Threshold Implementation in Software
Case Study of PRESENT

Pascal Sasdrich, René Bock, and Amir Moradi

Horst Görtz Institute for IT-Security, Ruhr-Universitäat Bochum, Bochum, Germany
{firstname.lastname}@rub.de

Abstract. Masking is one of the predominantly deployed countermeasures in order to prevent side-channel analysis (SCA) attacks. Over the years, various masking schemes have been proposed. However, the implementation of Boolean masking schemes has proven to be difficult in particular for embedded devices due to undisclosed architecture details and device internals. In this article, we investigate the application of Threshold Implementation (TI) in terms of Boolean masking in software using the PRESENT cipher as a case study. Since TI has proven to be a proper solution in order to implement Boolean masking for hardware circuits, we apply the same concept for software implementations and compare it to classical first- and second-order Boolean masking schemes. Eventually, our practical security evaluations reveal that amongst all our considered implementation variants only the TI can provide first-order security while all others still exhibit detectable first-order leakage.

Key words: Side-Channel Analysis, Boolean masking, Threshold Implementation, t-test, micro-controller, AVR, PRESENT

1 Introduction

Among the protection schemes against side-channel analysis (SCA) attacks, it can be dared to say that *masking* is the best studied countermeasure. Many different kinds of masking schemes for both software and hardware platforms have been introduced [1, 5, 10, 13, 15, 20, 25, 29, 36, 38]. Each of them comes with its own advantages (e.g., simplicity and scalability to high protection orders) and disadvantages (e.g., high area and time overheads) and some with shortcomings (see for example [19, 27]). Our focus in this work is the realization of Boolean masking scheme in software implementations.

It is already known that – due to the internal architecture of micro-processors – masked implementations may still exhibit undesired exploitable leakage (see [3] as an example). It indeed becomes more problematic when details of the internal architecture of the underlying commercial micro-processor are kept secret. For instance, the way the pipeline is built, the shared bus between ALU and memory together with the fashion in that the masked program code is written, can impact the leakage of the resulting implementation. As a simple example, suppose that two Boolean shares $(\boldsymbol{x}^1, \boldsymbol{x}^2)$ of a secret value $\boldsymbol{x}$ are consecutively transferred through a bus, that leads to leakage depending on distance between the shares,

i.e., $\boldsymbol{x}^1 \oplus \boldsymbol{x}^2 = \boldsymbol{x}$. The attack reported in [27] follows the same principle. In this case, the implementation would exhibit first-order leakage while it is not possible to detect such a flaw by analyzing the program code without considering the details of the internal architecture.

On the other hand, Threshold Implementation has been introduced as a proper way to realize Boolean masking in hardware platforms [30]. It provides a suitable guideline on how to avoid heuristics in masked hardware (see [8,31]) that can provide provable first-order security. In short, in this paper we examine the efficiency of applying such a scheme on a software implementation. As the case study, we focus on the PRESENT cipher [7] and an Atmel AVR microcontroller. We give details of different ways to realize a masked implementation including first- and second-order classical Boolean masking and the Threshold Implementation variant. Our investigations are based on the performance figures (code size and latency) as well as security analysis. More precisely, we present the result of leakage detection over practical SCA measurements.

Outline. In Section 2 we deal with the essential concepts to follow the rest of the paper including Boolean masking, Threshold Implementation, and possible ways to apply Threshold Implementation on PRESENT S-box. Section 3 gives the details of different variants of the masked PRESENT implementations, and in Section 4 the corresponding practical SCA analyses are presented. Finally, we conclude our research in Section 5.

2 Concept

2.1 Notation

We denote single-bit random variables using lower-case characters while we indicate multi-bit vectors by bold ones. Further, we use subscripts for elements within a vector, bars for shared representations of random variables and superscripts for elements of a shared representation. Functions are indicated using sans serif fonts and sets are denoted by calligraphic ones.

Moreover, let us denote any vector $\boldsymbol{x} \in \mathbb{GF}(2^m)$ of m single-bit elements by $\langle x_1, \ldots, x_m \rangle$. Then, the shared representation $\bar{\boldsymbol{x}}$ of a vector $\boldsymbol{x}$ under Boolean masking with s shares is given as $\bar{\boldsymbol{x}} = (\boldsymbol{x}^1, \ldots, \boldsymbol{x}^s)$, where:

$$\boldsymbol{x} = \bigoplus_{i=1}^{s} \bar{\boldsymbol{x}} = \bigoplus_{i=1}^{s} \boldsymbol{x}^i = \bigoplus_{i=1}^{s} \langle x_1^i, \ldots, x_m^i \rangle.$$

2.2 Boolean Masking

During the last two decades, Boolean masking has become the common approach to prevent information leakage of digital devices through physical side channels such as the instantaneous power consumption or electromagnetic radiations. Since sensitive information can be extracted from those physical observations by means of statistical analysis based on statistical moments of different orders, Boolean

masking uses the concept of *secret sharing* to split a sensitive variable $\boldsymbol{x}$ into s shares $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^s$ such that $\boldsymbol{x} = \boldsymbol{x}^1 \oplus \ldots \oplus \boldsymbol{x}^s$.

In general, Boolean masking can provide protection up to the d-th order using $s = d + 1$ shares that have to be processed independently. We should note that several physical effects, such as glitches or parasitic capacitances, can affect the security and lever the protection mechanism. Nevertheless, while linear operations can be applied independently to each share (due to the transparency of XOR over Boolean masking), all challenges of realizing a Boolean masked implementation are due to the non-linear functions (S-boxes) involved in any cryptographic primitive. To this end, masking in software is realized following two different approaches:

