# Arithmetic Addition over Boolean Masking
## Towards First- and Second-Order Resistance in Hardware

Tobias Schneider, Amir Moradi, Tim Güneysu

Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
{tobias.schneider-a7a, amir.moradi, tim.gueneysu}@rub.de

**Abstract.** A common countermeasure to thwart side-channel analysis attacks is algorithmic masking. For this, algorithms that mix Boolean and arithmetic operations need to either apply two different masking schemes with secure conversions or use dedicated arithmetic units that can process Boolean masked values. Several proposals have been published that can realize these approaches securely and efficiently in software. But to the best of our knowledge, no hardware design exists that fulfills relevant properties such as efficiency and security at the same time.
In this paper, we present two design strategies to realize a secure and efficient arithmetic adder for Boolean-masked values. First, we introduce an architecture based on the ripple-carry adder that targets low-cost applications. The second architecture is based on a pipelined Kogge-Stone adder and targets high-performance applications. In particular, all our implementations adopt the *threshold implementation* approach to improve their resistance against SCA attacks even in the presence of glitches. We evaluated the security of our designs practically against SCA using a non-specific statistical $t$-test. Based on our analysis, we show that our constructions not only achieve resistance against first- and (univariate) second-order attacks but also require fewer random bits per operation compared to any existing software-based approach.

**Keywords:** Side-channel analysis, threshold implementation, Boolean masking, arithmetic modular addition

## 1 Introduction

Side-channel analysis (SCA) poses a serious threat to any cryptographic implementation. If no dedicated countermeasure is applied, the secret of the underlying device can be easily extracted by SCA. A popular approach to increase the security of a cryptographic implementation is the use of *masking*. It is achieved by blinding the processed values by means of random masks [21] so that it should become impossible for an attacker to predict intermediate values.

To date there exist several types of masking schemes that differ in the level of abstraction and the target operation. In this work we focus on the techniques developed to be applied at algorithmic level, e.g., Boolean and arithmetic masking, which need to be adjusted according to the underlying cryptographic

algorithm [21]. Note that nearly all proposed ciphers employ both logical and arithmetic operations.

As an example, ARX-based designs consist of three operations: integer addition, rotation, and XOR. Such constructions are the foundation for block ciphers (like FEAL [23] or Threefish [13]), stream ciphers (Salsa20 [5], ChaCha [4], HC-128 [35]) and hash functions (BLAKE [2], Skein [13]). There are further examples that also include a mixture of Boolean and arithmetic operations like the TEA family of block ciphers [34] and SHA-2 [30]. To realize a masked implementation of these constructions, one option is to employ both Boolean and arithmetic masking schemes. Rotation and XOR operations can be protected by Boolean masking, while arithmetic masking is advantageous for the addition operations. However, the required conversions between both operations can also be the target of an SCA attacker and hence need to be implemented in a secure way. In particular, many existing results discussing this method identified the conversion between arithmetic and Boolean masking as a major hurdle [3, 9, 14].

**Related Works**

We now briefly highlight several works on the conversion between Boolean and arithmetic masking. The conversion techniques can be categorized into those which use *precomputation* [12] and those *without precomputation* [17]; however most of them were designed specifically for software platforms. Unfortunately, these constructions cannot be easily mapped to a dedicated hardware module without violating their claims on security. Roughly speaking, this is mainly due to critical glitches that occur inside masked circuits [22]. To avoid this problem, every step would need to be separated by a register stage which would be detrimental to the performance.

We like to remark that a hardware design for such conversions has been proposed in [15], but since both the mask and masked data are involved in the processes of the proposed techniques, such constructions are expected to still have first-order leakages (see [25]). Another problem is the transformation of conversion algorithms to higher orders. It has been shown in [10] how to secure the conversions against higher-order attacks, but this feature comes with a prohibitive overhead for any cryptographic implementation.

Along the same lines, in order to avoid the conversions a technique to securely perform modular arithmetic addition on Boolean masked operands has been introduced in [18]. However, this scheme has been developed to be used in software applications and cannot be easily applied on a hardware platform where performance is a key factor.

Recently, an approach was developed in [11] which uses the Kogge-Stone adder as a basis. But the conversion and masked addition requires more random bits compared to the solutions from [17] and [18] and are only faster for larger bit sizes (i.e., 64 bits). Still their focus lies on software applications which makes them inefficient in hardware.

**Contributions**
The target of this work is to design efficient hardware modules for modular addition of Boolean masked operands. More precisely, our goal is to develop a similar technique such as [18] for a hardware platform.

Since masked hardware designs face severe challenges due to glitches, we apply the concept of threshold implementations (TI) [29] that can satisfy the security requirements even in the presence of glitches. TI combines the ideas of Boolean secret sharing and multiparty computation. It has previously been applied to realize the secure hardware design of symmetric ciphers [6, 26, 31]. Although it has initially been developed with respect to first-order security, its extension to higher orders has been recently introduced [7].

In this paper we consider two factors to design the aforementioned module: (a) *throughput* and (b) *SCA security order*. With respect to performance (i.e., throughput) we consider two designs to implement a 32-bit arithmetic adder that is required by many cryptographic algorithms:

1. Ripple-Carry Adder (RCA) that requires 32 clock cycles to perform a complete addition, and
2. Kogge-Stone Adder (KSA) with 6 clock cycles latency and a fully pipelined architecture.

We present the first-order and (univariate) second-order secure threshold implementation of the two above mentioned designs. We show that our designs not only outperform the inefficient approaches of [10] but also reduce the number of fresh random mask bits required for each addition. We also present practical SCA evaluations performed on a Spartan-6 FPGA to confirm the claimed security levels. To the best of our knowledge, our four proposed architectures are the only available hardware-dedicated solutions that are supported by security proofs as well as by practical investigations.

## 2 Background

In this section, we introduce the used notations and present the basic ideas behind our designs.

### 2.1 Notations

In the following all equations are bit-level operations. An $n$-bit integer operand $a$ is represented as $(a_{n-1}a_{n-2}\cdots a_1a_0)$ where $a_0$ is the least significant bit. These integers are split up into shares of which the $j$-th share of a bit $a_{i\in\{0,\ldots,n-1\}}$ is denoted by $a_i^j$. Inside the equations two Boolean operators are used: $\oplus$ denotes the logical XOR and $\wedge$ the logical AND. AND operators are always evaluated before any XOR operators.

### 2.2 Ripple-Carry Adder

In [18] the authors presented a way to securely add two Boolean masked values. Instead of three conventional steps (conversion, addition, reconversion), the addition can be implemented in just one step. Depending on the application, this can signficantly increase the performance over the classical approach. The algorithm introduced in [18] is based on a ripple-carry adder (RCA). This adder has been rewritten into a sequence of Boolean operations that take the Boolean masks into consideration. The algorithm is word-oriented for efficiency in software but not in hardware.

Similarly, our design is based on the basic algorithm described in [18]. The underlying algorithm builds on the fact that one bit of sum $s$ can be computed as

$$s_i = a_i \oplus b_i \oplus c_i \ . \tag{1}$$

Therefore, the addition is replaced by a simple XOR of the two operands $a$ and $b$ and the carry $c$. The only unknown part in such an equation is the carry bit which can be computed using a recursive formula

$$c_{i+1} = a_i \wedge b_i \quad \oplus \quad a_i \wedge c_i \quad \oplus \quad b_i \wedge c_i, \tag{2}$$

where $c_0 = 0$. The costly part of the RCA is the recursive carry computation. Its function has to be evaluated iteratively which leads to a high circuit depth in case of a fully combinatorial design.

