A White-Box Masking Scheme Resisting Computational and Algebraic Attacks

Okan Seker, Thomas Eisenbarth, and Maciej Liskiewicz

University of Lübeck, Germany
{okan.seker, thomas.eisenbarth}@uni-luebeck.de
liskiewi@tcs.uni-luebeck.de

Abstract. White-box cryptography attempts to protect cryptographic secrets in pure software implementations. Due to its high utility, white-box cryptosystems (WBC) are deployed even though their secure construction is not well understood. A major breakthrough in generic cryptanalysis of WBC was Differential Computation Analysis (DCA), which requires minimal knowledge of the underlying white-box protection and also thwarts many obfuscation methods. To avert DCA, classic masking countermeasures originally intended to protect against highly related side channel attacks have been proposed for use in WBC. However, due to the controlled environment of WBCs, new algebraic attacks able to break all classic masking schemes have quickly been found. These algebraic DCA attacks break classic masking countermeasures efficiently, as they are independent of the masking order.

In this work, we propose a novel generic masking scheme that can resist both DCA and algebraic attacks. The proposed scheme extends the seminal work by Ishai et al. which is probing secure and thus resists DCA, to also resist algebraic attacks. To prove the security of our scheme, we demonstrate the connection between two main security notions in white-box cryptography: *Side Channel Analysis (SCA) security* and *prediction security*. Resistance of our masking scheme to DCA is proven for an arbitrary order of protection. Our masking scheme also resists algebraic attacks, which we show concretely for first and second order algebraic protection, and show how it can be generalized to any order. Moreover, we present an extensive performance analysis and quantify the overhead of our scheme, for a proof-of-concept protection of an AES implementation.

Keywords: White-box Cryptography · Boolean Masking · Non-linear Masking · Probing Security · Prediction Security · Differential Computation Analysis · Algebraic Attacks

1 Introduction

Protecting secrets purely in software is a great challenge, especially if a full system compromise is not simply declared out-of-scope of the security model. With fully homomorphic encryption still complex and computationally expensive [34] and secure enclaves being notoriously buggy at this time [12, 33, 44], industry may opt for white-box cryptosystems (WBC). White-box cryptography promises implementation security of cryptographic services in pure software solutions, mainly by protecting keys and intermediate cipher states through layers of obfuscation. While white-box cryptography is successfully sold by several companies as one ingredient of secure software solutions (e.g. [21]), analysis of deployed solutions is lacking, as is a sound framework to analyze white-box implementations. The white-box model assumes the cryptographic primitive to run in an untrusted environment where the white-box adversary has full control over the implementation. The adversary has full access to every memory access, can read and modify intermediate states and can interrupt the implementation at will. White-box cryptography was introduced in 2002 by Chow et al. [15, 16]. The main idea of their scheme is to represent a cryptographic algorithm as a network of look-up tables and key-dependent tables. In order to protect the key dependent tables, Chow et al. proposed to use *input and output encodings*. Although the method provides provable security guarantees for individual tables, the combinations of protected tables still leaks information [3]. In fact, all published academic proposals for WBC [11, 27, 31, 47] have been practically broken [3, 19, 30, 46].

Cryptanalysis of WBCs usually requires a time-consuming reverse engineering step to overcome included obfuscation layers [24]. To overcome this, *computational analysis* of white-box cryptosystems have been proposed. Computational analysis is inspired by physical grey box attacks, mainly *side-channel attacks*. Computational analysis attacks, like side channel attacks, perform statistical analysis of observable intermediate states of a cryptographic implementation, e.g. via its physical side channel [20, 22, 28]; if the implementation is not protected against this kind of attack, the side channel may reveal critical information, usually the secret key material used. At CHES 2016, Bos et al. [9] proposed *Differential Computation Analysis* (DCA) and showed that DCA can extract keys from a wide range of different white-box implementations very efficiently, without requiring a detailed reverse engineering of the implementation. Following this work, further generic computational analysis techniques have been proposed for white-box implementations, such as Zero Difference Enumeration [1], Collision Attacks, and Mutual Information Analysis [39]. Bock et al. [6] analyzed the ineffectiveness of internal encodings and explain why DCA works so well in the white-box setting. Even fault attacks [2, 8] have been shown to be an effective method for state and key recovery attacks on white-box implementations [5, 9]. Biryukov et al. [4] introduced two new types of fault attacks to reveal the structure of a white-box implementation, an important step of overcoming obfuscation in WBC.

To overcome the threat of DCA and other computational analysis, a natural protection mechanism are *masking schemes*. Masking splits a sensitive variable x into n shares, such that x can be recovered from $d+1$ ($n \geq d+1$) shares, while no information can be recovered from fewer than $d+1$ shares [14]. It is a popular and effective countermeasure in the SCA literature. Most important examples are *Boolean masking* introduced by Ishai et al. [26] which has been generalized by Rivain and Prouff [37], *Threshold Implementations* defined by Nikova et al. [35], and *polynomial masking* as defined in [40] based on Shamir's secret sharing [43]. And recently the idea of *combined countermeasures* to resist both side-channel and fault attacks are introduced in the literature [36, 41, 42].

Unlike the attacks, countermeasures cannot be applied to white-box implementations directly. For example, a dedicated masked white-box implementation introduced in [29] and it is broken in [39]. In addition, for secure WBC, other countermeasures such as fault protection and obfuscation layers need to be added [4] and additional randomness should be included in the input [7], as internal randomness generators could be disabled by the white-box adversary. Furthermore, higher order variants of DCA have been shown to be effective when applied to masked white-box implementations due to the adversary's ability to observe shares without noise [7]. Although the noise-free environment makes the attack easier, techniques like control flow obfuscation, input/output encodings and shuffling [45] create artificial noise in white-box environments [1, 7], effectively increasing the complexity of higher order DCA significantly. More *devastatingly*, a new class of generic *algebraic DCA* (or in short *algebraic attacks*) has been proposed recently [4, 24]. Algebraic DCA is shown to break masked WBC independently of the masking orders if the masking is linear. Yet all current masking proposals are vulnerable to algebraic DCA.

To sum up, although there exist informal ideas on how to create a secure white-box design that can resist both computational and algebraic DCA, formal and generic constructions with their security analysis are missing.

Our contribution: In this paper, we provide the first generic and combined masking scheme that resists state-of-the-art white-box attacks: DCA and algebraic attacks. Classic masking schemes can be applied to WBC, however none of them can *individually* achieve security against both attacks. To fill this gap, we examine the `ISW` transformation introduced by Ishai et al. [26] and extend it to the white-box context.

As explained earlier, a Boolean masking scheme provides protection against DCA, however it is shown in [4] that they are vulnerable to algebraic attacks, *independently* of the masking order. We improve the `ISW` transformation by replacing a secret share with a multiplicative representation in order to gain security against algebraic attacks. The secret sharing of our masking scheme consists of two components: linear and non-linear shares. The main aim of this separation can be summarized as follows:

1. Linear shares to resist DCA attacks (or *computational attacks*),
2. Non-linear shares to increase the degree of decoding and therefore prevent the algebraic attacks.

Using the generic construction, we give a comprehensive performance analysis and comparison of our scheme. The analysis includes the total number of bitwise operations and randomness requirements of the masking scheme with various degrees of protections.

To analyze the security of our construction, we focus on two security notions in cryptography: *SCA security* and *prediction security* that cover security against computational attacks and security algebraic attacks respectively. The first model that deals with *passive* adversaries at any order is introduced by Ishai et al. [26] and it is called *probing model.* The idea is then revised by Rivain et al. [37]. The new model is called n^{th} SCA security and it states that every tuple of n or less intermediate variables must be independent of any sensitive variable. The attacker can observe any set of intermediate variables with n elements. It is shown that an n^{th}-order Boolean masking provides security against n^{th} order SCA. The complexity of computational attacks grows with the masking order. Moreover, the model is also used in the white-box context. As stated in [7], an n^{th}-order masking provides security against n^{th}-order SCA and n^{th}-order DCA attacks with additional obfuscation layers. However, security in the SCA model is necessary but not *sufficient*, since the SCA model covers only computational attacks (DCA).

Another approach is given by the prediction security model, in which an attacker can observe *every* intermediate variable and can only use a d^{th} order function to combine them. For example, an n^{th}-order Boolean masking that is inherently protected against DCA is vulnerable against first order algebraic attacks since the adversary can utilize a linear function (i.e. a first order function) and combine a subset of intermediate variables to recover the secret value.

In this work, we further show that SCA security and prediction security notions *are incomparable.* The models cover different aspects of white-box leakages and both of them are required to achieve security in the white-box model. However, security in either model can be achieved without achieving the other, resulting in insecure schemes. Therefore, we prove the security of our constructions using both notions. First, we prove that our masking scheme is indeed secure against computational attacks by showing that it is secure in SCA model with the given order. Moreover we prove the first and second order prediction security of our scheme. Besides the formal proof, we update and use the tool given in [4] to experimentally verify the first order prediction security of our scheme. The updated version of the tool is available as open source[1].

Although the masking scheme is generic, the prediction security depends on the structure of the operations. We give a concrete construction for first and second order prediction security and prove their security. Furthermore, the presented methodology can be adapted to arbitrary orders of prediction security.

In the last part of the paper we introduce a proof-of-concept AES implementation to analyze the overhead and experimentally verify the security properties of our scheme using a simple leakage test. The analysis includes the number of needed gates and number of required randomness for different orders of protection. We show that our combined approach outperforms the previous approaches which required to use the combination of two different masking schemes to resist both attacks.

[1] `https://github.com/UzL-ITS/white-box-masking`

Outline of the Paper: Section 2 provides preliminaries. In Section 3, we present the structure of generic masking that resists computational and algebraic attacks for arbitrary orders of protection. In Section 4, we prove the security of our scheme using the notions *SCA security* and *prediction security.* Finally, in Section 5, we propose a proof-of-concept AES-128 implementation with the performance analysis using various security parameters.

2 Preliminaries

In this section, we provide the notation and definitions used in this paper. Also we identify the challenges that need to be addressed for secure white-box designs.

Firstly, we summarize the notation that is needed for the masked white-box design. We denote the Boolean (or linear) masking order by n and multiplicative (or non-linear) masking order as d. The letters $x, y, z, \ldots$ represent the sensitive variables. Random variables are represented by letter r, with an index as r_i or r^i. To denote a random selection of a variable from the field, we use $\in_R$. The subscripts $x_i, y_i, z_i, \ldots$ represent the i^{th} linear share of a variables while $\tilde{x}_i, \tilde{y}_i, \tilde{z}_i, \ldots$ represent the i^{th} non-linear share. A vector of shares $(\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n)$ is denoted by $\overline{x}$. Bold numbers $\mathbf{0}$ and $\mathbf{1}$ are used to denote constant functions.

As usual, we model the white-box implementations as Boolean circuits represented by directed acyclic graphs. Each node in a circuit C, with $k > 0$ inputs, corresponds to a k-ary Boolean function. Nodes with the indegree equal to zero are called inputs of C and nodes with the outdegree equal to zero are called outputs of C.

Let $\mathsf{x} = (\mathsf{x}_1, \ldots, \mathsf{x}_N)$ (resp. $\mathsf{y} = (\mathsf{y}_1, \ldots, \mathsf{y}_M)$) be a vector of input (resp. output) nodes in some fixed order. For each node v in C, we say that it computes a Boolean function $f_v : \mathbb{F}_2^N \to \mathbb{F}_2$ defined as follows:

- for all $1 \leq i \leq N$ set $f_{\mathsf{x}_i}(z) = z_i$,
- for all non-input nodes v in C set $f_v(z) = g(f_{c_1}(z), \ldots, f_{c_k}(z))$, where $c_1, \ldots, c_k$ are nodes having an outgoing edge to v.

The set of f_v for all nodes v in C is denoted $\mathcal{F}(C)$ and the set of f_{x_i} for all input nodes x_i is denoted $\mathcal{X}(C)$.

Differential Computational Analysis: The idea of using *side-channel attacks* to recover critical secrets in WBC has been introduced by Bos et al. [9]. Differential computational analysis utilizes internal states of the software execution (such as memory accesses) to generate software traces. DCA is regarded as one of the most efficient attacks against white-box implementations, since it does not require full knowledge of white-box design and thus makes the time-consuming reverse engineering process avoidable. The first part of DCA consists of collecting software traces using memory addresses, intermediate values or written/read values by the implementation. In the second part a statistical analysis is performed using the software traces collected in the first part.

To resist against DCA, a natural approach is to use the well-known side-channel analysis countermeasure *masking* [14]. The masking is carried out in two steps as defined in the seminal work by Ishai, Sahai, and Wagner in 2003 [26]. First, input data is transformed by representing each input x by $n+1$ shares in such a way that

$$x = x_0 \oplus \cdots \oplus x_n,$$

where $x \in \mathbb{F}_2$ and n of the shares are distributed uniformly and independently. Additionally, the circuit is adapted by replacing all AND and XOR gates with gadgets processing the shares of the inputs. Throughout the paper, the two stages of masking will be defined as `ISW` transformation.

Masking schemes rely on the availability of good randomness, which is usually provided by secure RNGs, e.g. in the form of a secure and efficient Pseudorandom Generator [18, 25]. Similarly, randomness generation for white-box implementations has been analyzed in the literature. Due to the adversarial ability to control the execution environment in white-box model, the attacker can simply disable an external randomness source. Therefore, white-box implementations have to rely on internal randomness sources in combination with additional obfuscation countermeasures [1, 4, 7]. Remark that the effectiveness of DCA comes from its universality and its ability to avoid reverse-engineering, which can be extremely costly [24]. By combining masking with an obfuscation layer, the adversary is this forced to invest on a time-consuming reverse engineering step to bypass the obfuscation which cannot be done by an automated tool, while the masking prevents obfuscation-oblivious attacks such as DCA.

Algebraic Attacks: Algebraic attacks have been introduced during the WhibOx contest of CHES2017 [17]. Although the majority of the implementations in the contest were broken in less than one day, the strongest design (by means of the surviving time: 28 days) was broken by algebraic analysis [4, 24]. Algebraic attacks try to find a set of circuit nodes whose d^{th}-order of combination equals to a predictable vector. Observe that if an implementation is protected by a linear masking, there exists a set of circuit nodes (corresponding to the secret shares) such that the linear combination (i.e. the first order combination) is always equal to a predictable secret value. This means that, linear maskings are inherently vulnerable to first-order algebraic attacks *independently of the masking order* [4, 24]. Like DCA, Algebraic attacks do not require complex reverse engineering and are thus a generic threat that any white-box implementation needs to address.

