrote our Sbox implementations using the Thumb instruction set [33]. In order to demonstrate the difference between Thumb and ARM instruction sets [34], we also show how those Sboxes can be computed on a more advanced core like the ARM M3.

4.2 Implementation Trade-offs

No optimization. It is worth mentioning that finer granularity only provides a possibility for further implementation trade-off: when such trade-off is not necessary, engineers can always do a TI implementation with $3n$ variables. Such an

implementation achieves its best performance when there are 32 concurrent data blocks (Sboxes) available. As the available bit-width is already fully occupied, the shift-invariant property will not provide any benefit in this case.

Size-based optimization. As each bit can be computed in the same way, with shift-invariant transformations, we can pack all n bits into one register. Take an 8 bit Sbox for instance, if there are 4 concurrent Sbox computations for $x^{[1]}$, $x^{[2]}$, $x^{[3]}$ and $x^{[4]}$, a 32-bit register can be filled with

$$\left(x_1^{[1]}, x_1^{[2]}, x_1^{[3]}, x_1^{[4]}, ..., x_8^{[1]}, x_8^{[2]}, x_8^{[3]}, x_8^{[4]}\right)$$

where x_i is the i-th bit of x. Correspondingly, each computation will be adjusted to ensure it takes the right input bit. Note that the rotated shift is still available in this form: instead of rotating 1 bit, now we are rotating 4 bits. Readers can verify that the transformation can still be computed correctly in this form, while the number of required concurrent data blocks shrinks from 32 to 4. Similar to the unprotected Sbox, the TI protection can be computed in exactly the same way. If all three shares are computed separately, such an optimization does not contradict with any TI requirement.

Extreme optimization. In theory, since the granularity of the TI protection is still 1, packing all 3 shares into one register is possible. Whether it contradicts with TI's security requirement (i.e. non-completeness) is debatable: ideally, if bit-wise instructions' leakage can be regarded as a sum of the leakages of all candidate bits (i.e. no "bit-interactions"), such implementation should be as secure as a hardware-based TI[3]. However, current results seem to suggest this may not always be the case: Sasdrich et al's work shows that for lookup tables (i.e. LDR instruction) on smart cards, bit-interaction clearly exists [5]. Our experiments with ARM M0 processors also prove the shift instructions (LSL,LSR,ROR) have the same issue. Moreover, as different bits and shares both get placed in one register, shifting becomes trickier. Only one of the shifts, whether shift bits or shares, can be operated with rotated shift instructions. The other one must be done manually with a few shifts and data masks. Considering the security loss and potential performance gain, we believe this is not a reasonable option.

4.3 Implementation on ARM M0/M3

Throughout this section, our evaluation is based on the size-based optimization. For the quadratic permutation S, we simply computed the TI-protected permutation according to its Algebraic Normal Form (ANF). Further customized optimizations may be possible but are out of the scope of this paper. To limit the usage of registers or memories, we compute all shifted results online, even

[3] Unlike its hardware counterpart, "coupling" effect [35] and "voltage fluctuation" are not the only concerns for the software TI. An AND instruction may not have the same leakage as 32 1-bit-AND gates, unless all the other combinational logic cells in the ALU are actually "silent".

if some of them appear repeatedly in the computation. Although this sounds far from ideal, as most commodity processors have a limited number of general purpose registers, such a compromise is inevitable in practice. For the linear transformation P, as the multiplication operation can only handle 1 bit at a time, all n-bit data shares must be executed one by one.

Despite the fact that our Sbox is computed online (rather than using pre-computed lookup tables), architecturally, its computation procedure is not that different from Sasdrich et al.'s implementation of PRESENT's Sbox [17]. Depending on the context, leakage might still show up when the CPU switches from one TI-shared function to another. Nonetheless, as the number of shared functions in TI is quite limited (compared with the number of AND-s in masking), implementing TI correctly requires much less effort than implementing bit-slice Boolean masking.

Table 5 illustrates the software implementation costs of our selected Sboxes, along with a few other well-known protected Sboxes, such as AES and PRESENT. It is not hard to see there is a significant performance difference between Thumb [33] and ARM [34] instruction sets. The major difference lies in rotation: as Thumb's ROR only shifts with a register rather than a constant, rotating r1 by n and storing the result in r2 has to be implemented as

MOV r3,#n
MOV r2,r1
ROR r2,r3

However, with the "Flexible Operand 2" [34] in ARM instruction set, such procedure can be implemented with only one line. In terms of executing cycles, implementations with ARM instructions have a significant bonus.