### 2.3 Kogge-Stone Adder

Another addition circuit with a lower depth is given by the Kogge-Stone adder (KSA) [19] that splits the carry generation into generate ($g$) and propagate ($p$) functions. Instead of evaluating the carry function recursively, the KSA benefits from a tree-like structure and achieves a logarithmic complexity. For a hardware design, a KSA can significantly increase the overall performance.

The basic structure of KSA for $n = 4$-bit operands is shown in Figure 1. For operands $a$ and $b$ it computes the carry bits in three steps. During preprocessing the initial $g_i$ and $p_i$ values are generated as

$$g_i = a_i \wedge b_i \ , \qquad p_i = a_i \oplus b_i \ . \tag{3}$$

In the following stages a function is used to combine the $g$ and $p$ values of different bit positions. This function receives 4 bits as input and returns 2 output bits. For $i > j$ the output values are computed as

$$g_{i:j} = g_i \oplus g_j \wedge p_i \ , \qquad p_{i:j} = p_i \wedge p_j \ . \tag{4}$$

After $\log_2(n = 4) = 2$ stages, the computation is finished and all carry bits can be derived as

$$c_{i \in \{2 \ldots n\}} = g_{i-1:0} \ , \qquad c_1 = g_0 \ , \qquad c_0 = 0.$$

Finally the sum $s$ can be obtained according to Equation (1).

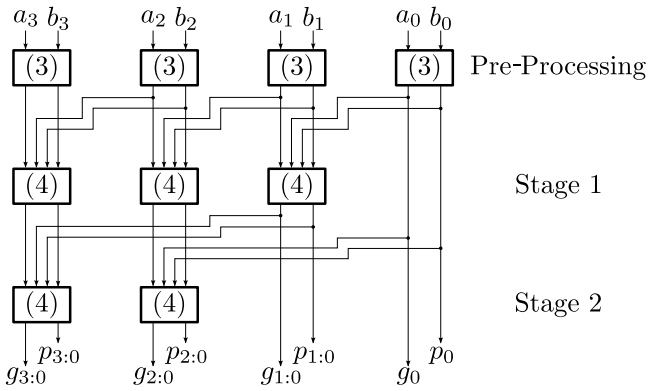



**Fig. 1.** The structure of the carry generation for 4-bit operands using the KSA

### 2.4 Threshold Implementations

In order to realize secure masked implementations and avoid the leakage caused by glitches (e.g., [22, 25]), the threshold implementation (TI) scheme has been introduced and developed in [7, 27–29]. Based on the algebraic degree $t$ of the targeted non-linear function (Sbox) as well as the desired order[1] of security $d$, the minimum number of input shares $s_{in}$ and the minimum number of output shares $s_{out}$ are defined as

$$s_{in} = t \times d + 1 , \qquad s_{out} = \binom{s_{in}}{t}.$$

The input $x$ of the e.g., Sbox is represented by $(x^1, \ldots, x^{s_{in}})$ in such a way that $x = \bigoplus_{i=1}^{s_{in}} x^i$. The output of the TI of the corresponding Sbox $(y^1, \ldots, y^{s_{out}})$ should be also a shared representation of $y = S(x) = \bigoplus_{j=1}^{s_{out}} y^j$ while each $y^j$ is provided by a component function $f^j(\ldots)$ over a subset of input shares $(x^1, \ldots, x^{s_{in}})$. This property is known as *correctness* while *non-completeness* is referred to the fact that any $d$ (security order) selection of component functions $f^1(\ldots), \ldots, f^{s_{out}}(\ldots)$ is independent of at least one input share. These two properties are relatively easy to achieve, but the third property *uniformity* is challenging. As the security of masking schemes is based on the uniform distribution of the masks, the output of a TI Sbox must be uniform as it is used as input in further parts of the implementation.

Suppose that for a certain input x all possible sharings $\left\{ ({}_1x^1, \ldots, {}_1x^{s_{in}}), ({}_2x^1, \ldots, {}_2x^{s_{in}}), \ldots, ({}_px^1, \ldots, {}_px^{s_{in}}) \right\}$ are given to the TI Sbox. The tuple

[1] With respect to [32] only univariate security at order $d > 1$ can be achieved.

$\left(f^1(\ldots), \ldots, f^{s_{out}}(\ldots)\right)$ should be drawn uniformly from the set $\Big\{({}_1y^1, \ldots, {}_1y^{s_{out}}), ({}_2y^1, \ldots, {}_2y^{s_{out}}), \ldots, ({}_qy^1, \ldots, {}_qy^{s_{out}})\Big\}$ as all possible sharings of y = S(x).

An important point is that the output of the component functions must be stored in dedicated registers to avoid the propagation of glitches. Another issue is related to the uniformity of the TI functions of security order $d > 1$. In such a case, the number of output shares $s_{out}$ is usually higher than the number of input shares $s_{in}$; hence uniformity cannot be achieved. Therefore, some of the registered output shares should be combined to reduce the number of output shares to $s_{in}$ at most. After such a combination the uniformity can be examined. For more detailed information, the interested reader is referred to the original works [7, 29].

## 3 Implementation

We present two designs of a modulo $2^{32}$ adder that provides resistance against first- and second-order SCA. This is a quite common type of addition used in many cryptographic algorithms (e.g., Salsa20, HC-128, SHA-2), but our architectures can be also easily adapted to other bit lengths.

### 3.1 Ripple-Carry Adder (First-Order SCA-Resistant)

Based on the scheme presented in Section 2.2 we build a first-order SCA-resistant adder. To achieve this, Equations (1) and (2) should be transformed to meet the three required TI properties.

Given that Equation (2) is of degree 2, at least 3 shares (for input as well as for output) are necessary. It is supposed that each processed value, e.g., $a_i$, is split into 3 shares as $(a_i^1, a_i^2, a_i^3)$. In case of Equation (1), due to its linearity the shares are easily combined via XOR as

$$s_i^1 = a_i^1 \oplus b_i^1 \oplus c_i^1 \ , \qquad s_i^2 = a_i^2 \oplus b_i^2 \oplus c_i^2 \ , \qquad s_i^3 = a_i^3 \oplus b_i^3 \oplus c_i^3 \ . \tag{5}$$

As mentioned before, Equation (2) is non-linear and has algebraic degree of 2. Following *direct sharing* approach represented in [8], we can construct a correct and uniform shared implementation of such a function. The shares of the carry bit can be computed as

$$c_{i+1}^1 = a_i^2 \wedge b_i^2 \oplus a_i^2 \wedge b_i^3 \oplus a_i^3 \wedge b_i^2 \oplus a_i^2 \wedge c_i^2 \oplus a_i^2 \wedge c_i^3 \oplus a_i^3 \wedge c_i^2 \oplus b_i^2 \wedge c_i^2 \oplus b_i^2 \wedge c_i^3 \oplus b_i^3 \wedge c_i^2 \tag{6}$$

$$c_{i+1}^2 = a_i^3 \wedge b_i^3 \oplus a_i^3 \wedge b_i^1 \oplus a_i^1 \wedge b_i^3 \oplus a_i^3 \wedge c_i^3 \oplus a_i^3 \wedge c_i^1 \oplus a_i^1 \wedge c_i^3 \oplus b_i^3 \wedge c_i^3 \oplus b_i^3 \wedge c_i^1 \oplus b_i^1 \wedge c_i^3 \tag{7}$$

$$c_{i+1}^3 = a_i^1 \wedge b_i^1 \oplus a_i^1 \wedge b_i^2 \oplus a_i^2 \wedge b_i^1 \oplus a_i^1 \wedge c_i^1 \oplus a_i^1 \wedge c_i^2 \oplus a_i^2 \wedge c_i^1 \oplus b_i^1 \wedge c_i^1 \oplus b_i^1 \wedge c_i^2 \oplus b_i^2 \wedge c_i^1 \tag{8}$$

Here we should note that Equations (1) and (2) can be seen as a function $f : (a_i, b_i, c_i) \mapsto (s_i, c_{i+1})$. At a first glance one may think of examining the uniformity of the $(s_i, c_{i+1})$ tuple[2]. However, such a tuple is never supplied to

[2] If sharing of $x$ and $y$ are uniform, the tuple of sharing of $(x, y)$ is not necessarily uniform if $x$ and $y$ are not independent.