Thus, to thwart both of the above-mentioned generic attacks, secure masking for white-box implementations needs to fulfill the following two requirements:

– The number of shares needs to be sufficiently high to prevent computation attacks (DCA).
– There may be no low degree decoder in order to counteract algebraic attacks.

Another challenge of the secure white-box implementation is the adversaries ability to collect noise-free measurements. Remark that the security of masking schemes against side-channel attacks or DCA comes from the inherently noisy measurements [13]. To deal with this problem, artificial noise sources such as control flow obfuscation [1], shuffling [7] and input and output encodings [6] have been analyzed in the literature. The artificial noise introduced by these methods increases the complexity of higher order DCA dramatically. It has been shown in [7] that the complexity of the attacks increases with the order of the masking and the order of the obfuscation layers. Therefore, the SCA model is a valid approach to analyze the security of masking schmes of white-box implementations against DCA. Due to the artificial noise sources, it becomes infeasible for an attacker to combine the required number of shares to recover the sensitive information. Throughout the paper we assume a reliable randomness source is provided as part of the implementation, that is fed internally and protected by obfuscation layers, as done in [1, 4, 39]. Therefore, the attacks on randomness sources and the adversaries' ability to disable randomness is out-of-scope of this work. For a full white-box implementation, other problems (fault protection, randomness generation, obscurity layers) need to be added [4, 7] in addition to a secure masking scheme, which we introduce throughout this work.

In the next Section, we introduce our masking scheme to resist both computational and algebraic attacks using an adapted version of the `ISW` transformation.

3 Secure Masking Construction

The proposed masking scheme is based on two ideas: an `ISW`-like masking to increase the number of shares required to eliminate the computation attacks and using a multiplicative sharing to increase the degree of the decoding function. We denote the first part as linear sharing of order n and the second part as non-linear sharing of degree d. And the resulting construction is named as (n, d)-masking. The summary and the security properties of the schemes are presented in Table 1.

d \ n	0	1	2	n
0	○	◐ `ISW Transformation` [26]		
1	◑	◑ [4]	● Ex. 3	●$[n, 1]$
2	◑	◑	● Ex. 4	●$[n, 2]$
d	◑	◑	●	● Sec. 3.1

Table 1. The security properties of masking schemes. The mark ○ (resp. ●) means the scheme is vulnerable (resp. resistance against) both to computational and algebraic attacks. Mark ◑ (resp. ◐) for vulnerability to computational but resistance against algebraic attacks (resp. resistance against computational but vulnerability to algebraic attacks). Remark that a masking scheme with $(n, 0)$ is the `ISW` transformation [26] while a masking scheme with $(1, 1)$ is the scheme in [4]. The example structures for the masking schemes with $(2, 1)$ and $(3, 1)$ can be found in Appendix B.

We start with the data transformation and define our masking function:

$$\texttt{Encode}(x, \tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_{n-1}) = (\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n)\,,$$

where $\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_{n-1} \in_R \mathbb{F}_2$ are chosen randomly and independently from $\mathbb{F}_2$, and

$$x_n = x \oplus \prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n-1} x_i\ .$$

Observe that our masking scheme is obtained from `ISW` transformation by replacing the first share x_0 in `ISW` by a non-linear sharing $x_0 = \prod_{j=0}^{d} \tilde{x}_j$. The unmasking function is defined as follows:

$$\texttt{Decode}(\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n) = \prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i.$$

The data transformation is followed by the transformations of each AND and XOR gate. Throughout the paper, we define the transformed gates as `And` and `Xor` (or `And`$[n, d]$ and `Xor`$[n, d]$) gadgets respectively.

3.1 Gate Transformations

In this section the generic constructions for `Xor`, `And` are presented. Additionally, we provide definition of the `RefreshMask` gadget, which is needed to protect against algebraic attacks. The scheme can be used for an arbitrary order n of linear masking and any degree d of the non-linear component. Though the constructions are general, the algebraic security depends on the structure of the nodes (the details can be found in Section 4). The intermediate variables (which we called the bottlenecks) that needs a special structure depending the non-linear degree d are the following:

- The intermediate variable $\mathcal{U}$ used in `Xor` and specified in Equation (1),
- The function $\mathcal{F}$ in Equation (2), used in `And`, outputs the variables $\mathcal{V}$,
- The intermediate variables $\mathcal{W}$ and $\mathcal{R}$ used in `RefreshMask`, Equation (3).

Let x and y be two bits and consider an (n, d)-masking scheme, i.e. x and y have been split into $(n + d + 1)$ shares such that $\prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i = x$ and $\prod_{j=0}^{d} \tilde{y}_j \oplus \bigoplus_{i=1}^{n} y_i = y$. Below, we describe each gadget and simultaneously give a corresponding algorithm in pseudocode and explicit structure of the bottlenecks of the $[n, 1]$ and $[n, 2]$ gadgets.

`Xor`$[n, d]$ Gadget: A masked representation of $z = x \oplus y$ with $n + d + 1$ shares such that $\prod_{j=0}^{d} \tilde{z}_j \oplus \bigoplus_{i=1}^{n} z_i = z$ can be calculated as follows:

Step-0: The input shares processed by `RefreshMask` gadgets;

$$\overline{x} \leftarrow \texttt{RefreshMask}(x) \text{ and } \overline{y} \leftarrow \texttt{RefreshMask}(y).$$

Step-1: The values of the non-linear shares are processed:

$$\tilde{z}_i = \tilde{x}_i \oplus \tilde{y}_i \text{ for } 0 \leq i \leq d.$$

Algorithm 1 $\texttt{Xor}(\overline{x}, \overline{y})$

```
Input: The shares $\overline{x} = ((\tilde{x}_j)_{j\in[0,d]}, (x_i)_{i\in[1,n]})$ and $\overline{y} = ((\tilde{y}_j)_{j\in[0,d]}, (y_i)_{i\in[1,n]})$.
Output: The shares of $x \oplus y$ as $\overline{z} = ((\tilde{z}_j)_{j\in[0,d]}, (z_i)_{i\in[1,n]})$.
1: $\overline{x} \leftarrow \texttt{RefreshMask}(\overline{x})$
2: $\overline{y} \leftarrow \texttt{RefreshMask}(\overline{y})$
3: for $0 \le j \le d$ do
4:     $\tilde{z}_j \leftarrow \tilde{x}_j \oplus \tilde{y}_j$
5: for $1 \le i \le n$ do
6:     $z_i \leftarrow x_i \oplus y_i$
7: $z_n \leftarrow x_n \oplus y_n \oplus \mathcal{U}$
8: return $\bar{z} = ((\tilde{z}_j)_{j\in[0,d]}, (z_i)_{i\in[1,n]})$
```

Step-2: Computation of linear shares are operated:

$$z_i = \begin{cases} x_i \oplus y_i, & \text{for } 1 \le i < n \\ x_i \oplus y_i \oplus \mathcal{U}, & \text{for } i = n. \end{cases}$$

where $\mathcal{U}$ can be defined as follows:

$$\mathcal{U} = \bigoplus_{\substack{I \subsetneq \{0,\ldots,d\} \\ I \ne \emptyset}} \prod_{i\in I} \tilde{x}_i \prod_{j \notin I} \tilde{y}_j \tag{1}$$

As the explicit constructions, we can introduce $\mathcal{U}$ as follows:
- $\texttt{Xor}[n, 1]$: $\mathcal{U} = \tilde{x}_0\tilde{y}_1 \oplus \tilde{x}_1\tilde{y}_0$
- $\texttt{Xor}[n, 2]$: $\mathcal{U} = \tilde{x}_1(\tilde{x}_2\tilde{y}_0 \oplus \tilde{y}_2(\tilde{x}_0 \oplus \tilde{y}_0)) \oplus \tilde{y}_1(\tilde{x}_2\tilde{y}_0 \oplus \tilde{x}_0(\tilde{x}_2 \oplus \tilde{y}_2)$
- $\texttt{Xor}[n, d]$ for $d \ge 3$, $\mathcal{U}$ can be calculated as in Equation (1). However the circuit nodes should be constructed carefully in order not to create vulnerabilities in algebraic security.

$\textit{And}[n, d]$ *Gadget:* A masked representation of $z = xy$ with $n + d + 1$ shares such that $\prod_{j=0}^{d} \tilde{z}_j \oplus \bigoplus_{i=1}^{n} z_i = z$ can be calculated as follows:

Step-0: The input shares processed by `RefreshMask` gadgets;

$$\overline{x} \leftarrow \texttt{RefreshMask}(x) \text{ and } \overline{y} \leftarrow \texttt{RefreshMask}(y).$$

Step-1: The calculations of the values of multiplicative representation are processed. Additional random bits $r^{i,j}$ are generated in order to attain algebraic security in the second step.

$$\tilde{z}_i = \tilde{x}_i\tilde{y}_{i'} \oplus r^{i,1} \oplus \cdots \oplus r^{i,n} \text{ for } 0 \le i \le d \text{ where } i' = i + 1 \bmod (d + 1).$$

Step-2: The variables $r_{j,i}$ for $0 \le i < j \le n$ are generated as follows:

$$r_{j,i} = \begin{cases} (r_{i,j} \oplus (\tilde{x}_0 \cdots \tilde{x}_d)y_j) \oplus x_j(\tilde{y}_0 \cdots \tilde{y}_d), & \text{for } i = 0 \quad \textbf{(a)} \\ (r_{i,j} \oplus x_iy_j) \oplus x_jy_i, & \text{for } 1 \le i \le n \text{ where } r_{i,j} \in_R \mathbb{F}_2 \quad \textbf{(b)} \end{cases},$$

The calculations for $1 \leq i \leq n$ are processed as identical to the `ISW-And` gadget. However, for $i = 0$ the calculations require special attention and we need to define a function $\mathcal{F}$ as follows:

$$r_{j,0} = \mathcal{F}(x_j, y_j) = [r_{0,j} \oplus (\tilde{x}_0 \cdots \tilde{x}_d) y_j] \oplus x_j (\tilde{y}_0 \cdots \tilde{y}_d) \text{ for } 1 \leq j \leq n. \quad (2)$$

Unlike to the **Step-2(a)**, $r_{0,j}$ cannot be assigned as random. Instead, $r_{0,j}$ should be defined in such a way that the following equation holds:

$$\bigoplus_{j=1}^{n} r_{0,j} = \bigoplus_{\substack{I \subset \{0,\ldots,d\} \\ I \neq \emptyset}} \prod_{i \in I} \tilde{x}_i \tilde{y}_{i'} \prod_{j \notin I} (r^{j,1} \oplus \cdots \oplus r^{j,n}) \text{ where } i' = i{+}1 \bmod (d{+}1).$$

Throughout the paper we denote right hand side of the above equation as $\mathcal{V}$. Note that the above structure for $\mathcal{F}(x_j, y_j)$ (given on the right hand side of Equation (2)) is not secure against algebraic attack even of the first order. Below we provide secure construction for the case $(n, 1)$ and $(n, 2)$-masking.

– `And`$[n,1]$: $\mathcal{F}(x_j, y_j) = \tilde{x}_1(\tilde{x}_0 y_j \oplus r^{0,j} \tilde{y}_0) \oplus \tilde{y}_1(\tilde{y}_0 x_j \oplus r^{1,j} \tilde{x}_0) \oplus r^{1,j}(r^{0,1} \oplus \ldots \oplus r^{0,n})$.
– `And`$[n,2]$:
$$\begin{aligned}\mathcal{F}(x_j, y_j) = {} & \tilde{x}_0 \left[\tilde{x}_2(\tilde{x}_1 y_j \oplus r^{0,j} \tilde{y}_0) \oplus r^{1,j} v \tilde{y}_1\right] \oplus \\ & \tilde{y}_0 \left[\tilde{y}_1(\tilde{y}_2 x_j \oplus r^{1,j} \tilde{x}_2) \oplus r^{0,j} u \tilde{x}_2\right] \oplus \\ & \tilde{x}_0 \tilde{y}_1 (r^{1,j} \tilde{x}_2 \tilde{y}_0 \oplus r^{2,j} \tilde{x}_1 \tilde{y}_2) \oplus r^{0,j} \tilde{x}_1 \tilde{y}_2 (v \oplus \tilde{x}_2 \tilde{y}_0) \oplus \\ & \tilde{x}_2 \tilde{y}_0 (r^{0,j} \tilde{x}_0 \oplus r^{1,j} \tilde{y}_1) \oplus u v r^{0,j}.\end{aligned}$$
where $u = r^{1,1} \oplus \cdots \oplus r^{1,n}$ and $v = r^{2,1} \oplus \cdots \oplus r^{2,n}$.
– `And`$[n,d]$ for $d \geq 3$ the circuit nodes that calculates $\mathcal{F}(x_j, y_j)$ should be structured in such a way that algebraic security properties are satisfied.

Step-3: The final step can be performed identical to an `ISW-And` gadget: For every $1 \leq i \leq n$, compute $z_i = x_i y_i \oplus \bigoplus_{i \neq j} r_{i,j}$.

`RefreshMask`$[n,d]$ Gadget: The operation has a crucial importance for generating an algebraically secure implementation. In fact, the gadget should be combined with each `Xor` and `And` gadget in order to obtain a fully secure masking scheme. The security details can be found in Section 4.