MOV r2,r1, ROR #n

	Size	Diff.	Lin.	Deg.	1st Order Protected		
					Randomness	Cycles	
						Thumb	ARM
PRESENT(BS) [36]	4	4	8	3	64	n/a	796/16
PRESENT($F \circ G$) [36]	4	4	8	3	128	n/a	686/8
$S4$ (our result)	4	4	8	3	0	870/8	654/8
AES(BS) [36]	8	4	32	7	512	n/a	4698/16
AES(KHL) [36]	8	4	32	7	192	n/a	2309/8
$S8$ (our result)	8	8	64	6	0	3627/4	2169/4

Table 5. Software Performance of various Sboxes

As the results in Table 5 are most likely parallel implementations for multiple Sboxes, we have listed the number of parallel Sboxes with the operation cycles. For our $S4$, Table 5 suggests it takes 870 cycles to compute 8 Sboxes

simultaneously. [4] For 4 bit Sboxes, our shift-invariant Sbox has similar performance as the PRESENT Sbox based on quadratic decomposition (654 v.s. 686). With bitslice masking, PRESENT Sbox can be much more efficient [36]. On the other hand, for the 8 bit case, both the KHL and bit-sliced masking are quite efficient, running twice faster than our shift-invariant Sbox. However, we would like to stress that the comparison of Table 5 is not as trivial as comparing the numbers of cycles. First of all, our implementation does not take any fresh randomness. As we can see in Table 5, all other Sboxes use quite a lot of random bits, even if they do not use any mask refreshing. Considering the cost of producing (pseudo)random numbers, it is clearly desirable to avoid fresh randomness. On the other hand, although all Sboxes in Table 5 claim first order security, a TI scheme has 3 shares whereas a bit-slice masking only has 2. Since the authors did not give any real traces based SCA evaluation [36], it is hard to argue whether these bit-sliced masking schemes provide the same security level as our threshold implementations. If we simply believe in the order-reduction theorem [21], a fair comparison would be using the second order bit-slice masking (3 shares), which degrades their performance to the same level of ours [36]. Last but not least, enormous effort has been invested in optimizing the implementations of both AES's Sbox and PRESENT's Sbox. In fact, the advantage of bit-slice masking is mainly inherited from the circuit optimization of the unprotected Sbox. On the contrary, we simply implemented the ANF of our shift-invariant Sboxes: further optimizations may be possible but they are out of the scope of this paper.

Another interesting observation would be our granularity gains. Technically, granularity determines how many concurrent Sbox computations we need to achieve the best possible throughput. For PRESENT and AES in Table 5, granularity does not cause an issue: both ciphers use SPN networks with many same Sboxes as their confusion layers. However, if the cipher uses smaller round functions with less concurrent Sboxes or a confusion layer with different Sboxes, it would be difficult to find enough data to "slice" within one plaintext block. Thanks to the fine granularity of our new Sboxes, in short encryption request, our construction has a better chance to reach its maximal throughput.

5 Hardware Implementation

5.1 Implementation Trade-off

Unlike software platforms, TI on hardware has been extensively studied for years. The only difference our Sboxes bring is a "double-rotating" feature: not only the 3 shares can be generated by rotating the inputs with the same circuit (i.e. serial TI [24]), all n-bit output can also be generated by rotating inputs. Note that these two rotations are different operations: one is rotating bits, the other is rotating shares. On software platforms, since there is only one rotation instruction,

[4] Note that this does not mean each Sbox can be computed only 109 cycles: if there is only one Sbox to compute, it will still take 870 cycles, as most of the bit-width will be wasted.

implementing both efficiently is not trivial. On hardware, double-rotation can be simply implemented with multiplexers. Thanks to the fine granularity, now we can implement only 1 bit Boolean function and compute the other $3n-1$ bits through rotations.

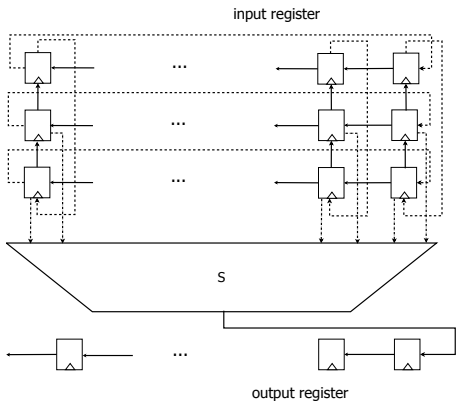


Fig. 2. Hardware schematic of shift invariant transformation S

As all other implementations are relatively trivial, in this section, our evaluation only uses this 1-bit serial implementation. Note that this implementation is by no means our "reference" design. The point of having a granularity 1 Sbox is that the engineers have the flexibility to choose the right trade-off. Although this 1-bit implementation leads to a very compact logic footprint, it trades area advantages with executing cycles. It takes $3 * n$ cycles to finish a 3-share n-bit Sbox computation. Besides, the multiple data paths cause the control logic to increase, which may compensate some of the footprint gain. Depending on the specific applications, engineers can also use a "single rotation" version, where only the shares or the bits are generated by rotations.

5.2 Pre-charge Issue

A well known issue for serial threshold implementation, is some first-order leakage might appear during the "shift-shares" procedure [37]. The reason behind is that the leakage for a combinational logic during an input transition depends on not only the current state, but also the previous state. The solution would be simply eliminating any transition of input shares in the combinational logic: i.e. add a pre-charge stage which charges the combinational logic with all zero between these two states. Obviously, this pre-charge stage penalizes the overall performance by one extra cycle. Interestingly, as our double-rotating design takes more cycles to proceed, the percentage of pre-charge time becomes smaller. Note that a pre-charge stage is only required when we are switching between different shares, not between different bits.