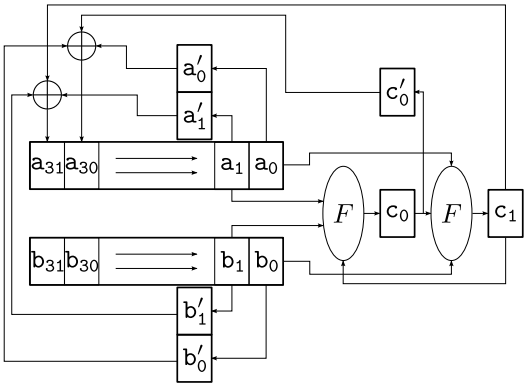



**Fig. 2.** Structure of the first-order secure adder based on RCA

any function within the RCA algorithm. Note that $s_i$ is an output bit and is not propagated while $c_{i+1}$ is given to the next stage where it is combined with $a_{i+1}$ and $b_{i+1}$ which are independent of $c_{i+1}$. Hence the uniformity of $c_{i+1}$ suffices to fulfill the corresponding property.

During the implementation of such a design we encountered an issue that has never been reported before. The output of the shared carry computation function (Equation (6) to (8)) cannot be directly used as feedback signal since the output of a function from a previous cycle is used as input in the next clock cycle.

As a remedy we constructed a two-stage design as depicted in Figure 2. The three shares of the two operands $a$ and $b$ are stored in shift registers. The RCA algorithm and the deployment of shift registers supports an efficient scanning of operand bits. Two instances of the shared carry computation function are implemented whose outputs are stored in carry registers $\mathtt{c}_0$ and $\mathtt{c}_1$. The carry registers are enabled alternately while the other intermediate registers ($\mathtt{c}'_0$, $\mathtt{a}'_0$, $\mathtt{a}'_1$, $\mathtt{b}'_0$ and $\mathtt{b}'_1$) are enabled every second clock cycle synchronized with that of $\mathtt{c}_1$. The operand registers are also shifted two bits every other second clock cycle. The additional registers, i.e., $\mathtt{c}'_0$, $\mathtt{a}'_0$, $\mathtt{a}'_1$, $\mathtt{b}'_0$, and $\mathtt{b}'_1$ synchronize the computation of the sum bits, which need to be performed one clock cycle after that of the carry bits. Note that we use the shift register of operand $a$ to save the result of the addition.

Another issue is related to the first stage, i.e., when $i = 0$. In our designs we suppose that input carry $c_0 = 0$ so that $(c_0^1, c_0^2, c_0^3)$ should be a shared representation of 0. Therefore, both carry registers have to be initialized with a random set representing 0. In other words, our design requires four fresh mask bits $fm_1, \ldots, fm_4$ only at the start of the addition to initialize $\mathtt{c}_0$ and $\mathtt{c}_1$ with $(fm_1, fm_2, fm_1 \oplus fm_2)$ and $(fm_3, fm_4, fm_3 \oplus fm_4)$ respectively. Note that all other stages of our design do not require fresh random bits leading to an efficient design with respect to the number of required fresh mask bits. For instance, our

design is considerably more efficient than the solutions proposed in [18], [10] and [11].

### 3.2 Ripple-Carry Adder (Second-Order SCA-Resistant)

The design described above can be simply transformed to support resistance against higher-order attacks. We now present a solution for the second-order resistant design. We increase the number of input shares $s_{in}$ to 5 and all corresponding functions have to be chosen according to the principles of higher-order TI [7].

Equation (5) just needs to be adapted to the increased number of shares. The computation of the carry has to be split up into two steps. In the first step, $s_{out} = 10$ component functions generate 10 output shares. Following the same concept presented in [7], these intermediate shares are then again reduced to 5 shares in the second step.

No major changes to the basic structure depicted in Figure 2 are necessary for implementation. Just the registers have to be adjusted to the increased number of shares and the $F$ block is split by an additional register stage. All uniform equations for the $F$ function, obtained by direct sharing, are described in detail in the appendix of this work.

As a consequence, the amount of utilized resources increases and the number of clock cycles needed for the carry computation doubles. Just as before, the carry registers need fresh randomness during the initialization. Therefore, the number of required fresh random masks increases to 8 bits.

### 3.3 Kogge-Stone Adder (First-Order SCA-Resistant)

The design based on the RCA has low requirements for space and randomness. However, the number of clock cycles for one addition grows linearly with the bit length of the operands. For increased performance we therefore implemented a design that uses a KSA as foundation and which is still secure against first-order attacks.

Equations (3) and (4) need to be split into shares. Since all the corresponding formulas are of degree two, similar to that of the RCA, at least 3 shares are required to realize a functional TI.

The two outputs of the preprocessing step are both given to the next stages; thus, the uniformity of each tuple $(g_i, p_i)$ must be taken into account. One part of Equation (3) needs to be implemented by the AND of the two operands for which no uniform TI with 3 shares exists [29]. For this, fresh mask bits have to be used to make it uniform (see remasking in [8, 26]). In our design, we adopted the solution from [8] with only a single virtual share. One fresh random bit $m_i$ is required for every invocation of the function in the preprocessing step:

$$g_i^1 = a_i^2 \wedge b_i^2 \oplus a_i^2 \wedge b_i^3 \oplus a_i^3 \wedge b_i^2 \oplus m_i \tag{9}$$

$$g_i^2 = a_i^3 \wedge b_i^3 \oplus a_i^1 \wedge b_i^3 \oplus a_i^3 \wedge b_i^1 \oplus a_i^1 \wedge m_i \oplus b_i^1 \wedge m_i \tag{10}$$

$$g_i^3 = a_i^1 \wedge b_i^1 \oplus a_i^1 \wedge b_i^2 \oplus a_i^2 \wedge b_i^1 \oplus a_i^1 \wedge m_i \oplus b_i^1 \wedge m_i \oplus m_i \ . \tag{11}$$

Further, preprocessing involves another linear XOR-function. We implement this part similar to Equation (5). Both functions as well as their joint output $(g_i, p_i)$ fulfill the three TI properties.

The preprocessing step is followed by stages in which the $g$ and $p$ values are updated according to Equation (4). These two functions can be considered as a 4-bit to 2-bit mapping. Similar to the preprocessing step, the tuple of the 2-bit output has to be considered for the uniformity check. For the computation of the $g$ part of Equation (4) (as an AND/XOR operation), we followed the *direct sharing* approach [8] and achieved:

$$g_{i:j}^1 = g_i^2 \oplus g_j^2 \wedge p_i^2 \oplus g_j^2 \wedge p_i^3 \oplus g_j^3 \wedge p_i^2 \tag{12}$$

$$g_{i:j}^2 = g_i^3 \oplus g_j^3 \wedge p_i^3 \oplus g_j^1 \wedge p_i^3 \oplus g_j^3 \wedge p_i^1 \tag{13}$$

$$g_{i:j}^3 = g_i^1 \oplus g_j^1 \wedge p_i^1 \oplus g_j^1 \wedge p_i^2 \oplus g_j^2 \wedge p_i^1 \ . \tag{14}$$

The other part (computation of $p$) of Equation (4) can be implemented similar to Equation (9). To reduce the amount of required fresh random bits, we replaced $m_i$ with $g_j^1$. This bit is not used in this equation and can take the role of a mask. Although our construction does not closely follow the assumptions in [8] considering the construction of virtual shares, we can demonstrate that this has no impact on security. Our simulation results show not only the uniformity of shared $p_{i:j}$ but also the uniformity of the shared tuple $(p_{i:j}, g_{i:j})$. We need to emphasize that due to the specific architecture of the KSA algorithm, $g_j^1$ is only used once as a mask to introduce uniformity into the computation of a $p$. In other words, the mask bit $g_j^1$ is never reused again what could potentially violate the uniformity in later stages.