Step-1: For $0 \leq i \leq d$, calculate $\tilde{x}'_i = \tilde{x}_i \oplus \tilde{r}_i$ where $\tilde{r}_i \in_R \mathbb{F}_2$.
Step-2: For $1 \leq i < n$, calculate $x'_i = x_i \oplus r_i$ and $x_n = x_n \oplus r_i$ where $r_i \in_R \mathbb{F}_2$.
Step-3: In the last step we need to define two intermediate variables as follows:

$$\mathcal{W}' = \bigoplus_{I \subsetneq \{0,\ldots,d\}} \prod_{i \in I} \tilde{x}_i \prod_{j \notin I} \tilde{r}_j \text{ and } \mathcal{W} = \bigoplus_{\substack{I \subsetneq \{0,\ldots,d\} \\ I \neq \emptyset}} \prod_{i \in I} (\tilde{x}_i \oplus r_0) \prod_{j \notin I} \tilde{r}_j,$$

Algorithm 2 And$(\overline{x}, \overline{y})$

```
Input: The shares $\overline{x} = ((\tilde{x}_j)_{j\in[0,d]}, (x_i)_{i\in[1,n]})$ and $\overline{y} = ((\tilde{y}_j)_{j\in[0,d]}, (y_i)_{i\in[1,n]})$.
Output: The vector of shares of $xy$ as $\overline{z} = ((\tilde{z}_j)_{j\in[0,d]}, (z_i)_{i\in[1,n]})$.
 1: $\overline{x} \leftarrow$ RefreshMask$(\overline{x})$
 2: $\overline{y} \leftarrow$ RefreshMask$(\overline{y})$
 3: for $0 \le i \le d$ do
 4:     $\tilde{z}_i = \tilde{x}_i\tilde{y}_{i'}$                                  ▷ $i' = i + 1 \bmod (d + 1)$
 5:     for $1 \le j \le n$ do
 6:         $r^{i,j} \leftarrow$ rand$(0, 1)$
 7:         $\tilde{z}_i = \tilde{z}_i \oplus r^{i,j}$
 8: for $0 \le i \le n$ do
 9:     for $i < j \le n$ do
10:         if $i = 0$ then $r_{j,0} \leftarrow \mathcal{F}(x_i, y_i)$ else
11:         $r_{i,j} \leftarrow$ rand$(0, 1)$
12:         $r_{j,i} \leftarrow (r_{i,j} \oplus x_iy_j) \oplus x_jy_i$
13: for $1 \le i \le n$ do
14:     $z_i \leftarrow z_i \oplus x_iy_i$
15:     for $0 \le j \le n$ and $j \ne i$ do
16:         $z_i \leftarrow z_i \oplus r_{i,j}$
17: return $\overline{z} = ((\tilde{z}_j)_{j\in[0,d]}, (z_i)_{i\in[1,n]})$
```

Here, as usually, a product over the empty set I is evaluated as 1. Using the above equations, we can introduce the variables that need to be added the share x_n as:

$$x'_n \leftarrow x_n \oplus \mathcal{W} \oplus \mathcal{R} \text{ where } \mathcal{W} \oplus \mathcal{R} = \mathcal{W}' \tag{3}$$

Remark that we cannot directly add $\mathcal{W}'$ to the final share x_n due to algebraic security properties. Therefore, the variables $\mathcal{W}$ with $\mathcal{R}$ should be added to the final share in order to define an algebraically secure mask refreshing gadget. The explicit structure of the circuit nodes to calculate $\mathcal{W}$ and $\mathcal{R}$ for `RefreshMask`$[n, 1]$ and `RefreshMask`$[n, 2]$ can be found below.

- `RefreshMask`$[n, 1]$: $\mathcal{W} = \tilde{r}_0(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_1(\tilde{x}_0 \oplus r_0)$ and $\mathcal{R} = (\tilde{r}_0 \oplus r_0)(\tilde{r}_1 \oplus r_0) \oplus r_0$.
- `RefreshMask`$[n, 2]$: $$\begin{aligned}\mathcal{W} = {} & \tilde{r}_1\tilde{r}_2(\tilde{x}_0 \oplus r_0) \oplus \tilde{r}_0\tilde{r}_2(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_0\tilde{r}_1(\tilde{x}_2 \oplus r_0) \oplus \\ & \tilde{r}_2(\tilde{x}_0 \oplus r_0)(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_1(\tilde{x}_0 \oplus r_0)(\tilde{x}_2 \oplus r_0) \oplus \\ & \tilde{r}_0(\tilde{x}_1 \oplus r_0)(\tilde{x}_2 \oplus r_0), \\ \mathcal{R} = {} & (\tilde{r}_0 \oplus r_0)(\tilde{r}_1 \oplus r_0)(\tilde{r}_2 \oplus r_0) \oplus \\ & \tilde{r}_2r_0(\tilde{x}_0 \oplus \tilde{x}_1) \oplus \tilde{r}_1r_0(\tilde{x}_0 \oplus \tilde{x}_2) \oplus \tilde{r}_0r_0(\tilde{x}_1 \oplus \tilde{x}_2).\end{aligned}$$
- `RefreshMask`$[n, d]$ for $d \ge 3$ the circuit nodes that calculates $\mathcal{W}$ and $\mathcal{R}$ should be structured in such a way that algebraic security properties are satisfied.

Algorithm 3 `RefreshMask`($\overline{x}$)

```
Input: The shares $\overline{x} = ((\tilde{x}_j)_{j\in[0,d]}, (x_i)_{i\in[1,n]})$
Output: The shares $\overline{x} = ((\tilde{x}'_j)_{j\in[0,d]}, (x'_i)_{i\in[1,n]})$
 1: for $0 \le j \le d$ do
 2:     $\tilde{r}_j \leftarrow \texttt{rand}(0,1)$
 3:     $\tilde{x}'_j \leftarrow \tilde{x}_j \oplus \tilde{r}_j$
 4: for $1 \le i < n$ do
 5:     $r_i \leftarrow \texttt{rand}(0,1)$
 6:     $x'_i \leftarrow x_i \oplus r_i$
 7:     $x'_n \leftarrow x_n \oplus r_i$
 8: $r_0 \leftarrow \texttt{rand}(0,1)$                    ▷ $r_0$ is used to compute $\mathcal{W}$ and $\mathcal{R}$
 9: $x'_n \leftarrow x_n \oplus \mathcal{W} \oplus \mathcal{R}$
10: return $(((\tilde{x}'_j)_{j\in[0,d]}, (x'_i)_{i\in[1,n]})$
```

3.2 Correctness and Performance Analysis

Next we introduce the transformation $\mathtt{T}^{(n,d)}$ to generate a Boolean circuit that is protected by an (n,d)-masking scheme and uses the gadgets described in Section 3.1. The following lemma summarizes the correctness of the transformation $\mathtt{T}^{(n,d)}$.

Lemma 1. *Let us denote the Boolean circuit that initialized with data D by $C[D]$. The transformation $\mathtt{T}^{(n,d)} : C[D] \mapsto C'[D']$ where C' uses `And`, `Xor`, `RefreshMask` gadgets and `Encoding`, `Decoding` functions described in Section 3 with randomness gates is a functionality preserving transformation, i.e. $C[D]$ and $C'[D']$ have the same input-output behaviour.*

Proof of the lemma can be found in Appendix A. In conclusion the transformation $\mathtt{T}^{(n,d)}$ can be used to transform any circuit to an (n,d)-masked circuit in a functionality preserving manner. Although we are using an n^{th} order linear masking, the scheme only provides an $(n-1)^{th}$ SCA security. Due to the non-linear sharing, the masking loses one share to increase the decoding order. Also the algebraic security depends on the structure of the Equations (1), (2), and (3) in each gadget as underlined above. The details can be found in Section 4.2.

Performance Analysis: In order to compare our construction with the previous schemes we analyze the performance of our scheme in terms of bitwise operations and randomness requirements. An analytical comparison of different orders and a comparison between `ISW` transformation and (n,d)-masking scheme can be found in Table 2.

In the following analysis for the simplicity, we use the symbol vertical bar $(|)$ to separate the number of Xor, And operations respectively. And we exclude the `RefreshMask` gadgets inside the `Xor` and `And` gadgets to analyze the constructions straightforwardly. Since the structure of the bottleneck variables depends on the non-linear degree d, we use a symbolic approach to analyze the performance numbers for the higher orders (i.e. for $d \le 3$). We use the subscripts

Table 2. The number of bitwise operations in Masked Operations. Remark that $(n,0)$-masking scheme corresponds to ISW gadgets. The last part of the table corresponds to the overhead of (n,d)-masking scheme compared to ISW transformation.

	Xor	And	Randomness
Xor$[n,0]$	$n+1$	-	-
And$[n,0]$	$2n(n+1)$	$(n+1)^2$	$n(n+1)/2$
RefreshMask$[n,0]$	$2n$	-	n
Xor$[n,1]$	$n+4$	2	-
And$[n,1]$	$2n^2+5n-1$	n^2+7n+2	$n(n+3)/2$
RefreshMask$[n,1]$	$2n+8$	3	$n+2$
Xor$[n,2]$	$n+9$	6	-
And$[n,2]$	$2n^2+15n-2$	$n^2+27n+3$	$(n+5)/2$
RefreshMask$[n,2]$	$2n+25$	20	$n+3$
Xor$[n,d]$	$n+d+2+\mathcal{U}_x$	$\mathcal{U}_a$	-
And$[n,d]$	$n(2n+d-1)+\mathcal{V}_x$	$n^2+d+1+\mathcal{V}_a$	$n(n+2d+1)/2$
RefreshMask$[n,d]$	$2n+d+1+\mathcal{W}_x+\mathcal{R}_x$	$\mathcal{W}_a+\mathcal{R}_a$	$n+d+1$
	Overhead		
Xor$[n,d]$	$d+1+\mathcal{U}_x$	$\mathcal{U}_a$	-
And$[n,d]$	$n(2n+d-3)+\mathcal{V}_x-1$	$d+\mathcal{V}_a-n$	nd
RefreshMask$[n,d]$	$d+1+\mathcal{W}_x+\mathcal{R}_x$	$\mathcal{W}_a+\mathcal{R}_a$	$d+1$

to denote the number of operations within $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{R}$, e.g., $\mathcal{U}_x$ and $\mathcal{U}_a$ represent the number of bitwise Xor, And operations within $\mathcal{U}$.

As seen in Table 2, the Xor gadget can be transformed efficiently. The cost of the gadget in the ISW transformation is $n+1$ bitwise Xor while an (n,d)-masking requires $n+d+2$ bitwise Xor and the additional cost of the variables $\mathcal{U}$. Therefore, the cost of the Xor gadget can be calculated as; $(n+d+2+\mathcal{U}_x)|\mathcal{U}_a$.

The cost of an And gadget can be analyzed easily by comparing the steps with ISW transformation. As seen in the construction in Section 3, the gadget can be divided into three stages.

– **Step-1** requires $n(d+1)$ random bits and the cost of processing these values can be calculated as $n(d+1)|d+1$.
– **Step-2(a)** includes the calculations of $r_{j,0}$ for $1 \leq j \leq n$. For the $(n,1)$ masking, $\mathcal{V}_x = 4n$ and $\mathcal{V}_a = 7n$. Additionally, the calculations of $r^{0,1} \oplus \ldots \oplus r^{0,n}$ require $n-1$ Xor. Similarly, $(n,2)$ masking $\mathcal{V}_x = 12n$ and $\mathcal{V}_a = 27n$. Also the intermediate variables u, v, and uv are calculated only once and they require $2(n-1)|1$.
– **Step-2(b) & Step-3** involve the calculations of $r_{j,i}$ for $1 \leq i < j \leq n$, $i \neq 0$ and Step-3. These parts can be processed as identical to the ISW transformation and cost $2n(n-1)|n^2$ while the required number of random bits is $n(n-1)/2$. Observe that the cost of these parts equals to an ISW-AND gadget with n shares.

To sum up, we express the cost of $\texttt{And}[n, d]$ gadget as $(n(2n+d-1)+\mathcal{V}_x)|(n^2+d+1+\mathcal{V}_a)$, and the required randomness as $n(n+2d+1)/2$.

We analyze the performance of the RefreshMask gadget using a similar methodology. The total number of required randomness and the number of required bitwise Xor operations can be calculated as $n+d+1$ and $2n+d+1$ respectively. As in the previous gadgets, the calculations of $\mathcal{W}$ and $\mathcal{R}$ add more calculations to the structure. The numbers for $\texttt{RefreshMask}[n, 1]$ and $\texttt{RefreshMask}[n, 2]$ can be seen in Table 2.

Using the performance analysis, we reveal the exact overhead of our scheme. The numbers in the overhead section of Table 2 can be calculated by comparing the cost of with n^{th}-order ISW transformation by an (n, d)-masking scheme. As seen in the table, the cost principally depends of the calculation on the values $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{R}$ while the randomness is affected by the masking degrees n and d.

4 Security Against Computational and Algebraic Attacks

Security in the grey-box model is a well-established issue of cryptography. In this paper, we use the definition for n^{th} order SCA security (security against t-probes for $t \leq n$ as proposed by Ishai et al. [26]) for white-box designs and security against algebraic attacks of degree d as proposed in [4]. First we recall briefly both security notions and then we prove that our construction is secure against SCA of any order up to $n-1$ and against algebraic attacks for $d = 1$ and $d = 2$. Remark that the security against SCA follows the security against computational attacks of the same order, since the underlying idea of computational attacks relies on side-channel analysis.

4.1 Security Models

Roughly speaking, in the setting of the n^{th} order SCA security, an adversary may invoke the (randomized) construction multiple times and adaptively choose the inputs. Prior to each invocation, the adversary may fix an arbitrary set of $t \leq n$ internal wires of the circuit values of which can be observed during that invocation. We use in this paper the following common definition of the SCA model (see e.g. [26, 37]).

Definition 1. *A randomized secret key encryption algorithm is said to achieve n^{th}-order SCA security if every t-tuple, with $t \leq n$, of its intermediate variables is independent of any sensitive variable.*

Here, by a *sensitive variable* of an encryption construction we mean any variable, with the exception of the resulting ciphertext or any deterministic function on it, that can be expressed as a deterministic function of the given plaintext and the secret key. Additionally, we assume that the function is not constant with respect to the secret key.

In [26] Ishai et al. provide a general construction of circuits using masking shares of size n and they proved that the generic construction achieves the $(n/2)^{th}$-order SCA security. Rivain and Prouff [37] improved the analysis in [26] showing that the `ISW` transformations achieves SCA security of order n.