5.3 Implementation on ASIC

In order to evaluate their performance on hardware, we have implemented our Sboxes with first order TI protections in Verilog. For synthesis, we used Synopsys Design Compiler with the TSMC 180nm standard cell library. Their area requirements as well as clock cycles are presented in Table 6. Note that only the combinational part is documented in Table 6: as most previous works excluded the multiplexers and registers as "required extra logic", we cannot further compare the whole design[5]. For clarity, Table 6 only shows one 8 bit Sbox and one 4 bit Sbox: other alternatives can be found in the Appendix.

	Size	Diff.	Lin.	Deg.	Rounds	Protected		
						Area(GE)	Delay(ns)	Cycles
PRESENT [24]	4	4	8	3	n/a	151	—	6
GIFT [7]	4	6	8	3	n/a	172.5	—[6]	6
$S4$	4	4	8	3	2	54	0.72	28
AES [38]	8	4	32	7	n/a	2224	—	3
SB_1 [15]	8	16	64	6	8	51	1.09	8
SB_4 [15]	8	8	56	7	5	202	2.10	5
$S8$	8	8	64	6	3	181	1.89	78

Table 6. Hardware evaluation of various Sboxes

Since most results in Table 6 are uniform first order threshold implementations, we did not present their fresh randomness requirements. Only the AES Sbox uses 32 random bits; all others do not take fresh randomness. Thanks to its fine granularity, our protected Sboxes can be implemented with 1-bit combinational logic, which leads to very compact implementations (Table 6). However, this is nothing more than a trade-off: the number of cycles clearly shows the price to pay. Besides, for a larger n, shift-invariant constructions lose most of their charms. Table 6 shows the area gain for 8 bit Sbox is neglectable (if any, considering a serial implementation uses more MUX-es), compared with Boss et. al's construction. The reason for this roots in the philosophy of shift invariance: shift invariance saves area by reducing the outputs of a logic circuit, but not the inputs. Our 1-bit implementation is still a $2n$-variate Boolean function. Boss et. al's construction uses smaller Sboxes, which reduces the input scale of the protected circuit. Technically, for an arbitrary vectorial Boolean function, the implementation cost grows linearly with its output, but exponentially with its input. Having said that, the main advantage of our construction is providing flexible implementation trade-offs, on both software and hardware platforms. Although Boss et. al's paper also mentioned software-efficiency, their prediction

[5] Depending on the specific implementation, such "extra logic" can actually predominates the overall area cost. To this end, we would like the stress that our serial implementation is only worthwhile if it has a significant advantage in area cost.

[6] Not given.

is actually based on the number of AND-s. We believe that software performance evaluation should use actual assembly code: due to the limited resources available (eg. instructions, registers, buses, etc.), high-level estimations could be misleading.

6 Security Evaluation

6.1 Software: ARM M0

In order to evaluate our protected Sbox in practice, we have implemented such Sboxes on both software and hardware platforms. For software implementation, our target chip is an NXP LPC1114 (ARM Cortex M0) processor. The measurement point connects to a 100 Ohm resistor on the VCC end. Power traces were captured with a PicoScope 2206B running at a sampling rate of 125MSa/s. The clock speed of the target core was set to 8MHz. For leakage detection, we use the non-specific fix-vs-random T-test [39]. In order to increase the detection power, we force all parallel Sboxes to use the same input shares (i.e. all the concurrent Sbox computations are exactly the same). Figure 3 shows the evaluation results for our 4 bit Sbox with 1 million traces:

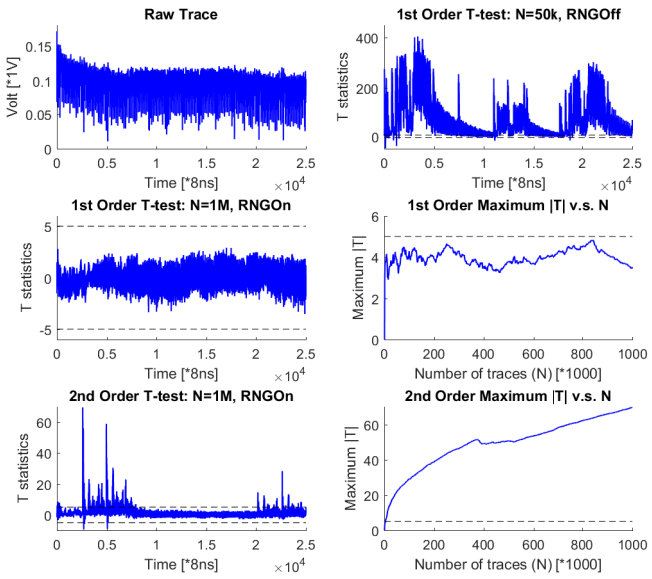


Fig. 3. Software evaluation of S4

Considering the Sbox computation includes 25000 time points, we increase the T-test threshold to 5 [40]. With 1 million traces, a first order T-test cannot find any significant leakage. As we have only implemented a first order TI protection, second order attacks are still feasible. In theory, the most efficient 2nd order attack should be multi-variate attacks which combine 2 independent samples on the trace. In practice, significant leakage can be detected by simply performing the same T-test on the second moment (Figure 3). Therefore, we did not enumerate all possible second order sample combinations on the trace. The 8-bit case is quite similar: due to the limited space, we present the results for S_8 in the Appendix.