Our design is optimized for maximum throughput by using a fully pipelined architecture. Figure 3 depicts the basic structure of our design. Since only the preprocessing step requires fresh random bits and – as stated above – all other stages are computed without additional mask, the total number of required fresh mask bits is $n = 32$. Compared to the other solutions like [18], [10] and [11] this is still reasonable.

### 3.4 Kogge-Stone Adder (Second-Order SCA-Resistant)

Similar as for the RCA, the design based on the KSA is also easily extensible to higher orders. We outline the exemplary procedure for second-order security. In this case, the number of input shares is again set to 5. The four aforementioned equations are adjusted to meet the requirements of the second-order TI.

The XOR part of Equation (3) is implemented as before but adapted to the higher number of shares. The AND part to compute $g_i$ of Equation (3) is split into two steps. As before, the first step results in 10 output shares and the second step merges the last 6 shares into a total number of 5 shares again. Furthermore, we have to use fresh masks to assure the uniformity. In this case,

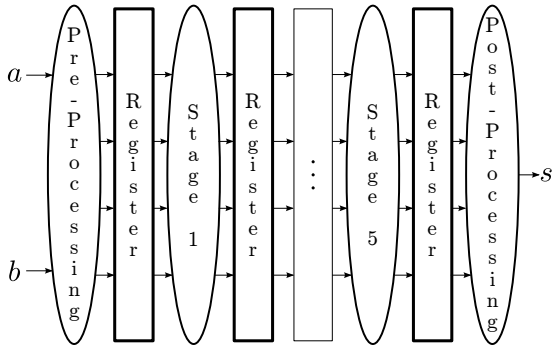



**Fig. 3.** Block diagram of the first-order secure adder based on KSA

four fresh random bits are necessary. The two functions of the following stages are also split into two steps. For the computation of $g_{i:j}$ of Equation (4) we use the second-order TI representation of the AND/XOR function given in [7]. For the $p_{i:j}$ part (the AND operation) we use the same construction of $g_i$ of the preprocessing step. Instead of four fresh mask bits, we used 4 shares of $g_j$ as fresh masks to reduce the required randomness. Details of the underlying uniform equations can be found in the appendix.

The basic structure as shown in Figure 3 is also the template for the architecture of most other parts. It has mainly to be adapted to 5 shares and the functions need to be split into two steps with a register in between. Hence the number of clock cycles for one addition is doubled. In terms of randomness the demand of our implementation quadruples, because each invocation of the AND operation in the preprocessing step requires 4 random bits.

### 3.5 Comparison

We now compare our designs in terms of size and performance that are implemented on a Spartan-6 FPGA with other solutions. All our findings are summarized in Table 1. In terms of size, the RCA-based variant is clearly superior to other solutions due to the iterative structure. On the contrary, the designs based on the KSA provide low latency and high-performance applications.

Due to the different implementation platforms, we cannot fairly compare our hardware designs with software-based solutions [18], [10] and [11]. Therefore, Table 1 restricts the comparison to the number of required fresh random bits.

We can conclude that the RCA-based design is most efficient regarding the number of required random bits. The requirement of $4{\cdot}d$ random bits outperforms all other proposals and is also independent of the operands bit length $n$. The approach based on KSA requires a higher number of random bits which also depends on $n$. Nevertheless, the first-order secure design uses the same amount of fresh masks as the solution of [18] and less than [11]. For higher orders it even outperforms the design of [10]. It is noteworthy that the number of fresh

**Table 1.** Results and comparison of our hardware architectures

| Design | LUTs | FFs | Latency (CLK) | Frequency (MHz) | Throughput (Mbit/s) | Randomness (bit) |
|---|---|---|---|---|---|---|
| RCA 1st order | 227 | 223 | 32 | 101 | 101 | 4 |
| RCA 2nd order | 388 | 387 | 65 | 107 | 52 | $8{=}4 \cdot d$ |
| KSA 1st order | 937 | 1330 | 6 | 62 | 330 | 32 |
| KSA 2nd order | 4223 | 5509 | 12 | 63 | 168 | $128{=}2 \cdot d \cdot n$ |
| [18] (1st order) | - | - | - | - | - | $n$ |
| [10] ($d$ order) | - | - | - | - | - | $(2 \cdot d^2 + d) \cdot n$ |
| [11] (1st order) | - | - | - | - | - | $3 \cdot n$ |

masks for $d$-order KSA with $d \geq 2$ can be decreased even further. For $d = 1$ we can use the trick presented in [8] that requires only one fresh mask bit for an AND operation. Such a construction – with one virtual variable – might be also found for higher-order TI of the AND operation thereby reducing the number of required fresh mask bits.

## 4 Analysis

For the practical SCA evaluations we employed a SAKURA-G platform [1] populated with a Spartan-6 FPGA as target. All SCA traces have been collected by a digital oscilloscope while measuring the voltage drop over a $1\,\Omega$ resistor in Vdd path. In order to obtain clean signals and reduce the electrical noise, we used the embedded amplifier of the SAKURA-G and restricted the bandwidth of the oscilloscope to 20 MHz.

As evaluation metric we applied a *non-specific* statistical $t$-test [16]. The feature of this test is to indicate the existence of any leakage at a defined order in the power traces. Following the concept of non-specific $t$-test, power traces corresponding to fixed and randomly selected inputs are collected. Hence this scheme is also denoted as fixed vs. random $t$-test. During the measurements the fixed and random inputs need to be randomly interleaved. Then, the traces $T$ are categorized into two groups $G_1$ and $G_2$ with respect to the fixed and random inputs, respectively. In the following explanation, we consider only one point of the collected traces for the sake of simplicity:

Recall that Welch's (two-tailed) $t$-test is computed as

$$t = \frac{\mu(T \in G_1) - \mu(T \in G_2)}{\sqrt{\frac{\delta^2(T \in G_1)}{|G_1|} + \frac{\delta^2(T \in G_2)}{|G_2|}}},$$

where $\mu$ and $\delta^2$ denote the sample mean and sample variance respectively, and $|.|$ represents the cardinality. The $t$-test indeed examines the validity of the *null hypothesis* as the samples in both groups were drawn from the same population.

If the null hypothesis is correct, it can be concluded with a high level of confidence that the device under test does not have any first-order leakage, given the recorded traces.

For such a conclusion the Student's $t$-distribution function (in addition to the degree of freedom) is applied to determine the probability of rejecting the aforementioned hypothesis (cf. [16] and [7]). For typical evaluations, a threshold for $|t|$ as $> 4.5$ is defined to reject the null hypothesis and indicate that a first-order attack is feasible. This process is repeated at each sample point independently to obtain a curve of $t$ value.