Note that SCA security is a necessary but not a *sufficient* condition for a secure white-box implementation. A white-box adversary can implement an algebraic attack to recover secret key from a masked white-box implementation. The main idea of the algebraic attack is finding a d^{th} order function of intermediate variables[2] such that the output of this function will be equal to a predictable vector. To cover the algebraic attacks a new security notion called *Prediction Security* is defined in [4]:

Definition 2 (Prediction Security (d-PS), [4]). *Let $C : \mathbb{F}_2^{N'} \times \mathbb{F}_2^{R_C} \to \mathbb{F}_2^M$ be a Boolean circuit, $E : \mathbb{F}_2^N \times \mathbb{F}_2^{R_E} \to \mathbb{F}_2^{N'}$ an arbitrary function, $d \geq 1$ an integer, and $\mathcal{A}$ an adversary. Consider the following security experiment:*

Algorithm 4 $\mathrm{PS}^{C,E,d}(\mathcal{A}, b)$

```
1: $(\tilde{f}, x^{[0]}, x^{[1]}, \tilde{y}) \leftarrow \mathcal{A}(C, E, d)$ where
       $\tilde{f} \in \mathcal{F}^{(d)}(C), x^{[l]} = (x_1^{[l]}, \ldots, x_Q^{[l]}), x_i^{[l]} \in \mathbb{F}_2^N, \tilde{y} \in \mathbb{F}_2^Q$
2: $(r_1, \ldots, r_Q) \overset{\$}{\leftarrow} (\mathbb{F}_2^{R_E})^Q$
3: $(\tilde{r}_1, \ldots, \tilde{r}_Q) \overset{\$}{\leftarrow} (\mathbb{F}_2^{R_C})^Q$
4: for $f \in \mathcal{F}^{(d)}(C)$ do
5:     $y(f) = (f(E(x_1^{[b]}, r_1), \tilde{r}_1), \ldots, f(E(x_Q^{[b]}, r_Q), \tilde{r}_Q))$
6: $F \leftarrow \{f \in \mathcal{F}^{(d)}(C) \mid y(f) = \tilde{y}\}$
7: if $F = \{\tilde{f}\}$ then return 1 else return 0
```

In the above experiment, $\overset{\$}{\leftarrow}$ means sampling uniformly at random. Define the advantage of an adversary $\mathcal{A}$ as

$$Adv_{C,E,d}^{PS}[\mathcal{A}] \;=\; \left| \Pr[PS^{C,E,d}(\mathcal{A}, 0) = 1] - \Pr[PS^{C,E,d}(\mathcal{A}, 1) = 1] \right|$$

The pair (C, E) is said to be d^{th} order prediction-secure (d-PS) if for any adversary $\mathcal{A}$ the advantage is negligible.

Although it may seem that one definition covers the other one, in fact they are *incomparable.* Therefore, both definitions are needed to analyze a secure white-box implementation.

To illustrate the incomparability of two notions, let us consider two examples; a white-box implementation protected with a n^{th}*-order Boolean masking* and *minimalist quadratic masking* defined in [4].

[2] The attacker can observe all intermediate variables, therefore she can look for any d^{th} order combination of intermediate variables

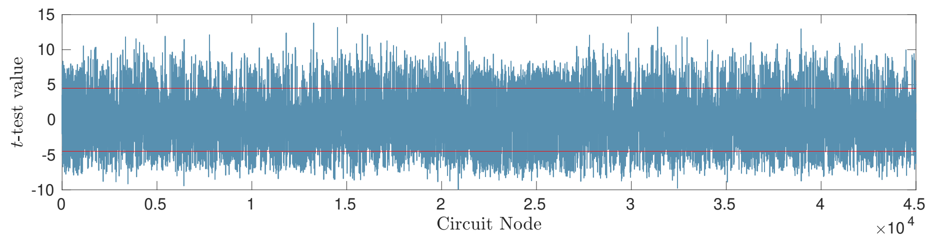

Fig. 1. A first-order leakage detection on a circuit that simulates AES-128 with the masking defined in [4]. Clearly, the t-test value exceeds the threshold values shown by red lines.

Example 1 (SCA Secure Masking Vulnerable to Algebraic Attacks). By definition, an `ISW` transformation to the circuit and the data results in an n^{th}-order SCA secure implementation. However, a first-order algebraic attack can exploit a first-order (linear) combinations of intermediate values which equal to a predictable value. Therefore, an n^{th}-order Boolean masking is secure in SCA model, but not secure in prediction security as seen in [4].

Example 2 (Algebraically secure masking vulnerable to SCA). As the second example, we use the encoding function $\texttt{Encode}(x, x_0, x_1) = (x_0, x_1, x_0x_1 \oplus x)$. As given in [4] the masking scheme satisfies the first order algebraic security. However, it is not SCA secure, even with respect to the first order, due to its unbalanced sharing which causes that intermediate variable $x_0x_1 \oplus x$ encoding the third share is dependent of the sensitive variable x. Indeed, for any value x we have $\Pr_{x_0,x_1 \in_R \mathbb{F}_2}[(x_0x_1 \oplus x) = x] = 3/4$. Thus, there exists no first order function that is equal to a predictable vector, but there exits one node (the last share) that is highly correlated with a predictable vector.

In order to verify this, we implement a basic bitwise AES-128 circuit using Sbox designed by Boyar and Peralta [10] and implement a basic leakage detection test using 500 traces with 45000 nodes ($N = 500$ and $M = 45000$). As seen in Figure 1, the test shows the intense leakage. The details of the experimental setup regarding the leakage detection, trace collection and the variable selection can be found Section 5.1.

As illustrated in Example 1, the prediction security is based on finding a degree-d function whose output equals to a predictable value. However, in SCA we only need to find a set of variables which depends on a predictable value as seen in Example 2. As a main result, we prove the security of our scheme in two steps:

– There exits no set of intermediate variables with $t \leq n$ elements such that the set depends on a predictable value.
– There exists no d^{th} order function such that the output equals to a predictable value.

4.2 Security Against Computational Attacks in the SCA model

We start with providing some auxiliary notions which generalize the corresponding definitions given in [37, 38]. A vector $\overline{x} = (\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n)$ of $n + d + 1$ intermediate variables is called an *(n, d)-family of shares* if every tuple of the form $((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ such that $|\tilde{I}| \leq d + 1$ and $|I| \leq n - 1$ of $\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n$ is uniformly distributed and independent of any sensitive variable and $\prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i$ is a sensitive variable. Two (n, d)-families of shares $\overline{x} = (\tilde{x}_0 \ldots, \tilde{x}_d, x_1, \ldots, x_n)$ and $\overline{y} = (\tilde{y}_0 \ldots, \tilde{x}_d, y_1, \ldots, x_n)$ are called to be *$(n - 1)$-independent* of one another if every tuple composed of $((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ and $((\tilde{y}_j)_{j \in \tilde{J}}, (y_j)_{j \in J})$ with $|\tilde{I}|, |\tilde{J}| \leq d + 1$ and $|I|, |J| \leq n - 1$ is uniformly distributed and independent of any sensitive variable. Two (n, d)-families are *$(n - 1)$-dependent* of one another if they are not $(n - 1)$-independent.

To prove the SCA security of an implementation C of an encryption scheme, we decompose C into basic components, which we call *randomized elementary transformations*. Such a component gets as input two $(n-1)$-independent (n, d)-families of shares, resp. one (n, d)-family of shares, and it returns a (n, d)-family of shares.

In this section we prove first that the randomized elementary transformations specified as Algorithm 1, 2, and 3 for computing `Xor`, `And`, respectively `RefrashMask` gadgets are $(n - 1)^{th}$ SCA secure. In the proofs we will use the following slight generalization of Lemma 1 given in the full version [38] of the work [37]. Since the proof in [37] can be easily modified for our setting, we skip it here.

Lemma 2 ([37]). *A randomized elementary transformation achieves $(n-1)^{th}$-order SCA security if and only if the distribution of every $t \leq n - 1$-tuple of its intermediate variables can be perfectly simulated from at most $d + 1$ non-linear shares and at most $n - 1$ linear shares of each of its input families.*

Now we are ready to prove SCA security of our basic constructions. We start with `RefreshMask` gadget.

Proposition 1. *Let $\overline{x} = (\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n)$ be an (n, d)-family of shares, with $n \geq 2$, in input of Algorithm 3 to refresh masking. Then the distribution of every tuple of $t \leq n - 1$ intermediate variables in Algorithm 3 is independent of the distribution of values taken by $x = \prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i$.*

Proof. In order to prove the proposition, we use Lemma 2 and show that every tuple of intermediate variables $(v_1, \ldots, v_t)$ with $t \leq n - 1$ elements can be simulated from two tuples of input shares $(\tilde{x}_i)_{i \in \tilde{I}}$ and $(x_i)_{i \in I}$ such that $|\tilde{I}| \leq d+1$ and $|I| \leq n-1$. We denote the concatenation of these tuples as $U = ((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$.

We first need to construct the sets of indices I and $\tilde{I}$ depending on the selected intermediate variables v_k which can be divided as follows:

- For all selected r_i, x_i and $x_i \oplus r_i$ add i to I.
- For all selected $\tilde{r}_i$, $\tilde{x}_i$ and $\tilde{x}_i \oplus \tilde{r}_i$ add i to $\tilde{I}$.

– For all selected x_n and $x_n \oplus r_i$ add n to I.

The above steps cover all the variables besides those used in line 11 in Algorithm 3. In order to simulate the variables in line 11 we need to consider the following (note that in the expression in line 11, only shares $\tilde{x}_i$ and random variables are used):

– For all selected values of the form $\tilde{x}_i \oplus r_0$ add i to $\tilde{I}$.
– If one of the variables of form $\prod_{i \in J}(\tilde{x}_i \oplus r_0) \prod_{i \notin J} \tilde{r}_i$ where $J \subsetneq \{0, \ldots, d\}$ is selected, add all $i \in [0, d]$ to $\tilde{I}$.

According to our selection, we add at most one index to I and in the worst case we add $d+1$ elements to $\tilde{I}$ per selected internal variable v_k. Next we will show how to simulate a t-tuple of intermediate variables in Algorithm 3 using the tuple U. First we need to consider the simulation of random values r_i and $\tilde{r}_i$ involved in the computation of v_k.

– All r_i (resp. $\tilde{r}_i$) are assigned random values.

After assigning the random values we can consider the intermediate variables.

– Every value of the form x_i, r_i, or $x_i \oplus r_i$ (resp. $\tilde{x}_i$, $\tilde{r}_i$, or $\tilde{x}_i \oplus \tilde{r}_i$) can be perfectly simulated since $i \in I$ (resp. $i \in \tilde{I}$) and the needed values of r_i (resp. $\tilde{r}_i$) have already been assigned in the simulation as stated above.
– Thus the only remaining variables are of the form x_n and $x_n \oplus r_i$.
  - if $n \notin I$ the values do not enter the computation of any selected value and therefore the values can be left unassigned.
  - if $n \in I$ then the value x_n can be simulated by the tuple U (since $n \in I$) and $x_n \oplus r_i$ can be simulated by assigning r_i a random value.
– We need to pay special attention to the values in $\mathcal{W}$ and $\mathcal{R}$ used in line 11.
  - Every value of the form $\tilde{x}_i \oplus r_0$ can be perfectly simulated since $i \in \tilde{I}$ and r_0 is assigned a random value as stated above.
  - Every value of the form $\prod_{i \in J}(\tilde{x}_i \oplus r_0) \prod_{i \notin J} \tilde{r}_j$ where $j \subsetneq \{0, \ldots, d\}$ can be simulated according to our selection. Either all $i \in [0, d]$ is in $\tilde{I}$ or the value of the form is not selected at all. In both cases we can perfectly simulate variables with the set U.
  - Every value in $\mathcal{R}$ contains the non-linear input shares or random values. Therefore we either have all $i \in [0, d]$ in $\tilde{I}$ or the values in $\mathcal{R}$ are not selected at all. In both cases we can perfectly simulate variables with the set U.

In conclusion, we show that any set of intermediate variables $(v_1, \ldots, v_t)$, with $t \leq n-1$ elements, can be simulated by $U = ((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ such that $|\tilde{I}| \leq d+1$ and $|I| \leq n-1$. By the definition of our masking U is uniformly distributed and independent of any sensitive variable and hence `RefreshMask` gadget seen in Algorithm 3 is an $(n-1)^{th}$ SCA secure gadget.

Proposition 2. *Let $\overline{x} = (\tilde{x}_0 \ldots, \tilde{x}_d, x_1, \ldots, x_n)$ and $\overline{y} = (\tilde{y}_0 \ldots, \tilde{y}_d, y_1, \ldots, x_n)$ be two $(n-1)$-independent (n,d)-families of shares, with $n \geq 2$, inputs of Algorithm 2 for* `And`. *Then the distribution of every tuple of $t \leq n-1$ intermediate variables in Algorithm 2 is independent of the distribution of values taken by $x = \prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i$ and $y = \prod_{j=0}^{d} \tilde{y}_j \oplus \bigoplus_{i=1}^{n} y_i$.*

Proof. In order to prove the proposition, we use the Lemma 2 and show that every set of intermediate variables $(v_1, \ldots, v_t)$ with $t \leq n-1$ elements can be simulated by two sets of input shares $(\tilde{x}_i)_{i \in \tilde{I}}$ and $(x_i)_{i \in I}$ such that $|\tilde{I}| \leq d+1$ and $|I| \leq n-1$, resp. $(\tilde{y}_j)_{j \in \tilde{J}}$ and $(y_j)_{j \in J}$ such that $|\tilde{J}| \leq d+1$ and $|J| \leq n-1$. We denote the concatenations of these tuples by $U = ((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ and $V = ((\tilde{y}_j)_{j \in \tilde{J}}, (y_j)_{j \in J})$.

We first need to construct the sets of indices I and $\tilde{I}$ corresponding to shares of x, and J and $\tilde{J}$ corresponding to shares of y. The following two cases cover every variable in **Step-2(b)** and **Step-3**:

- For all x_i, y_i, $x_i y_i$, $r_{i,j}$ or xor of these values add i to I and J.
- For all $x_i y_j$ or $r_{i,j} \oplus x_i y_j$ add i to I and j to J.