6.2 Hardware: SAKURA-X FPGA

For hardware implementations, we have tested our Sboxes on the SAKURA-X board with Xilinx Kintex-7 FPGA. In order to increase the signal-to-noise ratio, an Agilent 25db amplifier is connected to the measured signal. Moreover, considering our all-serial implementation has very limited power consumption, we extended a $3n$-bit protected Sbox to a 384-bit design: for the 4 bit case, this means there are 32 parallel Sboxes implemented on the board. For 8 bit Sboxes, there are 16 parallel Sboxes. Similar to software implementations, all the implemented Sboxes were given the same input shares. Our FPGA design run at 3MHz, while our Lecroy Waverunner 700 Zi scope was capturing traces at 500MSa/s. Obvious outliers were removed before T-test. Figure 4 shows the leakage detection results for our 4 bit Sbox after 5 million traces. Clearly, our protected design is first order secure. Since our implementation is a serial one, technically, the second order detection should use multi-variate T-test. However, it is not hard to see that the second moment already shows some clear leakage. Like the software case, we present the 8-bit results in the Appendix.

7 Conclusion

In this paper, we propose a novel Sbox construction using quadratic shift-invariant transformations. Thanks to the shift-invariant property, our Sbox constructions have a fine "granularity"which contributes to more flexible implementation trade-offs. Both software and hardware implementations have been discussed and evaluated (on ARM processors and an FPGA). The strong point of our Sboxes is that their first order protection can be efficiently tuned for the needs in different applications without using any fresh randomness. Experiments suggest our TI protection has effectively eliminated the 1-st order leakage. Meanwhile, to the best of our knowledge, this is the first computation based TI Sbox implementation in software (rather than the table-based TI implementation in [5]). Considering masked software implementations do not always back their security claims (eg. [17]), utilizing threshold implementations on software is of independent interest.

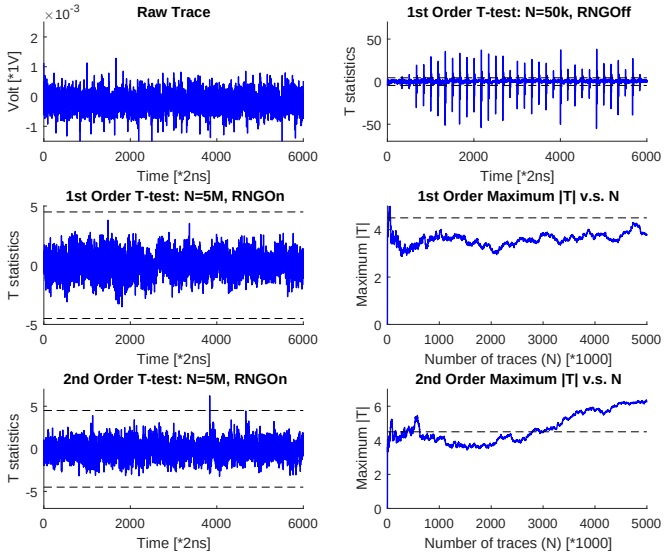


Fig. 4. Hardware evaluation of S4

Acknowledgements

We would like to thank all anonymous reviewers for improving the quality of this paper as well as providing insights from some other perspectives. This work has been funded in part by EPSRC under grant agreement EP/N011635/1 (LADA).

References

1. Nikova, S., Rechberger, C., Rijmen, V.: Threshold Implementations Against Side-Channel Attacks and Glitches. In: Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings. (2006) 529–545
2. Nikova, S., Rijmen, V., Schläffer, M.: Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. J. Cryptology **24**(2) (2011) 292–321
3. Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., Stütz, G.: Threshold Implementations of All 3×3 and 4×4 S-Boxes. In: Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings. (2012) 76–91
4. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. (2011) 69–88