- The S-box is represented by a sequence of operations including a unique (or a limited number of) non-linear function, e.g., a 2-bit AND gate. Then, based on the underlying protection order d, the masked (secure) version of such a unique non-linear function is developed as a gadget. As the final step, the operations of the S-box are replaced by their secure version. This needs fresh randomness every time the secure non-linear function (the gadget) is called, and due to the sequential nature of the algorithm its timing overhead is not negligible compared to a naive unprotected implementation. The interested reader is referred to [17,18,38] for a couple of examples.
- Alternatively, the S-box is realized using a randomized look-up table S' in terms of

$$\mathsf{S}'(\boldsymbol{x} \oplus \boldsymbol{m}^1 \oplus \ldots \oplus \boldsymbol{m}^{s-1}) = \mathsf{S}(\boldsymbol{x}) \oplus \boldsymbol{n}^1 \oplus \ldots \oplus \boldsymbol{n}^{s-1}, \tag{1}$$

  with $\boldsymbol{m}^1, \ldots, \boldsymbol{m}^{s-1}$ considered as input masks and $\boldsymbol{n}^1, \ldots, \boldsymbol{n}^{s-1}$ as output masks. Depending on the S-box size and the number of shares s, it is usually impossible to precompute and store the masked S-box S' for all possible masks (known as Global Look-Up-Table [35]). Therefore, S' is frequently recomputed to avoid large memory requirements. Examples include but are not restricted to [37, 39], and [42], where such a construction for AES at arbitrary order is presented while its flaw has been reported in [11].[1]

In this work, our focus is on the second approach, i.e., the pre-computed and randomized look-up table S', to which we refer as **classical Boolean masking**. In case Equation (1) is implemented as single look-up table, the input and output masks have to fulfill certain criteria in order to realize a secure Boolean masking scheme. In particular, input and output masks cannot be the same. Otherwise, if the masked S-box input $\boldsymbol{x} \oplus \boldsymbol{m}^1 \oplus \ldots \oplus \boldsymbol{m}^{s-1}$ is overwritten by the masked S-box output $\mathsf{S}(\boldsymbol{x}) \oplus \boldsymbol{m}^1 \oplus \ldots \oplus \boldsymbol{m}^{s-1}$, the leakage depends on unmasked value $\boldsymbol{x} \oplus \mathsf{S}(\boldsymbol{x})$ [27] (see [4] and [26, chap. 9] as examples where such a flaw exists). Hence, in a conservative manner the output masks have to be independent of the input masks. However, since this might be impracticable particularly

[1] Alternatively, there exist other solutions [9, 14, 15] which make use of the S-box construction, e.g., GF(2^8) inversion of AES S-box.

for higher orders, more practical approaches may use a function to derive the output masks from the input masks but have to ensure the uniformity. More precisely, if $\boldsymbol{n}^{i\in\{1,\ldots,s-1\}} = \mathsf{f}(\boldsymbol{m}^i)$, it must be ensured that $\boldsymbol{n}^i \oplus \boldsymbol{m}^i$ is uniform over $\mathbb{GF}(2^m)$. Otherwise, the above expressed distance (between the S-box input and output) would not be uniformly masked. We should also refer to low-entropy masking schemes [5,29] which are designed to enable keeping all masked tables in memory, i.e., no recomputation and mask update is required, but at the cost of limited protection [19,24,43]. For example, the Rotating S-box Masking (RSM) construction introduced in [29] (and used in DPA contest V4.1) makes use of a reduced 8-bit mask space of 2^4 elements $\{m_0,\ldots,m_{15}\}$. This allowed the authors to precompute all masked S-boxes as $\mathsf{S}'_i(\boldsymbol{x} \oplus \boldsymbol{m}_i) = \mathsf{S}(\boldsymbol{x}) \oplus \boldsymbol{m}_{i+1}$. In means that the output mask is derived from the input mask as $\mathsf{f}(\boldsymbol{m}_i) = \boldsymbol{m}_{i+1}$. As shown in [27], the distance between the input mask and the output mask $\boldsymbol{m}_i \oplus \boldsymbol{m}_{i+1}$ is not uniform, hence first-order leakage considering the distance between the S-box input and output $\boldsymbol{x} \oplus \mathsf{S}(\boldsymbol{x})$ is detectable.

2.3 Threshold Implementation

As a special case of Boolean masking using multi-party computation, Threshold Implementation (TI) has been proposed by Nikova *et al.* [30] as a provable secure first-order masking scheme for digital circuits even in the presence of glitches. In this work, we make use of its basic concept in software, which is defined by the following properties.

Correctness. In order to evaluate any function $\mathsf{F}(\boldsymbol{x}) = \boldsymbol{y}$ on the shared representation $\bar{\boldsymbol{x}} = (\boldsymbol{x}^1,\ldots,\boldsymbol{x}^s)$ with s shares, we can use corresponding component functions $\mathsf{f}^{i\in\{1,\ldots,n\}}(\bar{\boldsymbol{x}}) = \boldsymbol{y}^i$ in order to evaluate F for each output share individually but have to ensure *correctness*, i.e., the result $\bar{\boldsymbol{y}} = (\boldsymbol{y}^1,\ldots,\boldsymbol{y}^n)$ has to be the shared representation of $\boldsymbol{y}$ with $n \geq s$.

Non-Completeness. Security in the sense of the first-order statistical moment is achieved using *non-complete* component functions $\mathsf{f}^{i\in\{1,\ldots,n\}}$, i.e., each resulting share $(\boldsymbol{y}^1,\ldots,\boldsymbol{y}^n)$ should be independent of at least one input share.

Uniformity. The security of Boolean masking schemes is based on the *uniform* distribution of the masks. Supposing that the input of a TI function is uniformly shared, its output should also be a uniform sharing since it will be used as an input to another shared function (e.g., next cipher rounds). This means, given all possible input sharings $\mathcal{X} = \{\bar{\boldsymbol{x}} | \bigoplus_{i=1}^{s} \bar{\boldsymbol{x}} = \boldsymbol{x}\}$, the set of all possible output sharings $\{\mathsf{f}^1,\ldots,\mathsf{f}^n | \bar{\boldsymbol{x}} = \mathcal{X}\}$ should be *uniformly* drawn from $\mathcal{Y} = \{\bar{\boldsymbol{y}} | \bigoplus_{i=1}^{n} \bar{\boldsymbol{y}} = \boldsymbol{y}\}$ as all possible sharings of $\boldsymbol{y} = \mathsf{F}(\boldsymbol{x})$. Otherwise, the output would be shared with masks drawn from a biased source, and the first-order security cannot be guaranteed.