To cover **Step-1** and **Step-2(a)** we need to follow the steps below:

- For all $\tilde{x}_i$, $\tilde{y}_i$, $r^{i,j}$ and combination of these add i to $\tilde{I}$ and $\tilde{J}$.
- For all $\tilde{x}_i \tilde{y}_{i'}$ add i to $\tilde{I}$ and i' to $\tilde{J}$.
- For all $\tilde{x}_i y_j$ (resp. $\tilde{y}_j x_i$) add i to $\tilde{I}$ and j to J (resp. add i to I and j to $\tilde{J}$).
- For all values of the form $\prod_{i \in K} \tilde{x}_i \prod_{j \in L} \tilde{y}_j$ where $K, L \subsetneq \{0, \ldots, d\}$ add all $i \in K$ to $\tilde{I}$ and all $j \in L$ to J.

According to our selection, we add at most one index to I (resp. J) and in the worst case $d+1$ elements to $\tilde{I}$ (resp. $\tilde{J}$). Let us examine the simulation of the random values:

- If $i \notin I$ (resp. $i \notin \tilde{I}$) then r_i (resp. $\tilde{r}_i$) does not enter the computation of any selected value and therefore can be left unassigned.
- If $i \in I$ and $j \notin J$ then:
  - if $i < j$ then $r_{i,j}$ is assigned a random value,
  - otherwise $r_{j,i}$ is not involved in the computation of any selected value; therefore we can assign a random value to $r_{i,j}$.
- If $i, j \in I \cap J$ then the values x_i, x_j, y_i, y_j can be simulated and $r_{i,j}$ (assigned as random) and $r_{j,i}$ can be calculated as in Algorithm 2 i.e., the value $r_{j,i}$ can be calculated as $(r_{i,j} \oplus x_i y_j) \oplus x_j y_i$.
- If $i, j \in I$ and if $i, j \notin I \cap J$ then at least one of the $r_{i,j}$ or $r_{j,i}$ does not enter the computation of the selected value, therefore the values can be assigned a random value.

Note, that the above classification is based on the one given in [38]. In our construction we need to examine the additional randomness used in **Step-1**.

– If $i \notin \tilde{I}$ then $r^{i,j}$ for $j \in [1,n]$ is not involved in any computation and therefore it can be left unassigned.
– If $i \in \tilde{I}$ then $r^{i,j}$ for $j \in [1,n]$ should be assigned a random value.

Next we show how to simulate the variables using the tuples U and V.

– Every variable x_i, y_i, x_iy_i, $r_{i,j}$ or xor of these values can be simulated according to our selection.
– x_iy_j or $r_{i,j} \oplus x_iy_j$ can be perfectly simulated since $i \in I$, $j \in J$ which enables us to compute x_iy_j and $r_{i,j}$ have been assigned.

We note, that the above steps cover the variables in **Step-2(b)** and **Step-3** and are based on [38]. In order to simulate the remaining variables we need to examine the variables as follows:

– Every variable $\tilde{x}_i$, $\tilde{y}_i$, $\tilde{x}_i\tilde{y}_{i'}$, $\tilde{x}_iy_j$, $\tilde{y}_jx_i$ $r^{i,j}$ or xor of these values can be simulated according to our selection.
– Every variable in **Step-2(a)** can be simulated since (even in the worst case) $\tilde{I}$, $\tilde{J}$ can contain $\tilde{x}_{i\in[0,d]}$ and $\tilde{y}_{j\in[0,d]}$.

Hence, we show that any set of intermediate variables $(v_1, \dots, v_t)$ with $t \leq n-1$ elements can be simulated by $U = ((\tilde{x}_i)_{i\in\tilde{I}}, (x_i)_{i\in I})$ and $V = (\tilde{y}_j)_{j\in\tilde{J}}, (y_j)_{j\in J})$ such that $|\tilde{I}|, |\tilde{J}| \leq d+1$ and $|I|, |J| \leq n-1$. By the definition of our masking U and V are uniformly distributed and independent of any sensitive variable and hence the `And` gadget introduced in Algorithm 2 is an $(n-1)^{th}$ SCA secure elementary transformation.

Proposition 3. *Let* $\overline{x} = (\tilde{x}_0, \dots, \tilde{x}_d, x_1, \dots, x_n)$ *and* $\overline{y} = (\tilde{y}_0, \dots, \tilde{y}_d, y_1, \dots, x_n)$ *be two* $(n-1)$*-independent* (n,d)*-families of shares, with* $n \geq 2$*, in input of Algorithm 1 to compute* `Xor`*. Then the distribution of every tuple of* $t \leq n-1$ *intermediate variables in Algorithm 1 is independent of the distribution of values taken by* $x = \prod_{j=0}^{d} \tilde{x}_j \oplus \bigoplus_{i=1}^{n} x_i$ *and* $y = \prod_{j=0}^{d} \tilde{y}_j \oplus \bigoplus_{i=1}^{n} y_i$*.*

The proof of Proposition 3 can be found in Appendix A. Thus we prove the SCA security aspect of the individual gadgets introduced in Section 3. The following theorem analyzes an arbitrary circuit C as a combination of our gadgets and shows that C' (as defined below) is secure against $(n-1)^{th}$-order SCA attacks and therefore secure against $(n-1)^{th}$-order computational attacks.

Theorem 1. *Assume a circuit* C *is transformed to* C' *using* $T^{(n,d)}$*, with* $n \geq 2$ *and* $d \geq 1$*, described in Section 3. Then* C' *is secure against* $(n-1)^{th}$*-order computational attacks.*

Proof. The randomized circuit C' is expressed as a combination of `Xor`$[n,d]$, `And`$[n,d]$ and `RefreshMask`$[n,d]$ gadgets and the gadgets take either an (n,d)-*family of shares* or two (n,d)-*family of shares*. By the Propositions 1, 2, and 3 we know that the gadgets achieve $(n-1)^{th}$ order SCA security and any set of intermediate variables with $\leq n-1$ elements selected within the gadgets can be simulated by a set of inputs such that $((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ with $|\tilde{I}| \leq d+1$ and $|I| \leq n-1$. By the definition of our masking, the set of input stated above is uniformly distributed and independent of any sensitive variable. Any set of intermediate variables in C' with $\leq n-1$ elements can be perfectly simulated by a set of input shares which is uniformly distributed and independent of any sensitive variable. Hence C' is an $(n-1)^{th}$ SCA secure circuit.

4.3 Algebraic Security of the $(n, 1)$-Masking Scheme

In this section we analyze the algebraic security (Def. 2) of $(n, 1)$-masking scheme using the gadgets in Section 3.1. We use the auxiliary (ϵ-1-AS) security definition from [4].

Definition 3 (Circuit Algebraic Security (ϵ-1-AS), [4]). *Let $C(x, r) : \mathbb{F}_2^N \times \mathbb{F}_2^{R_C} \rightarrow \mathbb{F}_2^M$ be a Boolean circuit. Then C is called first-order algebraically ϵ-secure (ϵ-1-AS) if for any $f \in \mathcal{F}^{(1)}(C) \setminus \{\mathbf{0}, \mathbf{1}\}$ one of the following conditions holds:*

1. *f is an affine function of x,*
2. *for any $x \in \mathbb{F}_2^N$, $\mathcal{E}(f(x, \cdot)) \leq \epsilon$ where $f(x, \cdot) : \mathbb{F}_2^{R_C} \rightarrow \mathbb{F}_2$.*

where $\mathcal{E}$ represents the bias of a Boolean function such that $\mathcal{E}(f) = |1/2 - wt(f)/2^N|$ and $wt(f)$ is the weight function.

The methodology to prove the algebraic security in [4] can be divided into two steps. The first part consists of showing $\mathcal{E}(f(x, r)) \neq 1/2$ for all $f \in \mathcal{F}^{(1)}(C)$ and for all $x \in \mathbb{F}_2^N$ except the constant functions and affine functions of x. To solve this, a verification algorithm is given in [4]. Briefly speaking, the algorithm generates a truth table by evaluating the circuit on all possible inputs and recording each node in the circuit. Another truth table is formed by selecting the values where the input is fixed $x = c$. That is, the second truth table corresponds to the values of the circuit nodes where the input x is fixed to a value c while r takes all possible values. Observe that, the latter truth table is a subset of the former one. Finally, the algorithm compares the dimensions of the basis of the truth tables for each restriction, to check if there is a constant function f when the input is fixed to a value c.

The second part is processed by finding the maximum degree term (i.e. node in the circuit) and calculating the corresponding bias bound. As proven in [32], the degree of a Boolean function gives us a boundary for the weight of the function such that $wt(f) \leq 2^{N-deg(f)}$ where N is the number of inputs of the function f. Observe that, the maximum degree that $f \in \mathcal{F}^{(1)}(C)$ can have is equal to the maximum degree node in C since f contains *only* the linear combinations of the nodes. That is for all $f \in \mathcal{F}^{(1)}(C)$, $deg(f) \leq max(deg(c_i)_{c_i \in C})$ and thus $wt(f) \geq 2^{N-max(deg(c_i)_{c_i \in C})}$. Using this minimum weight value, the linear-bias bound of the gadget can be calculated as:

$$\epsilon = \left| \frac{1}{2} - \frac{wt(f)}{2^N} \right| \leq \left| \frac{1}{2} - \frac{2^{N-deg(f)}}{2^N} \right| = \left| \frac{1}{2} - \frac{1}{2^{deg(f)}} \right|.$$

Due to the first part of the proof, we know that there are no constant functions and therefore bias cannot grow.

Using the discussion above we will prove the security of our gadgets by showing that there exists no constant function $f(c,\cdot) \in \mathcal{F}^{(1)}(C)$ for all $c \in \mathbb{F}_2^N$ and by calculating the corresponding bias boundary of the gadgets. We start with the first order algebraic security proof for an $\texttt{RefreshMask}[n,1]$ gadget that uses the construction given in Section 3.1.

Proposition 4. *Let C be the circuit representation of the* $\textit{\texttt{RefreshMask}}$ *gadget using a masking scheme with an arbitrary order n and a fixed degree $d = 1$. C takes as input $n+2$ shares $(\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n})$ and outputs $n+2$ shares $(\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n})$. The gadget* $\textit{\texttt{RefreshMask}}[n,1]$ *is ϵ-1-AS with $\varepsilon \leq 1/4$.*

The proof of Proposition 4 can be found in Appendix A. We proceed with first order algebraic security proof for an $\texttt{And}[n,1]$ gadget that uses the construction given in Section 3.1.

Proposition 5. *Let C be the circuit representation of the* $\textit{\texttt{And}}$ *gadget using a masking scheme with an arbitrary order n and a fixed degree $d = 1$. C takes as input $n+2$ shares $(\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n})$ and $(\tilde{y}_0, \tilde{y}_1, (y_i)_{1\leq i\leq n})$ and outputs $n+2$ shares $(\tilde{z}_0, \tilde{z}_1, (z_i)_{1\leq i\leq n})$. The gadget* $\textit{\texttt{And}}[n,1]$ *is ϵ-1-AS with $\varepsilon \leq 7/16$.*

Proof. In the first part of the proof, we show that there exists no function $f \in \mathcal{F}^{(1)}(C)$ such that f is constant when inputs are fixed.

First, let us the reformulate the circuit C as follows:

$$C\colon ((\mathbb{F}_2^{n+2} \times \mathbb{F}_2^{n+2}), \mathbb{F}_2^{R_C}) \to \mathbb{F}_2^{n+2}$$
$$((\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n}), (\tilde{y}_0, \tilde{y}_1, (y_i)_{1\leq i\leq n}), \bar{r}) \mapsto (\tilde{z}_0, \tilde{z}_1, (z_i)_{1\leq i\leq n}).$$

where $\bar{r}$ denotes the set of randomness that is used in the circuit. Next, we define three classes of edges within the circuit:

- R: The set of random bits,
- B: The set of linear shares i.e. x_i and y_j for all $1 \leq i,j \leq n$,
- M: The set of non-linear shares i.e. $\tilde{x}_0$, $\tilde{x}_1$, $\tilde{y}_0$ and $\tilde{y}_1$.

Using the above classification we can analyze the nodes $c_{i 1\leq i\leq M} \in C$ with respect to its input edges where M is the number of nodes in C. We define the nodes as $c_i : (u_i^1, u_i^2) \mapsto v_i$ where $u_i^1,\ u_i^2 \in \mathbb{F}_2$ represent the input bits of the node and $v_i \in \mathbb{F}_2$ represents the output bit of the node. The classification of the nodes can be listed as follows;

1. $u_i^1 \in \mathsf{R}$ or $u_i^2 \in \mathsf{R}$,
2. $u_i^1 \in \mathsf{B}$ or $u_i^2 \in \mathsf{B}$,
3. $u_i^1 \in \mathsf{M}$ and $u_i^2 \in \mathsf{M}$.

Assume that there exists a function $f \in \mathcal{F}^{(1)}(C)$ such that f is constant when the inputs $\overline{x}$ and $\overline{y}$ are fixed. We can represent the function as $f = \bigoplus_{i \in I} v_i$ where $I \subseteq [1, M]$. Remark that the input shares are randomized, since they are first processed by `RefreshMask` gadgets. Therefore f should include a reconstructed combination of the shares i.e., f should include a combination of nodes such that $\tilde{x}_0\tilde{x}_1 \oplus x_1 \oplus \cdots \oplus x_n$ (resp. $\tilde{y}_0\tilde{y}_1 \oplus y_1 \oplus \cdots \oplus y_n$) is formed.

Any linear combination of the nodes of 1 and 2 cannot be constant due to `RefreshMask` gadgets, since either a node is random (non fixed by definition) or the node corresponds to linear masking (non fixed by `RefreshMask`). Therefore f should include at least one node from the 3^{rd} class to form the reconstructed multiplicative representation: x_0 or y_0. Clearly, the nodes from the 3^{rd} class can be found in **Step-1** and **Step-2(a)** where the following computations are processed:

- $\tilde{z}_0$ and $\tilde{z}_1$,
- $\tilde{x}_1(\tilde{x}_0 y_j \oplus r^{0,j}\tilde{y}_0) = \tilde{x}_1\tilde{x}_0 y_j \oplus r^{0,j}\tilde{x}_1\tilde{y}_0$ for $1 \leq j \leq n$,
- $\tilde{y}_1(\tilde{y}_0 x_j \oplus r^{1,j}\tilde{x}_0) = \tilde{y}_1\tilde{y}_0 x_j \oplus r^{1,j}\tilde{y}_1\tilde{x}_0$ for $1 \leq j \leq n$.