5. Kutzner, S., Nguyen, P.H., Poschmann, A., Wang, H.: On 3-Share Threshold Implementations for 4-Bit S-boxes. In: Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers. (2013) 99–113
6. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: A More Efficient AES Threshold Implementation. In: Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings. (2014) 267–284
7. Gupta, N., Jati, A., Chattopadhyay, A., Sanadhya, S.K., Chang, D.: Threshold Implementations of GIFT: A Trade-off Analysis. IACR Cryptology ePrint Archive **2017** (2017) 1040
8. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. (1999) 398–412
9. Messerges, T.S.: Securing the AES Finalists Against Power Analysis Attacks. In: Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings. (2000) 150–164
10. De Meyer, L., Bilgin, B.: Classification of Balanced Quadratic Functions. IACR Cryptology ePrint Archive **2018** (2018) 113
11. Bozilov, D., Bilgin, B., Sahin, H.A.: A Note on 5-bit Quadratic Permutations' Classification. IACR Trans. Symmetric Cryptol. **2017**(1) (2017) 398–404
12. Beyne, T., Bilgin, B.: Uniform First-Order Threshold Implementations. In: Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers. (2016) 79–98
13. De Meyer, L., Moradi, A., Wegener, F.: Spin me right round rotational symmetry for fpga-specific aes. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3) (Aug. 2018) 596–626
14. Daemen, J.: Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. (2017) 137–153
15. Boss, E., Grosso, V., Güneysu, T., Leander, G., Moradi, A., Schneider, T.: Strong 8-bit Sboxes with efficient masking in hardware extended version. J. Cryptographic Engineering **7**(2) (2017) 149–165
16. Meyer, L.D., Varici, K.: More Constructions for strong 8-bit S-boxes with efficient masking in hardware. In: Proceedings of the 38th Symposium on Information Theory in the Benelux, Delft,NE, Werkgemeenschap voor Informatie- en Communicatietheorie (2017) 11
17. Sasdrich, P., Bock, R., Moradi, A.: Threshold Implementation in Software - Case Study of PRESENT. In: Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings. (2018) 227–244
18. Jungk, B., Petri, R., St ottinger, M.: Efficient Side-Channel Protections of ARX Ciphers. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3) (Aug. 2018) 627–653
19. Balasch, J., Gierlichs, B., Reparaz, O., Verbauwhede, I.: DPA, Bitslicing and Masking at 1 GHz. In: Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings. (2015) 599–619

20. de Groot, W., Papagiannopoulos, K., de la Piedra, A., Schneider, E., Batina, L.: Bitsliced Masking and ARM: Friends or Foes? In: Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers. (2016) 91–109
21. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.: On the Cost of Lazy Engineering for Masked Software Implementations. In: Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. (2014) 64–81
22. Daemen, J.: Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis. K.U.Leuven (1995) http://jda.noekeon.org/.
23. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. (2013) 313–314
24. Kutzner, S., Nguyen, P.H., Poschmann, A., Wang, H.: On 3-Share Threshold Implementations for 4-Bit S-boxes. In: Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers. (2013) 99–113
25. Gupta, N., Jati, A., Chattopadhyay, A., Sanadhya, S.K., Chang, D.: Threshold Implementations of GIFT: A Trade-off Analysis. IACR Cryptology ePrint Archive **2017** (2017) 1040
26. Nyberg, K.: Differentially uniform mappings for cryptography. In Helleseth, T., ed.: Advances in Cryptology — EUROCRYPT '93, Berlin, Heidelberg, Springer Berlin Heidelberg (1994) 55–64
27. Božilov, D., Bilgin, B., Sahin, H.: A Note on 5-bit Quadratic Permutations' Classification. IACR Transactions on Symmetric Cryptology **2017**(1) (2017) 398–404
28. Meyer, L.D., Bilgin, B.: Classification of Balanced Quadratic Functions. Cryptology ePrint Archive, Report 2018/113 (2018) https://eprint.iacr.org/2018/113.
29. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. (2007) 450–466
30. Mariot, L., Picek, S., Leporati, A., Jakobovic, D.: Cellular automata based s-boxes. Cryptography and Communications (May 2018)
31. Khovratovich, D., Nikolic, I.: Rotational Cryptanalysis of ARX. In: Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers. (2010) 333–346
32. Leander, G., Poschmann, A.: On the Classification of 4 Bit S-Boxes. In: Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings. (2007) 159–176
33. ARM: Arm and thumb-2 instruction set. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/QRC0006_UAL16.pdf
34. ARM: Thumb 16-bit instruction set. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf
35. Cnudde, T.D., Bilgin, B., Gierlichs, B., Nikov, V., Nikova, S., Rijmen, V.: Does Coupling Affect the Security of Masked Implementations? In: Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers. (2017) 1–18

36. Goudarzi, D., Rivain, M.: How Fast Can Higher-Order Masking Be in Software? In: Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I. (2017) 567–597
37. Wegener, F., Moradi, A.: A First-Order SCA Resistant AES Without Fresh Randomness. In: Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings. (2018) 245–262
38. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Trade-Offs for Threshold Implementations Illustrated on AES. IEEE Trans. on CAD of Integrated Circuits and Systems **34**(7) (2015) 1188–1200
39. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side channel resistance validation. Technical report, CRI (2011)
40. Ding, A.A., Zhang, L., Durvaux, F., Standaert, F., Fei, Y.: Towards Sound and Optimal Leakage Detection Procedure. In: Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers. (2017) 105–122