2.4 Application to PRESENT Cipher

PRESENT has been designed as Substitution-Permutation Network (SPN) with 31 rounds, a 64-bit block size and either an 80-bit or 128-bit key size. Each round

Table 1: Non-linear function $\mathsf{N}(\boldsymbol{m}) = \boldsymbol{n}$.

$\boldsymbol{m}$	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\boldsymbol{n}$	E	4	F	9	0	3	D	5	7	8	A	2	B	1	6	C
$\boldsymbol{m} \oplus \boldsymbol{n}$	E	5	D	A	4	6	B	2	F	1	0	9	7	C	8	3

consists of a key addition, succeeded by a confusion phase which consists of the same 4-bit S-box that is applied to all 4-bit words of the state in parallel before the bit permutation layer[2] provides diffusion. In particular, the S-box is a non-linear, cubic, 4-bit function with truth table S : C56B90AD3EF84712. All round keys are derived from the initial key using bit-wise rotations, addition of round constants and the application of the S-box. Eventually, a final post-whitening key addition is performed after the last round.

Boolean Masking. Classical first-order Boolean masking uses 2 shares $\boldsymbol{x}^1, \boldsymbol{x}^2$ with $\boldsymbol{x}^1 = \boldsymbol{x} \oplus \boldsymbol{m}$ and $\boldsymbol{x}^2 = \boldsymbol{m}$. Due to its small size (4-bit to 4-bit), the entire masked S-box as an 8-bit to 4-bit look-up table $\mathsf{S}'(\boldsymbol{x} \oplus \boldsymbol{m}, \boldsymbol{m}) = \mathsf{S}(\boldsymbol{x}) \oplus \boldsymbol{n}$ can fit into even a restricted memory. Hence, the recomputation of the masked S-box when $\boldsymbol{m}$ changes is not required. In this case we need to derive the output mask $\boldsymbol{n}$ from the input mask $\boldsymbol{m}$ in such a way that the uniformity of $\boldsymbol{m} \oplus \boldsymbol{n}$ always holds. An example of such a function, so-called $\boldsymbol{n} = \mathsf{N}(\boldsymbol{m})$ is given in Table 1. Note that we have derived this table by a search through random bijections $\boldsymbol{m} \to \mathsf{N}(\boldsymbol{m})$.

Threshold Implementation. In several articles, the TI concept has been applied on the PRESENT S-box leading to first- and second-order uniform TI constructions. Under the TI definitions, the minimum number of required shares s depends not only on the desired level of security (order d) but also on the algebraic degree t of the underlying S-box, i.e., $s > t\,d$. Since the PRESENT S-box is a cubic bijection ($t = 3$), for first-order security ($d = 1$) at least $s > 3$ shares are required. Therefore, all the reported TI PRESENT designs have followed a decomposition fashion by representing the S-box by two quadratic bijections as $\mathsf{S} = \mathsf{F} \circ \mathsf{G}$. This allows to reduce the number of shares to 3 with the cost of adding a register between the shared functions F and G for hardware implementations.

In the first relevant article [33], the authors have followed a non-systematic way and provided F and G whose *direct sharing*[3] automatically satisfy the uniformity, i.e., a first-order secure PRESENT S-box. In other works [28, 40], the authors followed the principle explained in [6] and decomposed the S-box into forms like

$$\mathsf{S} = \mathsf{A}'' \circ \mathsf{Q}_2 \circ \mathsf{A}' \circ \mathsf{Q}_1 \circ \mathsf{A}, \tag{2}$$

[2] A detailed description and discussion of the permutation layer can be found in the original article [7].

[3] See [30] for the definition and examples for direct sharing.

with A, A', and A'' being affine transformations, and Q_1 and Q_2 the identifiers of quadratic classes whose uniform sharing can be easily achieved by direct sharing. Since application of affine transformations does not change the uniformity, such a construction inherently fulfills the uniformity property.

However, since not all 4-bit S-boxes can be decomposed following the concept of Equation (2), Kutzner et al. proposed the notion of *factorization* in order to enable 3-share decomposition for all possible 4-bit permutations [21, 22, 23]. Fortunately, the `PRESENT` S-box belongs to those permutations that natively allow a decomposition into quadratic terms which enables more efficient designs.

According to [6] the `PRESENT` S-box belongs to the class $\mathcal{C}_{266}$ which can be decomposed by quadratic classes[4] $(\mathcal{Q}_{12}, \mathcal{Q}_{12})$, $(\mathcal{Q}_{294}, \mathcal{Q}_{299})$, $(\mathcal{Q}_{299}, \mathcal{Q}_{294})$, and $(\mathcal{Q}_{299}, \mathcal{Q}_{299})$ as identifier for (Q_1, Q_2) in Equation (2). As an example, the $(\mathcal{Q}_{299}, \mathcal{Q}_{294})$ case has been used in [28] and $(\mathcal{Q}_{12}, \mathcal{Q}_{12})$ in [40].

We selected $(\mathcal{Q}_{12}, \mathcal{Q}_{12})$ with $\mathcal{Q}_{12}$: `0123456789CDEFAB`, A : `01AB892345EFCD67`, A' : `0B835ED61A924FC7`, and A'' : `C98D6327AFEB0541`. However, since our goal is to realize such functions (including the component functions of the shared $\mathcal{Q}_{12}$) by means of look-up tables on software, we represent the S-box as

$$\mathsf{S} = \underbrace{\mathsf{A}'' \circ \mathcal{Q}_{12} \circ \mathsf{A}}_{\mathsf{F}} \circ \underbrace{\mathsf{A}^{-1} \circ \mathsf{A}' \circ \mathsf{A}''^{-1}}_{\mathsf{A}'''} \circ \underbrace{\mathsf{A}'' \circ \mathcal{Q}_{12} \circ \mathsf{A}}_{\mathsf{F}}. \tag{3}$$

Hence, it lets us reduce the required look-up tables to F : `C905AF8D63EB4127` and A''' : `8FDACB9E43160752`.