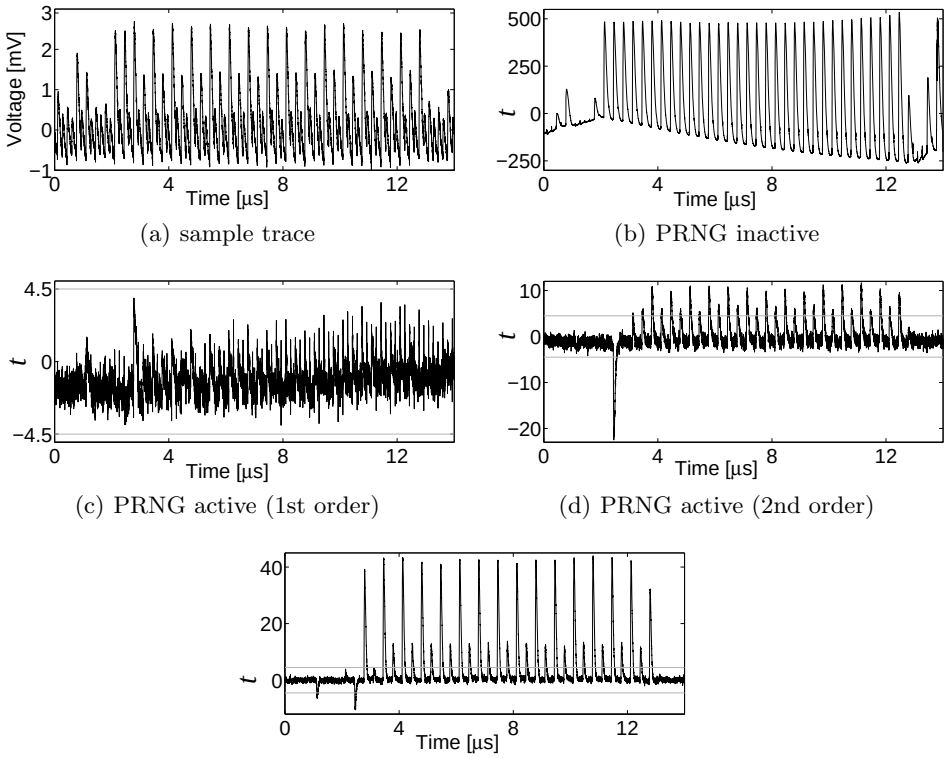
The aforementioned scheme can be easily extended to higher orders by preprocessing the traces as, for example, centered square (for the second-order), standardized cube (for the third order), etc. It is noteworthy that the same evaluation scheme has been applied in [7] and [20] to investigate the existence of first- and higher-order leakages. For detailed information on how to conduct the tests at higher orders the interested reader is referred to [33].

### 4.1 Ripple-Carry Adder

Now we analyze the security of our first-order SCA-resistant RCA design. A sample trace of such a design is shown in Figure 4(a). In order to have a reference for the existing leakage in our platform as well as an evidence for the suitability of the applied evaluation scheme, we first turned the PRNG off that provides the randomness for initial sharing and fresh masks. Hence all outputs of the PRNG are set to zero and the underlying design receives unshared inputs as $(a, 0, 0)$ and $(b, 0, 0)$. With such a setting we collected 100 000 traces corresponding to a mixture of fixed and random inputs. Therefore, we expect the $t$-test to report clearly exploitable first-order leakages, which is confirmed by the corresponding result shown in Figure 4(b). It can be seen that the $t$ value exceeds 400 during the cryptographic operation exceeding the defined threshold by far.

As the next step we activated the PRNG so that the adder circuit receives randomly shared inputs and fresh random masks. Hence the design is expected to provide first-order security. In order to examine this we collected 100 000 000 traces and performed the $t$-test up to third order. The corresponding results shown in Figure 4 indicate the resistance of the design to first-order attacks and – as expected – its vulnerability to second- and third-order attacks.

We continued our evaluation with the second-order-SCA-resistant RCA design with an active PRNG. Due to the high amount of randomness, i.e., fourth-order Boolean masking (five shares), exploiting a leakage from such a design needs a large number of traces. Therefore, following the above-explained procedure we collected 300 000 000 traces and ran the $t$-test evaluations. The results shown in Figure 5 confirm the resistance of our design to first- and second-order attacks. Similar to the results of [7], the third-order leakage still cannot be detected, but we observe fourth- and fifth-order leakages as the design with five shares is expected to be vulnerable to a fifth-order attack.




(a) sample trace

(b) PRNG inactive

(c) PRNG active (1st order)

(d) PRNG active (2nd order)

(e) PRNG active (3rd order)

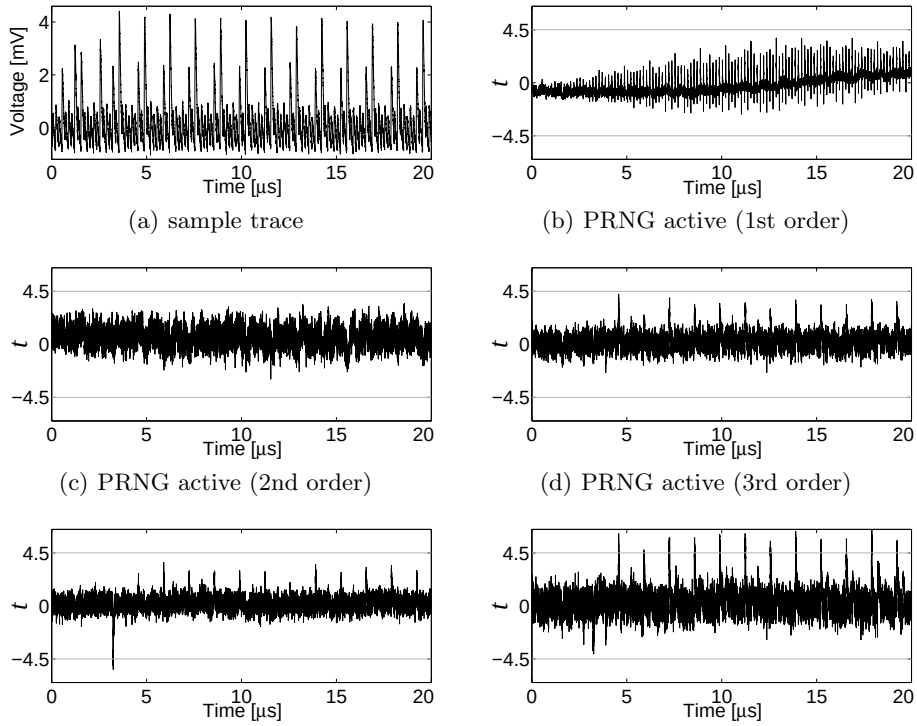
**Fig. 4.** RCA 1st order, $t$-test results using 100 000 000 traces

### 4.2 Kogge-Stone Adder

Both analyses on the first- and second-order RCA were repeated on the first- and second-order SCA-resistant KSA designs. We even collected the same number of traces, i.e., 100 000 000 traces to evaluate the 1st-order KSA and 300 000 000 traces for the second-order KSA. The results which confirm the resistance of our constructions are shown in Figure 6 and Figure 7, respectively.