The use of parenthesis indicates the order in which the nodes are used in the above equations. Therefore the order of the nodes eliminates the generation of an affine function of x_0 or y_0 (the shares represented by $\tilde{x}_0, \tilde{x}_1$ and $\tilde{y}_0, \tilde{y}_1$ respectively), although these nodes calculate the correct function ($\mathcal{F}(x_j, y_j)$ as seen in Equation (2)). Any linear combination of these nodes cannot be constant and thus there exists no constant function $f \in \mathcal{F}^{(1)}(C)$ such that inputs are fixed.

In the second part, we examine the highest degree term in the gadget and find the corresponding bias. For `And`$[n, 1]$ the maximum degree term can be found in line 18 of Algorithm 2. Specifically, $x_n y_n$ which contains a node of the form $\tilde{r}_0^x\tilde{r}_1^x\tilde{r}_0^y\tilde{r}_1^y$ where $\tilde{r}_0^x, \tilde{r}_1^x$ (resp. $\tilde{r}_0^y, \tilde{r}_1^y$) are the randomness used in `RefreshMask`$(\overline{x})$ (resp. `RefreshMask`$(\overline{y})$). Clearly the corresponding bias and the bias bound of the gadget can be calculated as 2^{-4} and $\epsilon \leq \left|1/2 - 1/2^4\right| = 7/16$ respectively. Thus `And` gadget is ϵ-1-AS with $\varepsilon \leq 7/16$.

Although we are not giving a proof for the `Xor` gadget (however the experimental verification of the `Xor` gadget can be found below), the same discussion can be carried out. Since any combination of algebraically secure gadgets is also algebraically secure by [4], we can use the gadgets in Section 3.1 to securely calculate an arbitrary Boolean circuit.

Experimental Verification: To support the results, we provide the experimental verification of the first order gadgets; `And`$[n, 1]$, `Xor`$[n, 1]$ (and inherently `RefreshMask`$[n, 1]$) for $n = 1, 2$ and 3 using the tool given in [4][3]. First we adapt our scheme to work with the tool, i.e. we implement our masking scheme (with the given orders n and d) as a class inside the tool. Next we run the verification algorithm as explained above. The updated version of the tool including our scheme is available as open source[4].

[3] `https://github.com/cryptolu/whitebox`

[4] `https://github.com/UzL-ITS/white-box-masking`

Table 3. First-order algebraic security verification of individual gadgets. Input corresponds the number of shares for both inputs (i.e. $2(n+2)$). Random states the number of random values (R_C) within the circuit and it is calculated by the randomness requirement of two `RefreshMask` gadgets and additional randomness in the gadget. The number of intermediate variables represents the number of nodes in the gadget.

	Max degree	Bias Bound	Input	Random	Intermediate	Time
`Xor`$[1,1]$	2	1/4	6	6	8	3.5 sec.
`And`$[1,1]$	4	7/16	6	6	12	4 sec.
`Xor`$[2,1]$	2	1/4	8	8	8	45.7 sec.
`And`$[2,1]$	4	7/16	8	13	24	$\approx$ 114min
`Xor`$[3,1]$	2	1/4	10	10	8	$\approx$ 17min
`And`$[3,1]$	4	7/16	10	19	36	$\approx$ 5 days

We confirm the first order algebraic security of our scheme for different orders and the details can be seen in Table 3. The algorithm is run on an Intel Xeon Silver 4114 CPU@2.20GHz and, as seen in the table, the time that algorithm takes increases exponentially with the increasing number of nodes within the gadgets. Observe that the bias bound does not depend on the linear degree n, since the maximum degree term is found within the terms that depend on the non-linear degree d.

4.4 Algebraic Security of the $(n, 2)$-Masking Scheme

In this section we use a similar strategy given in the Section 4.3 to prove the second order algebraic security of our gadgets. As we highlighted in Section 3.1, we can use the generic constructions for higher orders, however the sequence of nodes should be defined carefully in order to satisfy the algebraic security given in the previous sections. In order to prove the higher order algebraic security we propose the following lemma to *extend* a circuit. The main idea is to reduce the problem of the d^{th}-order algebraic security of the original circuit to the first-order algebraic security of the extended circuit.

Lemma 3. *Let C be a Boolean circuit with M nodes i.e. $|C| = M$ and let $C^{(d)}$ be the d^{th} order extension of the circuit C defined as follows:*

$$C^{(d)} = \{v_{i_1}\}_{1\le i_1\le M} \cup \{v_{i_1}v_{i_2}\}_{1\le i_1,i_2\le M} \cup \cdots \cup \{v_{i_1}v_{i_2}\cdots v_{i_d}\}_{1\le i_1,i_2,\ldots,i_d\le M}. \quad (4)$$

where v_i denotes the output bit of the i^{th} node. C is d^{th}-order prediction secure if and only if $C^{(d)}$ is first-order prediction secure.

Proof. By the definition, $C^{(d)}$ is generated by using all nodes and all up to d^{th} order combinations of the nodes of C. Thus the set of linear combinations of $C^{(d)}$ is equal to the set of d^{th} order combinations of the nodes of C, i.e. $\mathcal{F}^{(d)}(C) = \mathcal{F}^{(1)}(C^{(d)})$.

By Definition 2, we can define the connection between advantage of the adversaries as follows:

$$Adv^{PS}_{C,E,d}[A] = Adv^{PS}_{C^{(d)},E,1}[A]$$

Therefore, the d^{th} order prediction security aspect of C will be identical to first order prediction security aspect of $C^{(d)}$ Hence C is d^{th} order prediction secure if and only if $C^{(d)}$is first order prediction secure.

Using Lemma 3, we will prove the second order prediction security of our $(n, 2)$ construction by creating the second order extension of the circuit as shown in Equation (4). Then we show that there exists no constant function $f(c, \cdot) \in \mathcal{F}^{(1)}(C^{(2)})$ for all $c \in \mathbb{F}_2^N$ and calculate the corresponding *second-order* bias bound of the gadget. We start with the ϵ-2-AS of the $\texttt{RefreshMash}[n, 2]$ gadget.

Proposition 6. *Let C be the circuit representation of the* $\texttt{RefreshMask}$ *gadget using a masking scheme with an arbitrary order n and a fixed degree $d = 2$. C takes as input $n + 3$ shares $(\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, (x_i)_{1\leq i\leq n})$ and outputs $n + 3$ shares $(\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, (x_i)_{1\leq i\leq n})$. The gadget* $\texttt{RefreshMask}[n, 2]$ *is ϵ-2-AS with $\varepsilon \leq 31/64$.*

The proof of Proposition 6 can be found in Appendix A. Next we prove the second order algebraic security of $\texttt{And}[n, 2]$ gadget.

Proposition 7. *Let C be the circuit representation of the* $\texttt{And}$ *gadget using a masking scheme with an arbitrary order n and a fixed degree $d = 2$. C takes as input $n + 3$ shares $(\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, (x_i)_{1\leq i\leq n})$, $(\tilde{y}_0, \tilde{y}_1, \tilde{y}_2, (y_i)_{1\leq i\leq n})$ and outputs $n+3$ shares $(\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, (z_i)_{1\leq i\leq n})$. The gadget* $\texttt{And}[n, 2]$ *is ϵ-2-AS with $\varepsilon \leq (1/2-1/2^{12})$.*

Proof. Similar to the proof of Proposition 5, we reformulate the circuit C as follows:

$$C \colon ((\mathbb{F}_2^{n+3} \times \mathbb{F}_2^{n+3}), \mathbb{F}_2^{R_C}) \to \mathbb{F}_2^{n+3}$$
$$((\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, (x_i)_{1\leq i\leq n}), (\tilde{y}_0, \tilde{y}_1, \tilde{y}_2, (y_i)_{1\leq i\leq n}), \bar{r}) \mapsto (\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, (z_i)_{1\leq i\leq n}).$$

By the Lemma 3 we can define the second order extension of the circuit C as follows:

$$C' = \{v_i\}_{1\leq i\leq M} \cup \{v_i v_j\}_{1\leq i,j\leq M} \text{ where } |C| = M.$$

Next we use the classification of the nodes that we used in the proof of Proposition 5:

- R: The set of random bits,
- B: The set of linear shares i.e. x_i and y_j for all $1 \leq i, j \leq n$,
- M: The set of non-linear shares i.e. $\tilde{x}_0$, $\tilde{x}_1$, $\tilde{x}_2$, $\tilde{y}_0$, $\tilde{y}_1$ and $\tilde{y}_2$.

Using the above classification we can analyze the nodes $c_i \in C$ with respect to its input edges. We define the nodes as $c_i : (u_i^1, u_i^2) \mapsto v_i$ where $u_i^1,\ u_i^2 \in \mathbb{F}_2$ represent the input bits of the node and $v_i \in \mathbb{F}_2$ represents the output bit of the node. The classification of the nodes depending be listed as follows;

Table 4. The number of gadgets in one round of AES.

	SubBytes	MixColumns	AddRoundKey	ShiftRows
And	16×32	-	-	-
Xor	16×83	27	128	-

1. $u_i^1 \in \mathsf{R}$ or $u_i^2 \in \mathsf{R}$,
2. $u_i^1 \in \mathsf{B}$ or $u_i^2 \in \mathsf{B}$,
3. $u_i^1 \in \mathsf{M}$ and $u_i^2 \in \mathsf{M}$.

Assume that there exists a second-order combination $f \in \mathcal{F}^{(2)(C)}$ that is constant when the inputs are fixed. It follows that there exists a linear combination $f' \in \mathcal{F}^{(1)}(C')$ such that f' is constant when the inputs are fixed.

Let us denote the linear combination as $f' = \bigoplus_{i \in I} v_i'$ where $I \subset C'$. As in Proposition 5, input shares are randomized, due to the initial `RefreshMask` gadgets. Therefore f' should include a reconstructed combination of the shares i.e., f' should include a combination of nodes such that $\tilde{x}_0\tilde{x}_1\tilde{x}_2 \oplus x_1 \oplus \cdots \oplus x_n$ (resp. $\tilde{y}_0\tilde{y}_1\tilde{y}_2 \oplus y_1 \oplus \cdots \oplus y_n$) is formed. Using the same discussion we can see that f should include a node from the third class which can be found in **Step-1** and **Step-2(a)**.

However, the nodes $c_i \in C$ contains at most one value from the each multiplicative representation, i.e. all nodes in c_i are of the form $\tilde{x}_i\tilde{y}_j$ where $i, j \in \{0, 1, 2\}$. Therefore the nodes $c_i' \in C'$ can contain *at most* two non-linear share from an input. And hence any linear combinations of the nodes of C' i.e. for all $f' \in \mathcal{F}^{(1)}(C')$ cannot be fixed.

Thus we can conclude that there exits no constant function $f' \in \mathcal{F}^{(1)}(C')$ when the inputs are fixed. This result is followed by there exits no constant function $f \in \mathcal{F}^{(2)}(C)$ when the inputs are fixed by the Lemma 3.

In the second part of the proof we examine the highest degree term in the circuit. Similarly the maximum degree term can be found in line 18 of Algorithm 2 for `And`$[n, 2]$. We can see that the maximum degree term for this case is 6. Since we are looking into second-order combinations of the circuit i.e for all $f' \in \mathcal{F}^{(1)}(C')$, the maximum degree can be stated as 12. Therefore the bias bound can be calculated as: $\epsilon \leq |1/2 - 1/2^{12}|$. Thus C' is ϵ-1-AS circuit with $\epsilon \leq |1/2 - 1/2^{12}|$ and C (the circuit representation of `And`$[n, 2]$ gadget) is ϵ-2-AS circuit with $\epsilon \leq |1/2 - 1/2^{12}|$

Using the same idea we can prove that `Xor`$[n, 2]$ gadget is ϵ-2-AS circuit with $\epsilon \leq |1/2 - 1/2^6|$.

5 A Proof-of-Concept AES Implementation

In this section we introduce a white-box AES design based on the masking scheme defined in Section 3. The AES block cipher consists of multiple rounds of operations on its state. The operations include three linear layers: `MixColumns`,

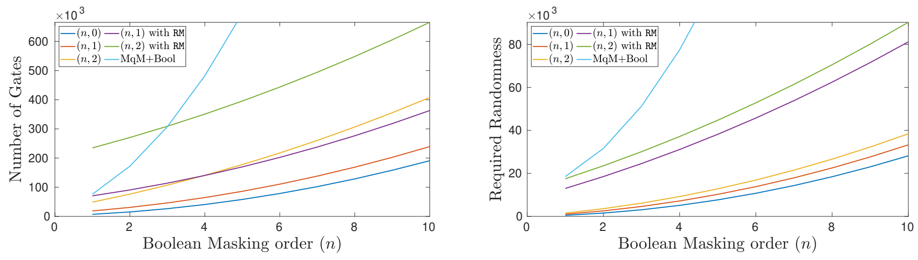

Fig. 2. Total number of bitwise operations and required randomness for one round of AES-128 with different $(n, 0)$, $(n, 1)$ and $(n, 2)$ masking schemes with and without initial `RefreshMask` gadgets

`ShiftRows`, and `AddRoundKey` and one non-linear layer `SubBytes`. The bitwise implementation for the linear operations can be defined straightforwardly. In our construction we use the bitwise AES-Sbox design by Boyar and Peralta [10] and the exact number of `And` and `Xor` gadgets within one round of AES-128 can be seen in Table 4. The total number of bitwise operations[5] can be calculated using Table 4 and the performance analysis in Table 2. A visual representation of AES-128 implementations with $(n, 0)$ (i.e. `ISW`-transformation), $(n, 1)$-masking scheme and $(n, 2)$-masking scheme is shown in Figure 2. Moreover the analysis contains the algebraically secure gadgets where each input is associated with a `RefreshMask` gadget, and the idea of using two different masking schemes (first Minimalist quadratic Masking and second Boolean masking as in [4]).