Applying direct sharing on $\mathcal{Q}_{12}$ would lead to a unique component function $\mathsf{f}_{\mathcal{Q}_{12}}(\langle a^1, b^1, c^1, d^1\rangle, \langle a^2, b^2, c^2, d^2\rangle) = \langle e, f, g, h\rangle$ as

$$\begin{aligned} e =& a^1, & f =& b^1 + b^2d^2 + c^2d^2 + d^2b^1 + d^2c^1 + b^2d^1 + c^2d^1, \\ g =& c^1 + b^2d^2 + d^2b^1 + b^2d^1, & h =& d^1, \end{aligned} \tag{4}$$

with $\langle a^1, b^1, c^1, d^1\rangle$ the 4-bit input share $\boldsymbol{x}^1$ (respectively for input share $\boldsymbol{x}^2$), $\langle e, f, g, h\rangle$ the 4-bit output share, and a and e the least significant bits. Hence, the three 4-bit output shares $\bar{\boldsymbol{y}} = (\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3)$ provided by $\boldsymbol{y}^1 = \mathsf{f}_{\mathcal{Q}_{12}}(\boldsymbol{x}^2, \boldsymbol{x}^3)$, $\boldsymbol{y}^2 = \mathsf{f}_{\mathcal{Q}_{12}}(\boldsymbol{x}^3, \boldsymbol{x}^1)$ and $\boldsymbol{y}^3 = \mathsf{f}_{\mathcal{Q}_{12}}(\boldsymbol{x}^1, \boldsymbol{x}^2)$ make a uniform first-order TI of $\mathcal{Q}_{12}$.

In a software implementation, we can make a look-up table

$$\mathsf{T}(\boldsymbol{x}^i, \boldsymbol{x}^j) = \mathsf{A}''\left(\mathsf{f}_{\mathcal{Q}_{12}}\left(\mathsf{A}\left(\boldsymbol{x}^i\right), \mathsf{A}\left(\boldsymbol{x}^j\right)\right)\right), \tag{5}$$

which is a component function of the shared function F in Equation (3). Therefore, in addition to a 4-bit to 4-bit look-up table $\mathsf{A}'''(.)$ it is sufficient to implement $\mathsf{T}(.,.)$ as an 8-bit to 4-bit look-up table to fully realize the TI S-box by 6 times look-ups through $\mathsf{T}(.,.)$ and 3 times look-ups through $\mathsf{A}'''(.)$ (see Equation (3)). As a reference to our construction, we below list the truth table of $\mathsf{T}(\boldsymbol{a}, \boldsymbol{b})$. Interestingly, the result is independent of the LSB of input $\boldsymbol{b}$ (see also Equation (4) which is independent of $\boldsymbol{a}^2$), hence we only have to store half of the table and can reduce memory requirements.

[4] Excluding the quadratic class $\mathcal{Q}_{300}$ whose uniform sharing needs two stages.

		b															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
a	**0**	c	c	2	2	c	c	c	c	c	c	6	6	8	8	c	c
	1	9	9	7	7	9	9	9	9	9	9	3	3	d	d	9	9
	2	e	e	0	0	0	0	0	0	a	a	0	0	0	0	4	4
	3	b	b	5	5	5	5	5	5	f	f	5	5	5	5	1	1
	4	a	a	a	a	a	a	4	4	e	e	a	a	a	a	0	0
	5	f	f	f	f	f	f	1	1	b	b	f	f	f	f	5	5
	6	8	8	8	8	6	6	8	8	8	8	c	c	2	2	8	8
	7	d	d	d	d	3	3	d	d	d	d	9	9	7	7	d	d
	8	6	6	c	c	2	2	6	6	6	6	8	8	6	6	6	6
	9	3	3	9	9	7	7	3	3	3	3	d	d	3	3	3	3
	a	4	4	e	e	e	e	a	a	0	0	e	e	e	e	e	e
	b	1	1	b	b	b	b	f	f	5	5	b	b	b	b	b	b
	c	0	0	4	4	4	4	e	e	4	4	4	4	4	4	a	a
	d	5	5	1	1	1	1	b	b	1	1	1	1	1	1	f	f
	e	2	2	6	6	8	8	2	2	2	2	2	2	c	c	2	2
	f	7	7	3	3	d	d	7	7	7	7	7	7	9	9	7	7

Table 2: Truth Table for $\mathsf{T}(\boldsymbol{a}, \boldsymbol{b})$

Higher-Order Boolean Masking. The above explained TI construction is a 2nd-order Boolean masking. Therefore, ignoring the non-completeness property of TI (which indeed has been defined considering hardware platforms), we can realize larger look-up tables hence reducing the latency. To this end we follow two procedures:

- As a classical 2nd-order Boolean masking we can implement a 12-bit to 12-bit look-up table which realizes the entire masked S-box. More precisely, we can build a look-up table $\mathsf{T}(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3) = (\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3)$ with $\boldsymbol{y}^1 \oplus \boldsymbol{y}^2 \oplus \boldsymbol{y}^3 = \mathsf{S}(\boldsymbol{x}^1 \oplus \boldsymbol{x}^2 \oplus \boldsymbol{x}^3)$. In order to ensure the uniformity, we can build such a look-up table in such a way that it realizes the above-explained TI S-box. In the following sections, this approach is referred to as "classical 2nd-order Boolean masking".
- As an alternative, we can build a 12-bit to 12-bit look-up table $\mathsf{T}(.,.,.)$ that implements the shared function F (see Equation (3)). Hence, by looking-up through such a table $\mathsf{T}(.,.,.)$ twice and thrice through the 4-bit to 4-bit look-up table A''', the output of the masked S-box can be computed which also guarantees the uniformity. We refer to this scheme as "classical 2nd-order Boolean masking with affine transformation".

In addition to the two above-expressed approaches, we consider two other implementation variants including *i*) classical 1st-order Boolean masking and *ii*) Threshold Implementation in our practical experiments presented in the next sections.

3 Implementation

In this section we introduce the target platform and describe and compare the performance figures of our implementations.