### 4.3 Higher-Order Security

Recently, Reparaz published a note [32] on the security of higher-order threshold implementations. It states that when different intermediates values, i.e., shares, from different clock cycles are combined, a second-order TI might be vulnerable to the corresponding second-order attack. Although confirming this statement in general, we like to emphasize that this is not addressed in [7]. The idea behind higher-order TI is to resist against *univariate* higher-order attacks where the leakage of different points (of different clock cycles) are not combined. Hence,

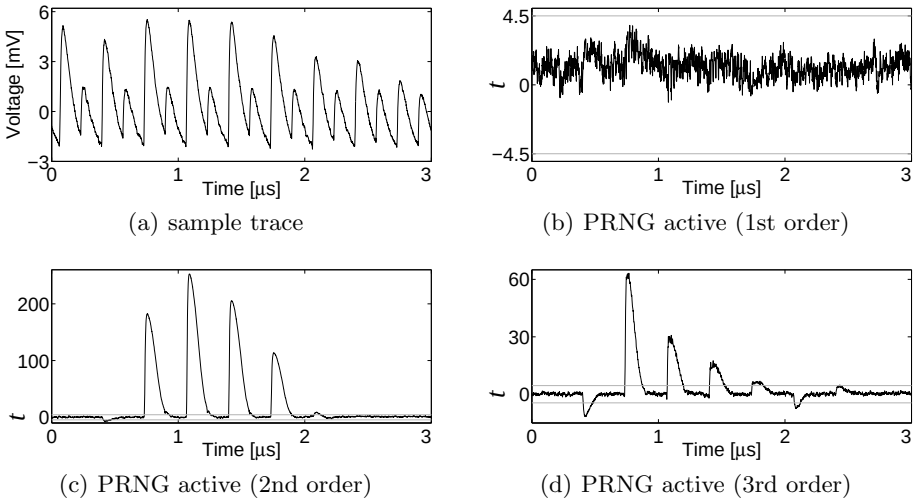



(a) sample trace (b) PRNG active (1st order)

(c) PRNG active (2nd order) (d) PRNG active (3rd order)

(e) PRNG active (4th order) (f) PRNG active (5th order)

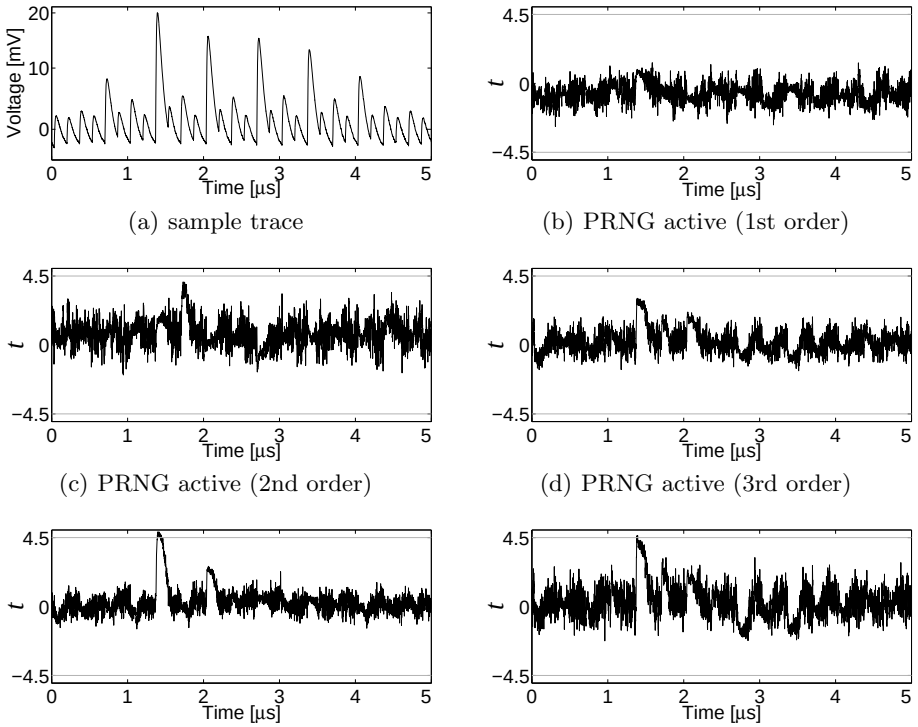
**Fig. 5.** RCA 2nd order, $t$-test results using 300 000 000 traces




(a) sample trace (b) PRNG active (1st order)

(c) PRNG active (2nd order) (d) PRNG active (3rd order)

**Fig. 6.** KSA 1st order, $t$-test results using 100 000 000 traces




(a) sample trace (b) PRNG active (1st order)

(c) PRNG active (2nd order) (d) PRNG active (3rd order)

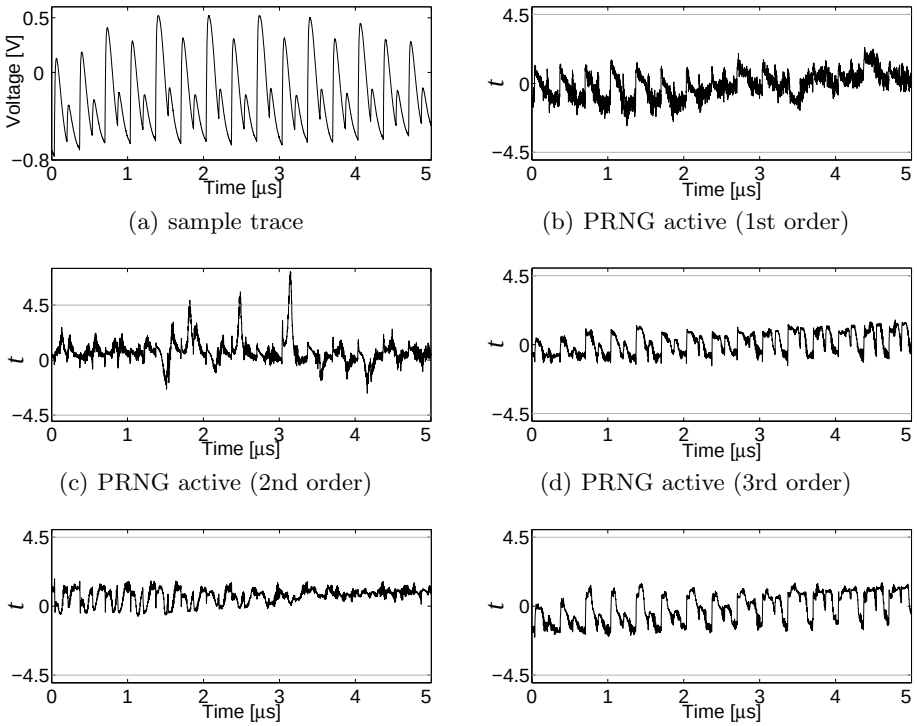
(e) PRNG active (4th order) (f) PRNG active (5th order)

**Fig. 7.** KSA 2nd order, $t$-test results using 300 000 000 traces

in the model of univariate higher-order attacks, all lemmas and proofs as given in [7] remain valid. Furthermore, this is backed by our practical investigations as shown above. Still we need to highlight that the second-order TI designs we presented in this work are designed to resist against univariate second-order attacks.

In this context, it has been previously shown in [24] that multivariate leakages can be easily summed up and be represented in a univariate form. The suggested approaches for such a combination include (a) running the target device at a relatively high frequency, e.g., 24 MHz - 48 MHz, and (b) making use of a DC blocker and/or certain amplifiers in the measurement setup. Both techniques cause overlapping the power peaks of adjacent clock cycles, and hence the leakage associated to consecutive clock cycles are somehow added together. In [24] it has been shown that employing any of the aforementioned techniques causes an implementation of a univariate second-order resistant design to be vulnerable to a univariate second-order attack.