A Full table of $S8$

This Sbox is constructed with 3-layer SPN, where S is a quadratic shift-invariant permutation: The content of $S8$ is presented as follow:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	4c	98	db	31	0a	b7	83	62	56	14	2f	6f	2c	07	4b
1	c4	dd	ac	ba	28	46	5e	3f	de	bf	58	36	0e	18	96	8f
2	89	ca	bb	f7	59	6d	75	4e	50	6b	8c	b8	bc	f0	7e	3d
3	bd	ab	7f	66	b0	d1	6c	02	1c	72	30	51	2d	34	1f	09
4	13	82	95	0b	77	91	ef	06	b2	5b	da	3c	ea	74	9c	0d
5	a0	64	d6	1d	19	aa	71	cd	79	c5	e1	52	fc	37	7a	be
6	7b	e5	57	c6	fe	17	cc	2a	61	87	a3	4a	d8	49	04	9a
7	38	f3	e4	20	60	dc	a2	11	5a	e9	68	d4	3e	fa	12	d9
8	26	ed	05	c1	2b	97	16	a5	ee	5d	23	9f	df	1b	0c	c7
9	65	fb	b6	27	b5	5c	78	9e	d5	33	e8	01	39	a8	1a	84
A	41	85	c8	03	ad	1e	3a	86	32	8e	55	e6	e2	29	9b	5f
B	f2	63	8b	15	c3	25	a4	4d	f9	10	6e	88	f4	6a	7d	ec
C	f6	e0	cb	d2	ae	cf	8d	e3	fd	93	2e	4f	99	80	54	42
D	c2	81	0f	43	47	73	94	af	b1	8a	92	a6	08	44	35	76
E	70	69	e7	f1	c9	a7	40	21	c0	a1	b9	d7	45	53	22	3b
F	b4	f8	d3	90	d0	eb	a9	9d	7c	48	f5	ce	24	67	b3	ff

Table 7. The quadratic shift-invariant permutation S

, where the diffusion layer uses two layers of A:

$$\mathbf{A} = \begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 1&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}$$

The overall Sbox is

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	6d	f1	8f	3d	80	b4	31	50	82	3f	2e	51	0f	1c	c1
1	a0	c4	25	12	5d	67	a4	65	81	1e	e0	1d	38	e5	97	05
2	19	f3	da	03	ba	91	07	b5	9e	7f	c7	77	32	76	a3	e1
3	98	93	94	5c	7e	17	c2	0a	70	43	cb	a6	5e	ac	7c	a1
4	8b	a5	d6	2a	18	ed	c0	57	9a	6b	23	06	88	08	2b	cd
5	24	7b	d2	2c	e7	59	69	dc	9f	0e	61	75	20	89	fc	ff
6	0c	bd	27	9d	16	b9	86	fd	73	d7	b1	5a	f0	5f	14	40
7	74	e3	df	d5	f2	36	e6	64	2f	e9	92	e4	fa	71	be	b2
8	9c	ce	41	42	b6	63	87	a2	30	29	cc	ef	8c	68	c6	3c
9	4a	66	b0	c9	bc	dd	8e	45	21	90	d1	ae	1f	62	56	db
A	48	96	f6	ab	8d	a7	58	b7	22	f8	ec	28	0d	f7	bb	f5
B	2d	6a	4d	fe	eb	0b	01	13	52	ea	7a	10	f9	72	7d	8a
C	6c	6e	34	95	d0	c5	6f	49	ee	4b	b3	4c	af	3b	a8	4f
D	4e	39	c3	9b	a9	84	78	11	60	55	aa	85	15	02	fb	09
E	37	ca	79	47	3e	f4	d8	e2	53	d9	26	3a	99	e8	c8	33
F	de	54	5b	b8	1a	83	46	35	d3	ad	44	d4	bf	04	cf	1b

Table 8. The overall Sbox

B Security evaluation on $S8$

C Other 8-bit Sbox candidates

C.1 $S8_1$

3 rounds, differential uniformity = 8, linearity = 72:

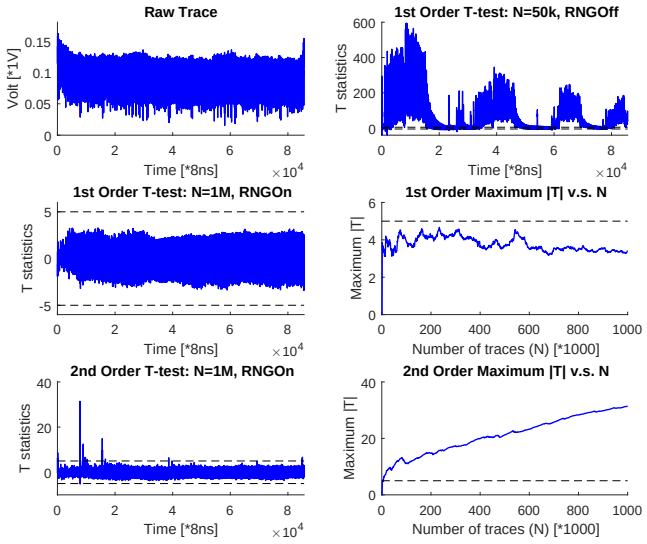


Fig. 5. Software evaluation of S8

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	a4	49	24	92	b1	48	a2	25	03	63	8c	90	31	45	2d
1	4a	77	06	f2	c6	7c	19	6a	21	9e	62	14	8a	b2	5a	ab
2	94	60	ee	d3	0c	7f	e5	5f	8d	fb	f8	47	32	c3	d4	ec
3	42	2f	3d	99	c4	2e	28	0b	15	fa	65	43	b4	dc	57	f6
4	29	6c	c0	4c	dd	1f	a7	ac	18	df	fe	f0	cb	8b	be	37
5	1b	c7	f7	e2	f1	aa	8e	1c	64	3a	87	10	a9	70	d9	c9
6	84	91	5e	82	7a	e8	33	68	89	1e	5c	02	50	40	16	cf
7	2a	a6	f5	b0	ca	c1	86	44	69	67	b9	7e	ae	27	ed	ad
8	52	12	d8	51	81	46	98	96	bb	79	3e	35	4f	0a	59	d5
9	30	e9	bf	af	fd	a3	e1	76	97	cc	17	85	7d	a1	6e	7b
a	36	26	8f	56	ef	78	c5	9b	e3	71	55	0e	1d	08	38	e4
b	c8	41	74	34	0f	01	20	e7	53	58	e0	22	b3	3f	93	d6
c	09	a8	23	4b	bc	9a	05	ea	f4	d7	d1	3b	66	c2	d0	bd
d	13	2b	3c	cd	b8	07	04	72	a0	1a	80	f3	2c	11	9f	6b
e	54	a5	4d	75	eb	9d	61	de	95	e6	83	39	0d	f9	88	b5
f	d2	ba	ce	6f	73	9c	fc	da	5d	b7	4e	6d	db	b6	5b	ff

Table 9. The quadratic shift-invariant permutation S

$$\mathbf{A} = \begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}$$

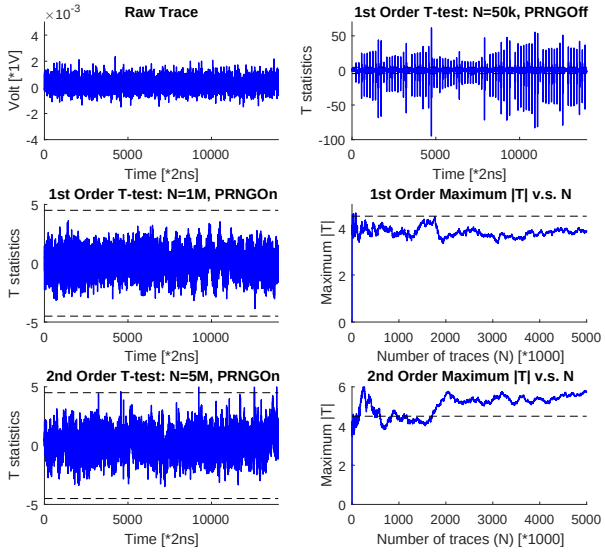


Fig. 6. Hardware evaluation of S8

C.2 $S8_2$

3 rounds, differential uniformity = 8, linearity = 72:

$$\mathbf{A} = \begin{bmatrix} 0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0 \\ 1\,0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,0 \\ 1\,0\,0\,0\,0\,0\,0\,1 \\ 1\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}$$

C.3 $S8_3$

3 rounds, differential uniformity = 8, linearity = 72:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	b5	6b	17	d6	e4	2e	d5	ad	9a	c9	37	5c	ec	ab	d2
1	5b	77	35	d0	93	38	6e	0c	b8	16	d9	be	57	7e	a5	45
2	b6	53	ee	c2	6a	08	a1	0a	27	40	70	de	dc	3c	18	31
3	71	0d	2c	99	b3	48	7d	4f	ae	50	fc	cb	4b	32	8a	3a
4	6d	39	a6	3b	dd	0e	85	9f	d4	02	10	0f	43	12	14	8c
5	4e	83	80	84	e0	aa	bd	3e	b9	f6	78	fe	30	f8	62	63
6	e2	e6	1a	d7	58	db	33	79	67	e1	90	df	fa	fb	9e	56
7	5d	c0	a0	f4	f9	e3	97	44	96	89	64	b2	15	8d	74	25
8	da	8b	72	ea	4d	9b	76	69	bb	68	1c	06	0b	5f	3f	a2
9	a9	61	04	05	20	6f	1e	98	86	cc	24	a7	28	e5	19	1d
a	9c	9d	07	cf	01	87	09	46	c1	42	55	1f	7b	7f	7c	b1
b	73	eb	ed	bc	f0	ef	fd	2b	60	7a	f1	22	c4	59	c6	92
c	c5	75	cd	b4	34	03	af	51	b0	82	b7	4c	66	d3	f2	8e
d	ce	e7	c3	23	21	8f	bf	d8	f5	5e	f7	95	3d	11	ac	49
e	ba	5a	81	a8	41	26	e9	47	f3	91	c7	6c	2f	ca	88	a4
f	2d	54	13	a3	c8	36	65	52	2a	d1	1b	29	e8	94	4a	ff