3.1 Target Platform

As the target platform, we have chosen an Atmel ATmega163 which is an 8-bit micro-controller with 16 KB programmable flash memory and 1024 B internal SRAM. It is constructed of two internal pipeline stages, provides 32 general purpose 8-bit registers, and uses an 8-bit RISC instruction set that can be programmed either using C compiler or AVR Assembler. In our experiments, we opted the micro-controller to operate at a frequency of 4 MHz. This choice has been made to obtain accurate side-channel measurements.

3.2 Pseudo-Code

Below we provide further implementation details on the realization of our considered implementation variants of Section 2. In particular, we want to stress that all implementations have been realized using AVR Assembler in order to maintain maximum control over the executed code and to prevent problems due to adverse compiler optimizations [3].

In general, all implementations consist of a key schedule routine and a round function that is sub-divided into key addition, substitution, and permutation layer. Since we opted to implement a key schedule without shared keys, this routine is the same for all implementation variants. Moreover, the `AddRoundKey` and `pLayer` are shared among the different variants as well and only the `sLayer` routine differs depending on the underlying masking scheme.

In the following, we provide pseudo-codes for all of our implementations and highlight important aspects and optimizations that have been applied.

Classical 1st-Order Boolean Masking. Algorithm 3.1 outlines our implementation of the classical Boolean masking scheme presented in Section 2.4 using a masked S-box look-up table S' and a non-linear mask update function N chosen in accordance with our presented concept. During the design and implementation process, we particularly took care of the processing of intermediate values in order to avoid side-channel leakage due to the distance between two successively processed values.

In general, if a masked value $\boldsymbol{x}^1 = \boldsymbol{x} \oplus \boldsymbol{m}$ and its mask $\boldsymbol{x}^2 = \boldsymbol{m}$ are processed consecutively, internal registers may be overwritten and leak through the distance of these values, i. e., $\boldsymbol{x}^1 \oplus \boldsymbol{x}^2 = \boldsymbol{x}$. In particular for load and store instructions of the ATmega163 an internal shadow register is involved in order to buffer the processed data which then creates a remnant of previous memory accesses [32]. Since this shadow register is not directly accessible, it can only be cleared by reading or writing a dummy value (e.g., 0). More precisely, every read and write operation has to be preceded by such a clear instruction to prevent leakage due to the distance between the consecutively accessed data. However, this holds not

Algorithm 3.1: Classical 1st-Order Boolean Masking

Input : $\bar{\boldsymbol{x}} = (\boldsymbol{x} \oplus \boldsymbol{m}, \boldsymbol{m})$: *shared plaintext*
$\boldsymbol{k}$: *cipher key*
Output : $\bar{\boldsymbol{y}} = (\boldsymbol{y}^1, \boldsymbol{y}^2)$: shared ciphertext

begin
  $\boldsymbol{rk} \leftarrow KeySchedule(\boldsymbol{k})$
  for $i \leftarrow 1$ **to** 31 **do**
    $\boldsymbol{x}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[i]$
    $\bar{\boldsymbol{x}} \leftarrow (\mathsf{S}'(\boldsymbol{x}^1, \boldsymbol{x}^2), \mathsf{N}(\boldsymbol{x}^2))$
    $\boldsymbol{x}^1 \leftarrow \mathsf{P}(\boldsymbol{x}^1)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{P}(\boldsymbol{x}^2)$
  end
  $\boldsymbol{y}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[32]$
  $\boldsymbol{y}^2 \leftarrow \boldsymbol{x}^2$
end

only for the shadow register but also for every internal register that is used for holding sensitive data.

Moreover, since the micro-controller has two internal pipeline stages [2], we have to ensure that a masked value and its corresponding mask are never processed consecutively, i.e., they never appear in the same pipeline. In particular for the substitution layer, this may occur if the two shares are loaded to perform the table look-up. In order to avoid insertion of unnecessary NOP operations, we start with loading the entire 64 bits of the first share into eight registers before we load the next 64 bits of the second share into another eight registers. Still, we process the last 8-bit chunk of the first share and the first 8-bit chunk of the second share in the same pipeline. However, since the mask is drawn uniformly from a random source, it is unrelated to the first share which is masked by another random value.

Threshold Implementation. Algorithm 3.2 presents the pseudo-code for our TI design according to Section 2.4, using the decomposition based on $\mathcal{Q}_{12}$ and an affine transformation A''' as described in Equation (3). As mentioned before, this decomposition improves the efficiency by limiting the number of look-up tables that have to be stored (one 8-bit to 4-bit and one 4-bit to 4-bit).

Again, processing the shared values has to be done carefully in order to avoid side-channel leakage due to internal (shadow) registers and the pipeline of the micro-controller. Fortunately, compared to the classical Boolean masking – due to its *non-completeness* property – our TI design always processes only two shares at once. However, special care has to be taken for the order of processing the individual shares (for all implementation variants). For instance, starting with the addition of the round key to the first share $\boldsymbol{x}^1$ and updating this share using the look-up table T would result in unintentional leakage since both shares $\boldsymbol{x}^2$ and $\boldsymbol{x}^3$ have to be loaded after $\boldsymbol{x}^1$ has been processed. Due to this, our

Algorithm 3.2: THRESHOLD IMPLEMENTATION

Input : $\bar{\boldsymbol{x}} = (\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$: *shared plaintext*
$\boldsymbol{k}$: *cipher key*
Output : $\bar{\boldsymbol{y}} = (\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3)$: shared ciphertext