In order to examine the effect of such an issue on our second-order TI designs, we considered the second aforementioned technique. In other words, we employed a DC blocker (BLK-89-S+ from Mini-Circuits) and two serially conected




(a) sample trace (b) PRNG active (1st order)

(c) PRNG active (2nd order) (d) PRNG active (3rd order)

(e) PRNG active (4th order) (f) PRNG active (5th order)

**Fig. 8.** (modified measurement setup) KSA 2nd order, $t$-test results using 300 000 000 traces

AC amplifiers (ZFL-1000LN+ from Mini-Circuits) in the measurement setup. By means of this setup we repeated the same measurements and evaluations of our developed second-order Kogge-Stone Adder using the same number of 300 000 000 traces. We kept the measurement settings, e.g., sampling rate, bandwidth, and the target frequency of operation, the same as the last experiments. The results shown in Figure 8 indeed practically confirm the note given in [32]. The second-order TI design demonstrates second-order leakages when the power peak of consecutive clock cycles are combined (by the measurement setup). Interestingly, by such a measurement setup the 4th-order and 5th-order analyses (in contrary to the previous experiment of Figure 7) do not show a detectable leakage. We believe that it is due to the noise introduced by the measurement setup, i.e., overlapping the adjacent power peaks, which can certainly affect the feasibility of higher-order attacks.

## 5 Conclusion

In this paper, we presented two ways of performing addition on Boolean masked values that are secure against SCA attacks on a hardware platform. Compared to the KSA-based approach, the RCA-based solution is slower but requires less space and the least amount of random bits. In terms of performance, the design based on the KSA provides a suitable choice due to its pipelined architecture. In comparison to other already published algorithms, our approaches are able to match and even reduce the randomness requirements especially for higher orders. The resistance of both approaches has been verified by practical evaluations showing the security of our constructions. Our proposed designs enable an efficient and secure implementation of ARX-based designs in hardware which have not been fully investigated yet.

### Acknowledgment

The authors would like to thank Begül Bilgin from Katholieke Universiteit Leuven, Dept. ESAT/SCD-COSIC and Iminds (Belgium) for her helpful discussions and comments. The research in this work was supported in part by the DFG Research Training Group GRK 1817/1.

## References

1. Side-channel AttacK User Reference Architecture. `http://satoh.cs.uec.ac.jp/SAKURA/index.html`.
2. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. *Submission to NIST*, 2008.
3. O. Benoît and T. Peyrin. Side-Channel Analysis of Six SHA-3 Candidates. In *CHES 2010*, volume 6225 of *LNCS*, pages 140–157. Springer, 2010.
4. D. J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, 2008.
5. D. J. Bernstein. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
6. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A More Efficient AES Threshold Implementation. In *AFRICACRYPT 2014*, volume 8469 of *LNCS*, pages 267–284. Springer, 2014.
7. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In *ASIACRYPT 2014*, LNCS. Springer, 2014. to appear.
8. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold Implementations of All 3 $\times$3 and 4 $\times$4 S-Boxes. In *CHES 2012*, volume 7428 of *LNCS*, pages 76–91. Springer, 2012.
9. C. Boura, S. Lévêque, and D. Vigilant. Side-Channel Analysis of Grøstl and Skein. In *Symposium on Security and Privacy Workshops 2012*, pages 16–26. IEEE Computer Society, 2012.
10. J. Coron, J. Großschädl, and P. K. Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, 2014.

11. J. Coron, J. Großschädl, P. K. Vadnala, and M. Tibouchi. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. Cryptology ePrint Archive, Report 2014/891, 2014. http://eprint.iacr.org/.
12. B. Debraize. Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking. In *CHES 2012*, volume 7428 of *LNCS*, pages 107–121. Springer, 2012.
13. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family. *http://www.skein-hash.info/sites/default/files/skein1.3.pdf*, 2010.
14. B. Gierlichs, L. Batina, C. Clavier, T. Eisenbarth, A. Gouget, H. Handschuh, T. Kasper, K. Lemke-Rust, S. Mangard, A. Moradi, and E. Oswald. Susceptibility of eSTREAM Candidates Towards Side-Channel Analysis. *Proceedings of SASC*, pages 123–150, 2008.
15. J. D. Golic. Techniques for Random Masking in Hardware. *IEEE Trans. on Circuits and Systems*, 54-I(2):291–300, 2007.
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. http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf.
17. L. Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In *CHES 2001*, volume 2162 of *LNCS*, pages 3–15. Springer, 2001.
18. M. Karroumi, B. Richard, and M. Joye. Addition with Blinded Operands. In *COSADE 2014*, volume 8622 of *LNCS*, pages 41–55. Springer, 2014.
19. P. M. Kogge and H. S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Trans. Comput.*, 22(8):786–793, 1973.
20. A. J. Leiserson, M. E. Marson, and M. A. Wachs. Gate-Level Masking under a Path-Based Leakage Metric. In *CHES 2014*, volume 8731 of *LNCS*, pages 580–597. Springer, 2014.
21. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.
22. S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, 2005.
23. S. Miyaguchi. The FEAL Cipher Family. In *CRYPTO 90*, volume 537 of *LNCS*, pages 627–638. Springer, 1991.
24. A. Moradi and O. Mischke. On the Simplicity of Converting Leakages from Multivariate to Univariate - (Case Study of a Glitch-Resistant Masking Scheme). In *CHES 2013*, volume 8086 of *LNCS*, pages 1–20. Springer, 2013.
25. A. Moradi, O. Mischke, and T. Eisenbarth. Correlation-Enhanced Power Analysis Collision Attack. In *CHES 2010*, volume 6225 of *LNCS*, pages 125–139. Springer, 2010.
26. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.
27. S. Nikova, C. Rechberger, and V. Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS 2006*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.
28. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. In *ICISC 2008*, volume 5461 of *LNCS*, pages 218–234. Springer, 2009.

29. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.
30. NIST. Secure Hash Standard (SHS) (FIPS PUB 180-4), 2012.
31. A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling. Side-Channel Resistant Crypto for Less than 2,300 GE. *J. Cryptology*, 24(2):322–345, 2011.
32. O. Reparaz. A note on the security of Higher-Order Threshold Implementations. Cryptology ePrint Archive, Report 2015/001, 2015. `http://eprint.iacr.org/`.
33. T. Schneider and A. Moradi. Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. Cryptology ePrint Archive, Report 2015/207, 2015. `http://eprint.iacr.org/`.
34. D. J. Wheeler and R. M. Needham. TEA, a Tiny Encryption Algorithm. In *FSE 94*, volume 1008 of *LNCS*, pages 363–366. Springer, 1995.
35. H. Wu. The Stream Cipher HC-128. In *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 39–47. Springer, 2008.

# A Second-Order RCA

## A.1 Carry (1. Step)

$$\widetilde{c}^1_{i+1} = a^2_i \wedge b^2_i \oplus a^1_i \wedge b^2_i \oplus a^2_i \wedge b^1_i \oplus a^2_i \wedge c^2_i \oplus a^1_i \wedge c^2_i \oplus a^2_i \wedge c^1_i \oplus c^2_i \wedge b^2_i \oplus c^1_i \wedge b^2_i \oplus c^2_i \wedge b^1_i \quad (15)$$

$$\widetilde{c}^2_{i+1} = a^3_i \wedge b^3_i \oplus a^1_i \wedge b^3_i \oplus a^3_i \wedge b^1_i \oplus a^3_i \wedge c^3_i \oplus a^1_i \wedge c^3_i \oplus a^3_i \wedge c^1_i \oplus c^3_i \wedge b^3_i \oplus c^1_i \wedge b^3_i \oplus c^3_i \wedge b^1_i \quad (16)$$

$$\widetilde{c}^3_{i+1} = a^4_i \wedge b^4_i \oplus a^1_i \wedge b^4_i \oplus a^4_i \wedge b^1_i \oplus a^4_i \wedge c^4_i \oplus a^1_i \wedge c^4_i \oplus a^4_i \wedge c^1_i \oplus c^4_i \wedge b^4_i \oplus c^1_i \wedge b^4_i \oplus c^4_i \wedge b^1_i \quad (17)$$