As seen in Figure 2, our hybrid constriction outperforms the idea of using a first order linear masking on top of a non-linear masking. As stated in [4] using a combination of two masks even with the first order protections requires roughly 200.000 gates per AES round. Since the foundation of our scheme is `ISW` transformation, we can increase the SCA security aspect of our scheme efficiently. However increasing the non-linear order is the bottleneck of our scheme. When we compare the smallest possible implementations, we see that one round of AES-128 with $(2, 0)$, $(2, 1)$ and $(2, 2)$-masking schemes requires 15201, 30808(90658) and 76358(270385) gates respectively where the values in the parenthesis correspond to the gadgets associated with `RefreshMask` gadgets. Clearly, `RefreshMask` gadgets impose a heavy overhead on our scheme. Therefore a significant performance advantage can be achieved by further optimizing the `RefreshMask` gadget. While the first order algebraically secure implementation requires a small overhead over an unprotected implementation, the second-order algebraically secure implementation comes with a substantial cost. One round of AES-128 with $(2, 1)$, $(3, 1)$ and $(4, 1)$-masking schemes requires 30808(90658), 46115(113945) and 64494(140304) gates respectively. Therefore we can conclude

[5] The bitwise `SubBytes` design by Boyar and Peralta [10] requires Not gates also. Although we didn't give the explicit description of a `Not` gadget in our masking scheme, it can be easily defined as identical to the `Not` gadget in `ISW` transformation i.e. by flipping the n^{th} share.

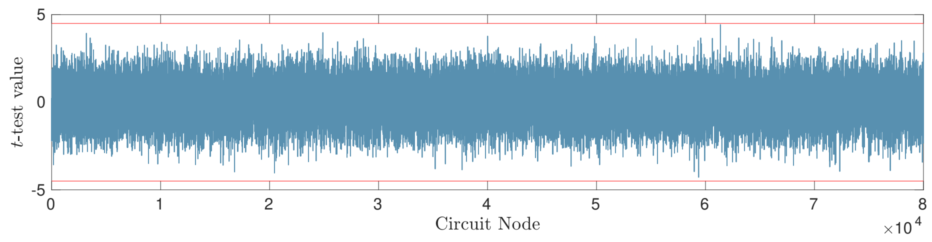

Fig. 3. A first-order leakage test on a circuit that simulates the AES-128 with $(2,1)$-masking defined in Section 3.1. Clearly, t-test value lie in threshold values as drawn by red lines ($[-4.5, 4.5]$).

that, one can increase the security against computational attacks with small overhead. Furthermore, the randomness requirements of our scheme increases similarly to the `ISW`-transformation as seen in Figure 2.

5.1 Experimental Setup

To experimentally verify the security properties of our scheme we used the proof-of-concept AES-128 implementation. The implementations using $(n,0)$, $(n,1)$ and $(n,2)$ masking schemes including the analysis are available as open source[6].

Software traces are simulated by encrypting N random plaintext and collecting the output of each node. We denote i^{th} trace (corresponds to the encryption of i^{th} plaintext) by $t_i = \{v_1^i, \ldots, v_M^i\}$ where v_j^i denotes the output of j^{th} node and M denotes the number of the nodes in the circuit. Using the software traces we demonstrate a simple leakage detection test by the test vector leakage assessment (TVLA) as proposed by Gilbert et al. [23]. In the first part of the test, two different sets of side-channel traces are collected by processing either a fixed input or a random input under the same conditions in a random pattern. After collecting the traces means (μ_f, μ_r) and standard deviations (σ_f, σ_r) for two sets are calculated. Welch's t-test is executed as in Equation (5) where n_f and n_r denote the number of traces for fixed and random sets respectively.

$$t = \frac{\mu_f - \mu_r}{\sqrt{(\sigma_f^2/n_f) + (\sigma_r^2/n_r)}}. \tag{5}$$

Using the experimental setup we implement a first order leakage detection test using 10000 traces (i.e. $n_f + n_r = 10000$) and $M = 80000$ (corresponds to the two round of AES-128). As expected the test results in no observable leakage. The illustration of the test can be seen in Figure 3.

6 Conclusion

White-box cryptography has become a popular method to protect cryptographic keys in an insecure software realm potentially controlled by the adversary. All

[6] `https://github.com/UzL-ITS/white-box-masking`

white-box cryptosystems in the literature have been practically broken due to differential computation analysis. Moreover algebraic attacks have shown the inefficiency of classic side-channel countermeasures when they are applied in the white-box setting. Therefore, the need for a secure and reliable white-box implementation protected against both attacks has become evident.

We proposed the first masking scheme that combines linear and non-linear components to achieve resistance against computational and algebraic attacks. The new scheme extended the `ISW` transformation to resist algebraic attacks by increasing the order of the decoding function. It has been defined generic and can be applied to any orders of n and d, however the structure of the nodes should satisfy the algebraic properties. We have examined the implementation cost of our scheme for arbitrary orders of protection and compare it with the `ISW` transformation.

We analyzed the two prevalent security notions in white-box model, *side channel analysis security* and *prediction security*, and underlined the incompatibility of the notions, which reveals that a scheme should satisfy both notions. We used the well-known SCA security notion to prove the $(n-1)^{th}$ order SCA security of an (n, d)-masking scheme and thus we showed that our scheme can resist $(n-1)^{th}$-order computation attacks. We proved first and second order prediction security for the concrete construction of the $(n, 1)$ and $(n, 2)$ masking scheme, respectively. Furthermore, the proposed methodology to prove the algebraic security can be extended to higher orders. We implemented our scheme to the algebraic verification tool to support our results and the code has been made publicly available. Finally, a proof-of-concept AES-128 bitwise implementation was provided to perform leakage detection and extensive performance analysis. The analysis showed that the new combined masking scheme outperforms the previous approaches which requires to combine two different masking schemes to resist both attacks.

References

1. Banik, S., Bogdanov, A., Isobe, T., Jepsen, M.B.: Analysis of software countermeasures for whitebox encryption. IACR Trans. Symmetric Cryptol. **2017**(1), 307–328 (2017)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE **94**(2), 370–382 (2006)
3. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a white box aes implementation. In: Handschuh, H., Hasan, M.A. (eds.) Selected Areas in Cryptography. pp. 227–240. Springer (2005)
4. Biryukov, A., Udovenko, A.: Attacks and countermeasures for white-box designs. In: Peyrin, T., Galbraith, S. (eds.) Advances in Cryptology – ASIACRYPT 2018. pp. 373–402. Springer International Publishing, Cham (2018)
5. Bock, E.A., Bos, J.W., Brzuska, C., Hubain, C., Michiels, W., Mune, C., Gonzalez, E.S., Teuwen, P., Treff, A.: White-box cryptography: don't forget about grey-box attacks. Journal of Cryptology **32**(4), 1095–1143 (2019)
6. Bock, E.A., Brzuska, C., Michiels, W., Treff, A.: On the ineffectiveness of internal encodings-revisiting the dca attack on white-box cryptography. In: International

Conference on Applied Cryptography and Network Security. pp. 103–120. Springer (2018)

7. Bogdanov, A., Rivain, M., Vejre, P.S., Wang, J.: Higher-order dca against standard side-channel countermeasures. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 118–141. Springer (2019)
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) Advances in Cryptology EUROCRYPT'97, Lecture Notes in Computer Science, vol. 1233, pp. 37–51. Springer (1997)
9. Bos, J.W., Hubain, C., Michiels, W., Teuwen, P.: Differential computation analysis: Hiding your white-box designs is not enough. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings. pp. 215–236. Springer (2016)
10. Boyar, J., Peralta, R.: New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191 (2009)
11. Bringer, J., Chabanne, H., Dottax, E.: White box cryptography: Another attempt. IACR Cryptology ePrint Archive **2006**, 468 (2006)
12. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: 27th USENIX Security Symposium (USENIX Security 18). p. 991–1008 (Aug 2018)
13. Chari, S., Jutla, C., Rao, J.R., Rohatgi, P.: A cautionary note regarding evaluation of aes candidates on smart-cards. In: Second Advanced Encryption Standard Candidate Conference. pp. 133–147. Citeseer (1999)
14. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M. (ed.) Advances in Cryptology – CRYPTO 99, Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999)
15. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A white-box des implementation for drm applications. In: Feigenbaum, J. (ed.) Digital Rights Management. pp. 1–15. Springer (2003)
16. Chow, S., Eisen, P., Johnson, H., Van Oorschot, P.C.: White-box cryptography and an aes implementation. In: Nyberg, K., Heys, H. (eds.) Selected Areas in Cryptography. pp. 250–270. Springer (2003)
17. Contest, T.W.: Ches 2017 capture the flag challenge the whibox contest, an ecrypt white-box cryptography competition. `https://whibox-contest.github.io/`
18. Coron, J.S., Greuet, A., Zeitoun, R.: Side-channel masking with pseudo-random generator. Cryptology ePrint Archive, Report 2019/1106 (2019), `https://eprint.iacr.org/2019/1106`
19. De Mulder, Y., Wyseur, B., Preneel, B.: Cryptanalysis of a perturbated white-box aes implementation. In: Gong, G., Gupta, K.C. (eds.) Progress in Cryptology - INDOCRYPT 2010. pp. 292–310. Springer (2010)
20. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Cryptographic Hardware and Embedded Systems CHES 2001. pp. 251–261. Springer (2001)
21. Gemalto: Sentinel® ldk product brief. `https://sentinel.gemalto.com/resources/software/sentinel-ldk-feature-brief/`
22. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Advances in Cryptology–CRYPTO 2014, pp. 444–461. Springer (2014)

23. Gilbert Goodwill, B.J., Jaffe, J., Rohatgi, P., et al.: A testing methodology for side-channel resistance validation. In: NIST non-invasive attack testing workshop (2011)
24. Goubin, L., Paillier, P., Rivain, M., Wang, J.: How to reveal the secrets of an obscure white-box implementation. Journal of Cryptographic Engineering (Apr 2019)
25. Ishai, Y., Kushilevitz, E., Li, X., Ostrovsky, R., Prabhakaran, M., Sahai, A., Zuckerman, D.: Robust pseudorandom generators. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) Automata, Languages, and Programming. pp. 576–588. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
26. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings. pp. 463–481. Springer (2003)
27. Karroumi, M.: Protecting white-box aes with dual ciphers. In: Rhee, K.H., Nyang, D. (eds.) Information Security and Cryptology - ICISC 2010. pp. 278–291. Springer (2010)
28. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 388–397. Springer (1999)
29. Lee, S., Kim, T., Kang, Y.: A masked white-box cryptographic implementation for protecting against differential computation analysis. IEEE Transactions on Information Forensics and Security **13**(10), 2602–2615 (2018)
30. Lepoint, T., Rivain, M., De Mulder, Y., Roelse, P., Preneel, B.: Two attacks on a white-box aes implementation. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) Selected Areas in Cryptography – SAC 2013. pp. 265–285. Springer (2014)
31. Link, H.E., Neumann, W.D.: Clarifying obfuscation: improving the security of white-box des. International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II **1**, 679–684 Vol. 1 (2005)
32. MacWilliams, F.J., Sloane, N.J.A.: The theory of error-correcting codes, vol. 16. Elsevier (1977)
33. Moghimi, A., Irazoqui, G., Eisenbarth, T.: Cachezoom: How sgx amplifies the power of cache attacks. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 69–90. Springer (2017)
34. Moore, C., O'Neill, M., O'Sullivan, E., Doröz, Y., Sunar, B.: Practical homomorphic encryption: A survey. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 2792–2795. IEEE (2014)
35. Nikova, S., Rijmen, V., Schläffer, M.: Secure hardware implementation of non-linear functions in the presence of glitches. In: Information Security and Cryptology–ICISC 2008, pp. 218–234. Springer (2009)
36. Reparaz, O., Meyer, L.D., Bilgin, B., Arribas, V., Nikova, S., Nikov, V., Smart, N.: Capa: The spirit of beaver against physical attacks. Cryptology ePrint Archive, Report 2017/1195 (2017)
37. Rivain, M., Prouff, E.: Provably secure higher-order masking of aes. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings. pp. 413–427. Springer (2010)
38. Rivain, M., Prouff, E.: Provably secure higher-order masking of aes. Cryptology ePrint Archive, Report 2010/441 (2010)
39. Rivain, M., Wang, J.: Analysis and improvement of differential computation attacks against internally-encoded white-box implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(2), 225–255 (Feb 2019)

40. Roche, T., Prouff, E.: Higher-order glitch free implementation of the AES using secure multi-party computation protocols. Journal of Cryptographic Engineering **2**(2), 111–127 (2012)
41. Schneider, T., Moradi, A., Güneysu, T.: ParTI – Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, pp. 302–332. Springer (2016)
42. Seker, O., Fernandez-Rubio, A., Eisenbarth, T., Steinwandt, R.: Extending glitch-free multiparty protocols to resist fault injection attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 394–430 (Aug 2018)
43. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
44. Van Bulck, J., Piessens, F., Strackx, R.: Sgx-step: A practical attack framework for precise enclave execution control. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution. SysTEX'17, Association for Computing Machinery, New York, NY, USA (2017)
45. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against side-channel attacks: A comprehensive study with cautionary note. In: Wang, X., Sako, K. (eds.) Advances in Cryptology – ASIACRYPT 2012. Springer (2012)
46. Wyseur, B., Michiels, W., Gorissen, P., Preneel, B.: Cryptanalysis of white-box des implementations with arbitrary external encodings. In: Adams, C., Miri, A., Wiener, M. (eds.) Selected Areas in Cryptography. pp. 264–277. Springer (2007)
47. Xiao, Y., Lai, X.: A secure implementation of white-box aes. In: 2009 2nd International Conference on Computer Science and its Applications. pp. 1–6 (2009)

A Additional Proofs

In this Appendix, we give proofs for Lemma 1, proof of correctness of our scheme, and Propositions that concern the security features of the gadgets whose proof is not given in the paper.

Proof (Lemma 1: Correctness of Circuit Transformation $T^{(n,d)}$).
For simplicity, let us denote `Encode` as:

$$\texttt{Encode}(x, \tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_{n-1}) = \texttt{Encode}(x).$$

Next we prove the functionality preserving property of each gadget.