Table 10. The quadratic shift-invariant permutation S

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	4a	94	8e	29	4e	1d	2a	52	56	9c	c8	3a	13	54	2d
1	a4	77	ac	2f	39	c7	91	3f	74	e9	26	eb	a8	18	5a	ba
2	49	ca	ee	3d	59	f7	5e	a0	72	bf	8f	12	23	c3	7e	ce
3	e8	f2	d3	99	4c	7b	d7	b0	51	05	30	34	b4	cd	75	5c
4	92	93	95	c4	dd	f1	7a	06	b2	fd	ef	f0	bc	de	41	73
5	e4	7c	7f	b7	1f	aa	24	c1	46	90	87	01	fc	07	9d	36
6	d1	19	e5	7d	a7	42	33	86	98	1e	f6	20	af	04	61	9a
7	a2	f3	0a	0b	60	1c	68	44	69	76	9b	d4	ea	d8	b8	da
8	25	47	27	15	2b	64	89	96	bb	97	e3	9f	f4	f5	0c	5d
9	65	9e	fb	50	df	09	e1	67	79	cc	bd	58	82	1a	e6	2e
a	c9	62	f8	03	fe	78	6f	b9	3e	db	55	e0	48	80	83	1b
b	8c	be	21	43	0f	10	02	4d	f9	85	0e	22	3b	6a	6c	6d
c	a3	8a	32	4b	cb	cf	fa	ae	4f	28	84	b3	66	2c	0d	17
d	31	81	3c	dc	ed	70	40	8d	5f	a1	08	a6	c2	11	35	b6
e	45	a5	e7	57	14	d9	16	8b	c0	6e	38	c6	d0	53	88	5b
f	d2	ab	ec	c5	37	63	a9	ad	d5	e2	b1	d6	71	6b	b5	ff

Table 11. The quadratic shift-invariant permutation S

$$\mathbf{A} = \begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}$$

C.4 $S8_4$

3 rounds, differential uniformity = 8, linearity = 72:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	5b	b6	42	6d	1b	84	5d	da	30	36	73	09	ce	ba	d2
1	b5	77	60	0d	6c	83	e6	a6	12	61	9d	41	75	2b	a5	54
2	6b	06	ee	2c	c0	80	1a	f5	d8	04	07	74	cd	3c	4d	13
3	24	d0	c2	99	3b	e2	82	f4	ea	af	56	bc	4b	23	a8	6f
4	d6	c6	0c	b3	dd	e0	58	ca	81	20	01	0f	34	b8	eb	c8
5	b1	38	08	2e	0e	aa	e8	e3	9b	a3	78	ef	9a	8f	26	9c
6	48	6e	a1	28	85	8e	33	97	76	e1	c5	fd	05	bf	e9	fc
7	d5	6a	5f	4f	ac	3e	79	44	96	98	46	e7	51	72	de	52
8	ad	21	8d	ae	18	b9	67	69	bb	86	c1	53	b0	a0	95	2a
9	03	16	40	fa	02	3a	1e	89	68	cc	71	7a	d7	5e	91	b7
a	63	d9	70	65	10	87	5c	64	1c	17	55	f1	d1	f7	c7	4e
b	37	14	47	cb	f0	fe	df	7e	35	a7	1f	22	4c	f3	39	29
c	90	57	dc	b4	43	a9	50	15	0b	7d	1d	c4	66	3d	2f	db
d	ec	b2	c3	32	8b	f8	fb	27	0a	e5	7f	3f	d3	11	f9	94
e	ab	5a	d4	8a	be	62	9e	ed	59	19	7c	93	f2	9f	88	4a
f	2d	45	31	f6	8c	c9	cf	25	a2	7b	e4	92	bd	49	a4	ff

Table 12. The quadratic shift-invariant permutation S

$$\mathbf{A} = \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}$$

C.5 $S8_5$

3 rounds, differential uniformity = 8, linearity = 64:

$$\mathbf{A} = \begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \end{bmatrix}$$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	9b	37	e7	6e	1b	cf	f1	dc	1d	36	bc	9f	b0	e3	87
1	b9	88	3a	40	6c	b3	79	ed	3f	54	61	41	c7	42	0f	c1
2	73	a3	11	8a	74	4a	80	f5	d8	52	67	a6	f2	96	db	f4
3	7e	04	a8	99	c2	56	82	5d	8f	af	84	ef	1e	d0	83	06
4	e6	c6	47	2c	22	ec	15	90	e8	92	94	a5	01	95	eb	34
5	b1	3b	a4	65	ce	aa	4d	62	e5	35	2d	b6	b7	89	e9	9c
6	fc	97	08	28	51	d4	33	fd	85	b4	ac	d6	05	da	ba	2e
7	1f	de	5f	d5	09	26	df	bb	3c	a7	a1	71	07	72	0c	32
8	cd	f3	8d	f8	8e	5e	58	c3	44	20	d9	f6	2a	a0	21	e0
9	d1	45	25	fa	29	53	4b	7a	02	cc	2b	ae	d7	f7	68	03
a	63	16	76	48	49	d2	ca	1a	9d	b2	55	31	9a	5b	c4	4e
b	cb	14	6a	fe	5a	6b	6d	17	6f	ea	13	dd	d3	b8	39	19
c	f9	7c	2f	e1	10	7b	50	70	a2	7d	a9	3d	66	57	fb	81
d	0b	24	69	0d	59	98	ad	27	0a	7f	b5	8b	75	ee	5c	8c
e	3e	f0	bd	38	be	9e	ab	c0	12	86	4c	93	bf	c5	77	46
f	78	1c	4f	60	43	c9	e2	23	0e	30	e4	91	18	c8	64	ff

Table 13. The quadratic shift-invariant permutation S