begin
  $\boldsymbol{rk} \leftarrow KeySchedule(\boldsymbol{k})$
  for $i \leftarrow 1$ **to** 31 **do**
    $\boldsymbol{x}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[i]$
    $\boldsymbol{t}^3 \leftarrow \mathsf{T}(\boldsymbol{x}^1, \boldsymbol{x}^2)$
    $\boldsymbol{t}^2 \leftarrow \mathsf{T}(\boldsymbol{x}^3, \boldsymbol{x}^1)$
    $\boldsymbol{t}^1 \leftarrow \mathsf{T}(\boldsymbol{x}^2, \boldsymbol{x}^3)$
    $\boldsymbol{t}^3 \leftarrow \mathsf{A}'''(\boldsymbol{t}^3)$
    $\boldsymbol{t}^2 \leftarrow \mathsf{A}'''(\boldsymbol{t}^2)$
    $\boldsymbol{t}^1 \leftarrow \mathsf{A}'''(\boldsymbol{t}^1)$
    $\boldsymbol{x}^3 \leftarrow \mathsf{T}(\boldsymbol{t}^1, \boldsymbol{t}^2)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{T}(\boldsymbol{t}^3, \boldsymbol{t}^1)$
    $\boldsymbol{x}^1 \leftarrow \mathsf{T}(\boldsymbol{t}^2, \boldsymbol{t}^3)$
    $\boldsymbol{x}^1 \leftarrow \mathsf{P}(\boldsymbol{x}^1)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{P}(\boldsymbol{x}^2)$
    $\boldsymbol{x}^3 \leftarrow \mathsf{P}(\boldsymbol{x}^3)$
  end
  $\boldsymbol{y}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[32]$
  $\boldsymbol{y}^2 \leftarrow \boldsymbol{x}^2$
  $\boldsymbol{y}^3 \leftarrow \boldsymbol{x}^3$
end

implementation starts with updating the third share first before the remaining shares are processed (see Algorithm 3.2).

Classical 2nd-Order Boolean Masking. This implementation, as presented in Algorithm 3.3, uses three shares (similar to the TI), but the masked S-box instead is realized by a single look-up table $\mathsf{T}(.,.,.)$ as described in Section 2.4.

In particular the realization of a 12-bit to 12-bit look-up table on an 8-bit micro-controller is challenging. On one hand, the 12-bit look-up table will increase the memory requirements significantly. On the other hand, 12-bit addresses can be realized easily by combining two 8-bit registers but at the cost of wasting the four most significant bits. Still, we opted for this approach in order to reduce the overhead due to additional and more complex control logic as well as to guarantee a constant-time implementation (i.e., to prevent data-dependent timings).

Classical 2nd-Order Boolean Masking with Affine Transformation. Eventually, Algorithm 3.4 extends the classical second-order Boolean masking using an affine transformation in order to realize the masked S-box. In particular, the table look-up is done twice and interleaved by applying the affine transformations

Algorithm 3.3: Classical 2nd-Order Boolean Masking

Input : $\bar{\boldsymbol{x}} = (\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$: *shared plaintext*
$\boldsymbol{k}$: *cipher key*
Output : $\bar{\boldsymbol{y}} = (\boldsymbol{y}^1, \boldsymbol{y}^2, \boldsymbol{y}^3)$: shared ciphertext

begin
  $\boldsymbol{rk} \leftarrow KeySchedule(\boldsymbol{k})$
  for $i \leftarrow 1$ **to** 31 **do**
    $\boldsymbol{x}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[i]$
    $\bar{\boldsymbol{x}} \leftarrow \mathsf{T}(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$
    $\boldsymbol{x}^1 \leftarrow \mathsf{P}(\boldsymbol{x}^1)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{P}(\boldsymbol{x}^2)$
    $\boldsymbol{x}^3 \leftarrow \mathsf{P}(\boldsymbol{x}^3)$
  end
  $\boldsymbol{y}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[32]$
  $\boldsymbol{y}^2 \leftarrow \boldsymbol{x}^2$
  $\boldsymbol{y}^3 \leftarrow \boldsymbol{x}^3$
end

(see Equation (3)). However, this variant still has to face the same challenges as the former approach. The motivation to include this variant in our analyses is to examine whether the algebraic degree of the underlying function of the masked look-up table has any effect on observable SCA leakage. The former implementation variant is not formed following the TI principles; its look-up tables have only been extracted from a TI construction hence fulfilling the uniformity. However, this variant additionally stays with 3 shares per quadratic function.

3.3 Comparison

Table 3 provides a summary and comparison of our implementation variants in terms of code size, memory usage (SRAM), and performance (clock cycles). Since all implementations use the same key schedule routine, 256 B of the SRAM usage of all variants are due to the 32 derived round keys and only the remaining memory usage is implementation-specific.

The code size of each implementation comprises the key schedule and the round function including all look-up tables which are stored in the flash memory. Obviously, the classical 2nd-order Boolean masking schemes have the largest code due to the 12-bit to 12-bit look-up tables that require complex handling on an 8-bit micro-controller. Similarly, the TI design has a slightly larger code size than the classical 1st-order Boolean masking due to its more extensive substitution layer that has to handle three shares.

Considering the performance, we measured the latency in terms of clock cycles using the simulator of the Atmel Studio 6.2 environment. In order to prevent any vulnerabilities against timing attacks, we ensured data-independent and constant execution time for all of our implementations. Notably, the latency is

Algorithm 3.4: Classical 2nd-Order Boolean Masking with Affine Transformation

Input : $\bar{\boldsymbol{x}} = (\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$: *shared plaintext*
$\boldsymbol{k}$: *cipher key*
Output : $\bar{\boldsymbol{y}} = (\boldsymbol{y^1}, \boldsymbol{y^2}, \boldsymbol{y^3})$: shared ciphertext

begin
  $\boldsymbol{rk} \leftarrow KeySchedule(\boldsymbol{k})$
  for $i \leftarrow 1$ **to** 31 **do**
    $\boldsymbol{x}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[i]$
    $\bar{\boldsymbol{x}} \leftarrow \mathsf{T}(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$
    $\boldsymbol{x}^1 \leftarrow \mathsf{A}'''(\boldsymbol{x}^1)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{A}'''(\boldsymbol{x}^2)$
    $\boldsymbol{x}^3 \leftarrow \mathsf{A}'''(\boldsymbol{x}^3)$
    $\bar{\boldsymbol{x}} \leftarrow \mathsf{T}(\boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3)$
    $\boldsymbol{x}^1 \leftarrow \mathsf{P}(\boldsymbol{x}^1)$
    $\boldsymbol{x}^2 \leftarrow \mathsf{P}(\boldsymbol{x}^2)$
    $\boldsymbol{x}^3 \leftarrow \mathsf{P}(\boldsymbol{x}^3)$
  end
  $\boldsymbol{y}^1 \leftarrow \boldsymbol{x}^1 \oplus \boldsymbol{rk}[32]$
  $\boldsymbol{y}^2 \leftarrow \boldsymbol{x}^2$
  $\boldsymbol{y}^3 \leftarrow \boldsymbol{x}^3$
end

particularly dependent on the number of shares and decomposition of the S-box. Hence, the classical 1st-order Boolean masking scheme has the smallest latency, since it operates on only two shares and the substitution layer is realized as a single table look-up. Consequently, the TI design has the highest number of clock cycles, since it uses three shares and the S-box is realized by six table look-ups interleaved with three affine transformations.