$$\widetilde{c}^4_{i+1} = a^1_i \wedge b^1_i \oplus a^1_i \wedge b^5_i \oplus a^5_i \wedge b^1_i \oplus a^1_i \wedge c^1_i \oplus a^1_i \wedge c^5_i \oplus a^5_i \wedge c^1_i \oplus c^1_i \wedge b^1_i \oplus c^1_i \wedge b^5_i \oplus c^5_i \wedge b^1_i \quad (18)$$

$$\widetilde{c}^5_{i+1} = a^2_i \wedge b^3_i \oplus a^3_i \wedge b^2_i \oplus a^2_i \wedge c^3_i \oplus a^3_i \wedge c^2_i \oplus c^2_i \wedge b^3_i \oplus c^3_i \wedge b^2_i \quad (19)$$

$$\widetilde{c}^6_{i+1} = a^2_i \wedge b^4_i \oplus a^4_i \wedge b^2_i \oplus a^2_i \wedge c^4_i \oplus a^4_i \wedge c^2_i \oplus c^2_i \wedge b^4_i \oplus c^4_i \wedge b^2_i \quad (20)$$

$$\widetilde{c}^7_{i+1} = a^5_i \wedge b^5_i \oplus a^2_i \wedge b^5_i \oplus a^5_i \wedge b^2_i \oplus a^5_i \wedge c^5_i \oplus a^2_i \wedge c^5_i \oplus a^5_i \wedge c^2_i \oplus c^5_i \wedge b^5_i \oplus c^2_i \wedge b^5_i \oplus c^5_i \wedge b^2_i \quad (21)$$

$$\widetilde{c}^8_{i+1} = a^3_i \wedge b^4_i \oplus a^4_i \wedge b^3_i \oplus a^3_i \wedge c^4_i \oplus a^4_i \wedge c^3_i \oplus c^3_i \wedge b^4_i \oplus c^4_i \wedge b^3_i \quad (22)$$

$$\widetilde{c}^9_{i+1} = a^3_i \wedge b^5_i \oplus a^5_i \wedge b^3_i \oplus a^3_i \wedge c^5_i \oplus a^5_i \wedge c^3_i \oplus c^3_i \wedge b^5_i \oplus c^5_i \wedge b^3_i \quad (23)$$

$$\widetilde{c}^{10}_{i+1} = a^4_i \wedge b^5_i \oplus a^5_i \wedge b^4_i \oplus a^4_i \wedge c^5_i \oplus a^5_i \wedge c^4_i \oplus c^4_i \wedge b^5_i \oplus c^5_i \wedge b^4_i \quad (24)$$

## A.2 Carry (2. Step)

$$c^1_{i+1} = \widetilde{c}^1_{i+1} \quad (25)$$

$$c^2_{i+1} = \widetilde{c}^2_{i+1} \quad (26)$$

$$c^3_{i+1} = \widetilde{c}^3_{i+1} \quad (27)$$

$$c^4_{i+1} = \widetilde{c}^4_{i+1} \quad (28)$$

$$c^5_{i+1} = \widetilde{c}^5_{i+1} \oplus \widetilde{c}^6_{i+1} \oplus \widetilde{c}^7_{i+1} \oplus \widetilde{c}^8_{i+1} \oplus \widetilde{c}^9_{i+1} \oplus \widetilde{c}^{10}_{i+1} \quad (29)$$

# B Second-Order KSA

## B.1 AND (1. Step)

$$\widetilde{g}_i^1 = a_i^2 \wedge b_i^2 \oplus a_i^1 \wedge b_i^2 \oplus a_i^2 \wedge b_i^1 \oplus m_i^1 \tag{30}$$
$$\widetilde{g}_i^2 = a_i^3 \wedge b_i^3 \oplus a_i^1 \wedge b_i^3 \oplus a_i^3 \wedge b_i^1 \oplus m_i^2 \tag{31}$$
$$\widetilde{g}_i^3 = a_i^4 \wedge b_i^4 \oplus a_i^1 \wedge b_i^4 \oplus a_i^4 \wedge b_i^1 \oplus m_i^3 \tag{32}$$
$$\widetilde{g}_i^4 = a_i^1 \wedge b_i^1 \oplus a_i^1 \wedge b_i^5 \oplus a_i^5 \wedge b_i^1 \oplus m_i^4 \tag{33}$$
$$\widetilde{g}_i^5 = a_i^2 \wedge b_i^3 \oplus a_i^3 \wedge b_i^2 \tag{34}$$
$$\widetilde{g}_i^6 = a_i^2 \wedge b_i^4 \oplus a_i^4 \wedge b_i^2 \oplus m_i^1 \tag{35}$$
$$\widetilde{g}_i^7 = a_i^5 \wedge b_i^5 \oplus a_i^2 \wedge b_i^5 \oplus a_i^5 \wedge b_i^2 \tag{36}$$
$$\widetilde{g}_i^8 = a_i^3 \wedge b_i^4 \oplus a_i^4 \wedge b_i^3 \oplus m_i^2 \tag{37}$$
$$\widetilde{g}_i^9 = a_i^3 \wedge b_i^5 \oplus a_i^5 \wedge b_i^3 \oplus m_i^3 \tag{38}$$
$$\widetilde{g}_i^{10} = a_i^4 \wedge b_i^5 \oplus a_i^5 \wedge b_i^4 \oplus m_i^4 \tag{39}$$

## B.2 AND/XOR (1. Step)

$$\widetilde{g}_{i:j}^1 = g_i^2 \oplus g_j^2 \wedge p_i^2 \oplus g_j^1 \wedge p_i^2 \oplus g_j^2 \wedge p_i^1 \tag{40}$$
$$\widetilde{g}_{i:j}^2 = g_i^3 \oplus g_j^3 \wedge p_i^3 \oplus g_j^1 \wedge p_i^3 \oplus g_j^3 \wedge p_i^1 \tag{41}$$
$$\widetilde{g}_{i:j}^3 = g_i^4 \oplus g_j^4 \wedge p_i^4 \oplus g_j^1 \wedge p_i^4 \oplus g_j^4 \wedge p_i^1 \tag{42}$$
$$\widetilde{g}_{i:j}^4 = g_i^1 \oplus g_j^1 \wedge p_i^1 \oplus g_j^1 \wedge p_i^5 \oplus g_j^5 \wedge p_i^1 \tag{43}$$
$$\widetilde{g}_{i:j}^5 = g_j^2 \wedge p_i^3 \oplus g_j^3 \wedge p_i^2 \tag{44}$$
$$\widetilde{g}_{i:j}^6 = g_j^2 \wedge p_i^4 \oplus g_j^4 \wedge p_i^2 \tag{45}$$
$$\widetilde{g}_{i:j}^7 = g_i^5 \oplus g_j^5 \wedge p_i^5 \oplus g_j^2 \wedge p_i^5 \oplus g_j^5 \wedge p_i^2 \tag{46}$$
$$\widetilde{g}_{i:j}^8 = g_j^3 \wedge p_i^4 \oplus g_j^4 \wedge p_i^3 \tag{47}$$
$$\widetilde{g}_{i:j}^9 = g_j^3 \wedge p_i^5 \oplus g_j^5 \wedge p_i^3 \tag{48}$$
$$\widetilde{g}_{i:j}^{10} = g_j^4 \wedge p_i^5 \oplus g_j^5 \wedge p_i^4 \tag{49}$$