4 Evaluation

4.1 Measurement Setup

For the SCA evaluations, by means of a digital oscilloscope we measured the voltage drop over an $1\,\Omega$ resistor placed at the GND path of the target micro-controller. During the measurements, the micro-controller was operating at a low clock frequency of 4 MHz (provided internally), and the traces have been collected at a sampling rate of 125 MS/s. We have also made use of one of the I/O pins of the micro-controller to provide a stable and jitter-free signal to trigger the oscilloscope.

Table 3: Comparison between Different Implementation Variants

Variant	Code (*Bytes*)	Memory (*Bytes*)	Time (*Cycles*)
Classical 1st-Order Boolean Masking	1 542	272	53 861
Threshold Implementation	1 576	304	165 802
Classical 2nd-Order Boolean Masking	9 328	280	91 557
Classical 2nd-Order Boolean Masking with Affine	9 448	280	148 012

4.2 Non-Specific Statistical t-Test

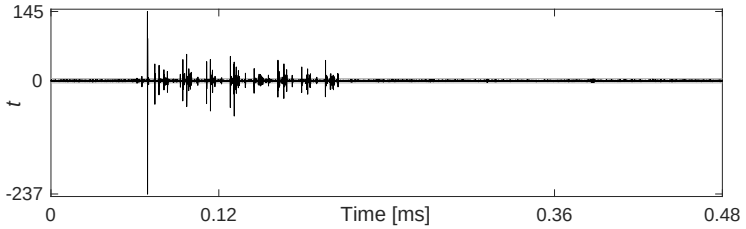
During the entire measurements, we kept the key constant (allowing us to forgo masking of the key schedule), and provided the input masks externally, i.e., the random $\boldsymbol{m}_t$ have been generated by a PC and in addition to the masked plaintexts $\boldsymbol{x}_t$ are sent to the micro-controller. This way we could easily examine and ensure the uniform distribution of the masks. As a metric to evaluate the existence of 1st-order leakage in our implementations, we applied the fixed versus random t-test [16,41]. In short, a fixed plaintext is selected, and prior to every measurement a coin is flipped, based on that either the fixed plaintext or a random plaintext is given to the micro-controller. Indeed, such a t-test can examine whether there is a detectable leakage in the measurements without giving any impression about its exploitability. However, the intuition is that if the leakage is exploitable, it is also detectable. Therefore, as a conservative condition, if there is no detectable leakage, no exploitable leakage exists.

4.3 Results

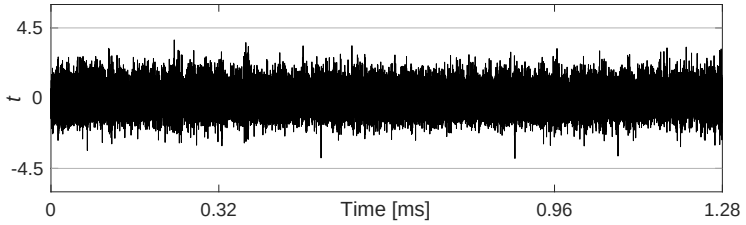
For each of our considered implementation variants we collected 100 000 power traces following the procedure explained in [41]. In our analyses we focused on the first cipher round as well as on a 1st-order t-test.

Figure 1 presents the corresponding evaluation results for all four implementations. Interestingly, it can be seen that the TI design is the only variant which does not exhibit detectable leakage. In all other implementations, either with 2 shares or 3 shares, 1st-order leakage is detectable. We have localized the points in time where the t statistics exceeds the 4.5 threshold; they are exactly corresponding to the timing of the performed table look-ups.

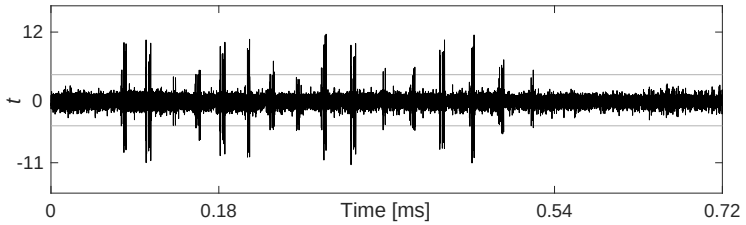
Notably, we observe the 1st-order leakage for both variants of the classical 2nd-order Boolean masking. We should highlight that the only difference between these two implementations and the TI design is the way the look-up tables are realized. In these two variants all three shares are present at the input of the table look-ups while in the TI design at most two shares form the input of every table look-up. Our intuition is that the observed leakage is due to the unknown details of the internal architecture of the underlying micro-controller. Similar to the shadow register which we could identify to buffer data for load and store operations, further hidden architecture details of the memory bus and unit could be responsible for the detected leakage. To this end, it seems that the memory



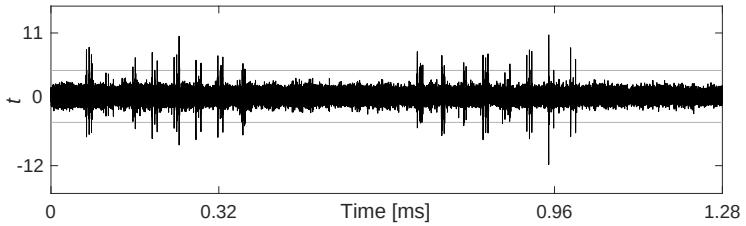
(a) Classical 1st-Order Boolean Masking



(b) Threshold Implementation



(c) Classical 2nd-Order Boolean Masking



(d) Classical 2nd-Order Boolean Masking with Affine Transformations

Figure 1: SCA evaluation results based on 1st-order non-specific t-test using 100 000 power traces.

control unit exhibits non-linear leakage depending on the given address during the table look-ups. Hence, following the non-completeness principle of TI seems to be a suitable choice which avoids all three shares to appear as an address for a look-up, since it is hardly possible to get the necessary but missing details

of the architecture. We should emphasize that we have just shown that if all shares appear at the address of a table look-up, there exists detectable first-order leakage. On one hand, with the current experiments we cannot comment on the exploitability of such observed leakages. On the other hand, the very high t-test statistics for the classical 1st-order Boolean masking shown in Figure 1(a) induce the exploitability of the leakage.

5 Conclusion

In this paper, we have investigated the application of Threshold Implementations for software implementations in order to provide first-order security against side-channel analysis attacks. In this context, we have developed and implemented a classical first-order Boolean masking scheme, two second-order Boolean masking schemes and a first-order TI. In general, our findings show that the classical Boolean masking schemes (through pre-computed look-up tables) could not be implemented securely on the chosen AVR micro-controller. More precisely, only the first-order TI variant does not exhibit detectable first-order leakage using up to 100 000 power measurements.

In addition to our practical side-channel evaluation, we could efficiently realize the Threshold Implementation in terms of code size and memory requirements, eventually implementing the TI variant with 1 576 B of code and 304 B of memory which is close to the classical Boolean masking with only two shares. However, the code size and memory reduction comes at cost of increased latency results in terms of clock cycles. In particular, the TI requires about 165 k cycles whereas the first-order classical Boolean masking takes only 53 k clock cycles. All in all, this work shows that although TI has been proposed for hardware platforms, the concept can be transferred and applied for software as well in order to realize first-order secure implementations.

Acknowledgments

The work described in this paper has been supported in part by the German Federal Ministry of Education and Research BMBF (grant nr. 16KIS0602 VeriSec).

References

1. Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.
2. Atmel. *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash*, 02 2003. Rev. 1142E.
3. Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.
4. Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic application of power analysis countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.
5. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. A low-entropy first-degree secure provable masking scheme for resource-constrained devices. In *Proceedings of the Workshop on Embedded Systems Security, WESS 2013, Montreal, Quebec, Canada, September 29 - October 4, 2013*, pages 7:1–7:10. ACM, 2013.

6. Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015.
7. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.
8. D. Canright and Lejla Batina. A very compact "perfectly masked" s-box for AES. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, 6th International Conference, ACNS 2008, New York, NY, USA, June 3-6, 2008. Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 446–459, 2008.
9. Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain. Higher-Order Masking Schemes for S-Boxes. In *FSE 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2012.
10. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
11. Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In *CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007.
12. Aurélien Francillon and Pankaj Rohatgi, editors. *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*. Springer, 2014.
13. Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. Affine masking against higher-order side channel analysis. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, volume 6544 of *Lecture Notes in Computer Science*, pages 262–280. Springer, 2010.
14. Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Secure Multiplicative Masking of Power Functions. In *ACNS 2010*, volume 6123 of *Lecture Notes in Computer Science*, pages 200–217, 2010.
15. Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Thwarting Higher-Order Side Channel Analysis with Additive and Multiplicative Maskings. In *CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2011.
16. 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.
17. Dahmun Goudarzi and Matthieu Rivain. How Fast Can Higher-Order Masking Be in Software? In *EUROCRYPT 2017*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, 2017.
18. Vincent Grosso, Emmanuel Prouff, and François-Xavier Standaert. Efficient Masked S-Boxes Processing - A Step Forward -. In *AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2014.

19. Vincent Grosso, François-Xavier Standaert, and Emmanuel Prouff. Low entropy masking schemes, revisited. In Francillon and Rohatgi [12], pages 33–43.
20. Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 463–481, 2003.
21. Sebastian Kutzner, Phuong Ha Nguyen, and Axel Poschmann. Enabling 3-Share Threshold Implementations for all 4-Bit S-Boxes. In *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, pages 91–108, 2013.
22. Sebastian Kutzner, Phuong Ha Nguyen, Axel Poschmann, and Marc Stöttinger. Minimizing S-Boxes in Hardware by Utilizing Linear Transformations. In *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, pages 235–250, 2014.
23. Sebastian Kutzner, Phuong Ha Nguyen, Axel Poschmann, and Huaxiong Wang. 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*, pages 99–113, 2013.
24. Sebastian Kutzner and Axel Poschmann. On the security of RSM - presenting 5 first- and second-order attacks. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2014.
25. Houssem Maghrebi, Sylvain Guilley, and Jean-Luc Danger. Leakage squeezing countermeasure against high-order attacks. In Claudio Agostino Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, volume 6633 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2011.
26. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
27. Amir Moradi, Sylvain Guilley, and Annelie Heuser. Detecting hidden leakages. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2014.
28. Amir Moradi and Alexander Wild. Assessment of Hiding the Higher-Order Leakages in Hardware - What Are the Achievements Versus Overheads? In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 453–474, 2015.
29. Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. RSM: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1173–1178. IEEE, 2012.
30. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.
31. Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES s-box. In Henri Gilbert and

Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.
32. Kostas Papagiannopoulos and Nikita Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.
33. Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-Channel Resistant Crypto for Less than 2,300 GE. *Journal of Cryptology*, 24(2):322–345, 2011.
34. Bart Preneel and Tsuyoshi Takagi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.
35. Emmanuel Prouff and Matthieu Rivain. A Generic Method for Secure SBox Implementation. In *WISA 2007*, volume 4867 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2008.
36. Emmanuel Prouff and Thomas Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In Preneel and Takagi [34], pages 63–78.
37. Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. In *FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2008.
38. Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In *CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
39. Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In *CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009.
40. Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Affine Equivalence and Its Application to Tightening Threshold Implementations. In *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, pages 263–276, 2015.
41. Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.
42. Kai Schramm and Christof Paar. Higher Order Masking of the AES. In *CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.
43. Xin Ye and Thomas Eisenbarth. On the vulnerability of low entropy masking schemes. In Francillon and Rohatgi [12], pages 44–60.