– $$\begin{aligned} x &= \texttt{Decode}(\texttt{RefreshMask}(\texttt{Encode}(x)) \\ &= \texttt{Decode}(\texttt{RefreshMask}((\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n)) \\ &= \texttt{Decode}(\tilde{x}_0 \oplus \tilde{r}_0, \ldots, \tilde{x}_d \oplus \tilde{r}_d, x_1 \oplus r_1, \ldots, x_{n-1} \oplus r_{n-1}, x_n \oplus \textstyle\bigoplus_{i=1}^{n-1} r_i \oplus \mathcal{W} \oplus \mathcal{R}) \\ &= (\tilde{x}_0 \oplus \tilde{r}_0) \cdots (\tilde{x}_d \oplus \tilde{r}_d) \oplus x_1 \oplus \cdots \oplus x_n \oplus \mathcal{W} \oplus \mathcal{R} \\ &= \tilde{x}_0 \cdots \tilde{x}_d \oplus \mathcal{W}' \oplus x_1 \oplus \cdots \oplus x_n \oplus \mathcal{W} \oplus \mathcal{R} \\ &= \tilde{x}_0 \cdots \tilde{x}_d \oplus x_1 \oplus \cdots \oplus x_n \\ &= x \end{aligned}$$

– $\texttt{Decode}(\texttt{Xor}(\texttt{Encode}(x), \texttt{Encode}(y)))$
$= \texttt{Decode}(\tilde{x}_0 \oplus \tilde{y}_0, \ldots, \tilde{x}_d \oplus \tilde{y}_d, x_1 \oplus y_1, \ldots, x_{n-1} \oplus y_{n-1}, x_n \oplus y_n \oplus \mathcal{U})$
where $(\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n) = \texttt{RefreshMask}(\texttt{Encode}(x))$ and
$(\tilde{y}_0, \ldots, \tilde{y}_d, y_1, \ldots, y_n) = \texttt{RefreshMask}(\texttt{Encode}(y))$.
$= [(\tilde{x}_0 \oplus \tilde{y}_0) \cdots (\tilde{x}_d \oplus \tilde{y}_d)] \oplus [(x_1 \oplus y_1) \oplus \cdots \oplus (x_{n-1} \oplus y_{n-1}) \oplus (x_n \oplus y_n \oplus \mathcal{U})]$
$= [\tilde{x}_0 \cdots \tilde{x}_d \oplus \mathcal{U} \oplus \tilde{y}_0 \cdots \tilde{y}_d] \oplus [(x_1 \oplus y_1) \oplus \cdots \oplus (x_{n-1} \oplus y_{n-1}) \oplus (x_n \oplus y_n \oplus \mathcal{U})]$
$= (\tilde{x}_0 \cdots \tilde{x}_d \oplus x_1 \oplus \cdots x_n) \oplus (\tilde{y}_0 \cdots \tilde{y}_d \oplus y_1 \oplus \cdots y_n)$
$= x \oplus y.$

– $xy = \texttt{Decode}(\texttt{And}(\texttt{Encode}(x), \texttt{Encode(y)}))$
$= \texttt{Decode}(\texttt{And}((\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n), (\tilde{y}_0, \ldots \tilde{y}_d, y_1, \ldots, y_n)))$
where $(\tilde{x}_0, \ldots, \tilde{x}_d, x_1, \ldots, x_n) = \texttt{RefreshMask}(\texttt{Encode}(x))$ and
$(\tilde{y}_0, \ldots, \tilde{y}_d, y_1, \ldots, y_n) = \texttt{RefreshMask}(\texttt{Encode}(y))$.
$= \texttt{Decode}(\tilde{z}_0, \ldots, \tilde{z}_d, z_1, \ldots, z_n)$

where the output shares can be listed as follows:

$$\tilde{z}_i = \tilde{x}_i \tilde{y}_{i'} \oplus r^{i,1} \oplus \cdots \oplus r^{i,n} \text{ for } 0 \leq i \leq d,$$
$$z_i = x_i y_i \oplus \bigoplus_{\substack{j=1 \\ j \neq i}}^{n} r_{i,j} \text{ for } 1 \leq i \leq n.$$

Also, the values $r_{i,j}$ can be listed as:

$$r_{0,j} = \mathcal{F}(x_j, y_j) = [r_{0,j} \oplus (\tilde{x}_0 \ldots \tilde{x}_d) y_j] \oplus x_j (\tilde{y}_0 \ldots \tilde{y}_d) \text{ for } 1 \leq j \leq n,$$
$$r_{i,j} = (r_{i,j} \oplus x_i y_j) \oplus x_j y_i \text{ for } 1 \leq i < j \leq n.$$

Therefore,

$$\begin{aligned}
\texttt{Decode}(\bar{z}) &= \tilde{z}_0 \cdots \tilde{z}_d \oplus z_1 \oplus \ldots \oplus z_n \\
&= \prod_{i=0}^{d} \left[\tilde{x}_i \tilde{y}_{i'} \oplus r^{i,1} \oplus \cdots \oplus r^{i,n}\right] \oplus \bigoplus_{i=1}^{n} \left[x_i y_i \oplus \bigoplus_{\substack{j=0 \\ j \neq i}}^{n} r_{i,j} \right] \\
&= \left[(\tilde{x}_0 \cdots \tilde{x}_d)(\tilde{y}_0 \cdots \tilde{y}_d) \oplus \mathcal{V}\right] \oplus \left[\bigoplus_{i=1}^{n} (x_i y_i \oplus \bigoplus_{\substack{j=1 \\ j \neq i}}^{n} r_{i,j}) \right] \oplus \\
&\left[\bigoplus_{j=1}^{n} ((r_{0,j} \oplus (\tilde{x}_0 \ldots \tilde{x}_d) y_i) \oplus x_i (\tilde{y}_0 \ldots \tilde{y}_d)) \right] \\
&= \left[(\tilde{x}_0 \cdots \tilde{x}_d)(\tilde{y}_0 \cdots \tilde{y}_d) \oplus \mathcal{V}\right] \oplus \left[\bigoplus_{1 \leq i,j \leq n} x_i y_j \right] \oplus \\
&\left[\mathcal{V} \oplus \bigoplus_{i=1}^{n} (\tilde{x}_0 \ldots \tilde{x}_d) y_i \oplus x_i (\tilde{y}_0 \ldots \tilde{y}_d) \right] \\
&= (\tilde{x}_0 \cdots \tilde{x}_d)(\tilde{y}_0 \cdots \tilde{y}_d) \oplus \bigoplus_{1 \leq i,j \leq n} x_i y_j \oplus \bigoplus_{i=1}^{n} ((\tilde{x}_0 \ldots \tilde{x}_d) y_i \oplus x_i (\tilde{y}_0 \ldots \tilde{y}_d)) \\
&= (\tilde{x}_0 \cdots \tilde{x}_d \oplus x_1 \oplus \cdots x_n)(\tilde{y}_0 \cdots \tilde{y}_d \oplus y_1 \oplus \cdots y_n) \\
&= xy.
\end{aligned}$$

Hence we showed that the gadgets introduced in Section 3 are functionally preserving gadgets. Therefore, the transformation that generate an (n, d)-masked circuit is a functionally preserving transformation.

Proof (Proposition 3: $(n-1)^{th}$ order SCA Security of $\texttt{Xor}[n, d]$ gadget). In order to prove the proposition, we show that every set of intermediate variables with $\leq n-1$ elements can be simulated by two sets of input shares $(\tilde{x}_i)_{i \in \tilde{I}}$ and $(x_i)_{i \in I}$ such that $|\tilde{I}| \leq d+1$ and $|I| \leq n-1$ (resp. $(\tilde{y}_j)_{j \in \tilde{J}}$ and $(y_j)_{j \in J}$ such that $|\tilde{J}| \leq d+1$ and $|J| \leq n-1$). We denote the concatenations of these tuples by $U = ((\tilde{x}_i)_{i \in \tilde{I}}, (x_i)_{i \in I})$ and $V = ((\tilde{y}_j)_{j \in \tilde{J}}, (y_j)_{j \in J})$.

We first need to construct the sets of indices I and $\tilde{I}$ corresponding to shares of x, and J and $\tilde{J}$ corresponding to shares of y.

– For all x_i, y_i, $x_i \oplus y_i$ (resp. $\tilde{x}_i$, $\tilde{y}_i$, $\tilde{x}_i \oplus \tilde{y}_i$) add i to I and J (resp. $\tilde{I}$ and $\tilde{J}$).
– For all $\tilde{x}_i \tilde{y}_j$ add i to $\tilde{I}$ and j to $\tilde{J}$.
– For all $\prod_{i \in K} \tilde{x}_i \prod_{j \notin K} \tilde{y}_j$ where $K \subsetneq \{0, \ldots, d\}$ add all $i \in K$ to $\tilde{I}$ and add all $j \notin K$ to $\tilde{J}$.

According to our selection, we add at most one index to I (resp. J) and in the worst case $d+1$ elements to $\tilde{I}$ (resp. $\tilde{J}$).

Clearly, every variable of the form x_i, y_i, $x_i \oplus y_i$ (resp. $\tilde{x}_i$, $\tilde{y}_i$, $\tilde{x}_i \oplus \tilde{y}_i$) can be simulated by the sets U and V. Moreover every variable of the form $\prod_{i\in K} \tilde{x}_i \prod_{j\notin K} \tilde{y}_j$ where $K \subsetneq \{0,\ldots,d\}$ can be simulated according to our selection since even in the worst case scenario $|\tilde{I}| = |\tilde{J}| = d+1$.

Hence, we show that any set of intermediate variables with $\leq n-1$ elements can be simulated by $U = ((\tilde{x}_i)_{i\in\tilde{I}}, (x_i)_{i\in I})$ and $V = ((\tilde{y}_j)_{j\in\tilde{J}}, (y_j)_{j\in J})$ such that $|\tilde{I}|, |\tilde{J}| \leq d+1$ and $|I|, |J| \leq n-1$. By the definition of our masking, U and V are uniformly distributed and independent of any sensitive variable and hence the `Xor` gadget seen in Algorithm 1 is an $(n-1)^{th}$ SCA secure gadget.

Proof (Proposition 4: ϵ-1-AS of ***RefreshMask***$[n,1]$ *Gadget).* In the first part of the proof, we show that there exists no function $f \in \mathcal{F}^{(1)}(C)$ such that f is constant when inputs are fixed. Assume that there exists a function $f \in \mathcal{F}^{(1)}(C)$ such that f is constant when the inputs $(\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n})$ are fixed. As seen in Algorithm 3 the only nodes that does not contain a random (i.e. not fixed) can be found in line 11 where the values $\mathcal{W}$ and $\mathcal{R}$ are processed. By the definition of $\mathcal{W}$ each input is accompanied by a random value. And $\mathcal{R}$ contains only random values. Therefore each each node is accompanied by a random node and any linear combination of these nodes cannot be constant. Hence there exists no constant function $f \in \mathcal{F}^{(1)}(C)$ such that inputs are fixed.

In the second part, we examine the highest degree term in the gadget. The maximum degree term can be found in $\mathcal{R}$ with degree 2. Therefore the corresponding bias and the bias bound of the gadget can be calculated as 2^{-2} and $\epsilon \leq \left|1/2 - 1/2^2\right| = 1/4$ respectively. Thus `RefreshMask` gadget is ϵ-1-AS with $\varepsilon \leq 1/4$.

Proof (Proposition 6: ϵ-2-AS of ***RefreshMask***$[n,2]$ *Gadget).*

The first part of the proof follows the same structure of the proof of Proposition 7, and the same circuit extension idea used in the proof of Proposition 7. We reformulate the circuit C as follows:

$$C\colon ((\mathbb{F}_2^{n+3} \times \mathbb{F}_2^{n+3}), \mathbb{F}_2^{R_C}) \to \mathbb{F}_2^{n+3}$$
$$((\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, (x_i)_{1\leq i\leq n}), (\tilde{y}_0, \tilde{y}_1, \tilde{y}_2, (y_i)_{1\leq i\leq n}), \bar{r}) \mapsto (\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, (z_i)_{1\leq i\leq n}).$$

By the Lemma 3 we can define the second order extension of the circuit C as follows:

$$C' = \{v_i\}_{1\leq i\leq M} \cup \{v_i v_j\}_{1\leq i,j\leq M} \text{ where } |C| = M.$$

Assume that there exists a function $f \in \mathcal{F}^{(1)}(C)$ such that f is constant when the inputs $(\tilde{x}_0, \tilde{x}_1, (x_i)_{1\leq i\leq n})$ are fixed. As seen in Algorithm 3 the only nodes that does not contain a random (i.e. not fixed) can be found in line 11 where the values $\mathcal{W}$ and $\mathcal{R}$ are processed. By the definition of $\mathcal{W}$ each input is accompanied by a random value. And $\mathcal{R}$ contains only random values. Since each each node is accompanied by a random node any linear combination of the nodes in C' cannot be constant. This is followed by there exists no constant function $f \in \mathcal{F}^{(2)}(C)$ by Lemma 3.

In the second part, we examine the highest degree term in the gadget. The maximum degree term can be found in $\mathcal{R}$ with degree 3. Therefore the corresponding second order-bias and the bias bound of the gadget can be calculated as 2^{-6} and $\epsilon \leq \left|1/2 - 1/2^6\right| = 31/64$ respectively. Thus `RefreshMask` gadget is ϵ-2-AS with $\varepsilon \leq 31/64$.

B Example Constructions

Example 3. $n = 2$, $d = 1$

Here is an example construction for $(2,1)$-masking scheme:

- `Encode`$(x, x_1, \tilde{x}_0, \tilde{x}_1) = (\tilde{x}_0, \tilde{x}_1, x_1, x_2)$ where $x_2 = \tilde{x}_0\tilde{x}_1 \oplus x_1 \oplus x$.
- `Decode`$(\overline{x}) = \tilde{x}_0\tilde{x}_1 \oplus x_1 \oplus x_2$.
- `Xor`$(\overline{x}, \overline{y}) = (\tilde{z}_0, \tilde{z}_1, z_1, z_2)$ such that $z = x \oplus y$:
  - $\tilde{z}_0 = \tilde{x}_0 \oplus \tilde{y}_0$,
  - $\tilde{z}_1 = \tilde{x}_1 \oplus \tilde{y}_1$,
  - $z_1 = x_1 \oplus y_1$,
  - $z_2 = x_2 \oplus y_2 \oplus \tilde{x}_1\tilde{y}_0 \oplus \tilde{x}_0\tilde{y}_1$.
- `And`$(\overline{x}, \overline{y}) = (\tilde{z}_0, \tilde{z}_1, z_1, z_2)$ such that $z = xy$;

  Step-1: First, calculate the multiplicative representations of the output share z_0:
  - $\tilde{z}_0 = x_0y_1 \oplus r^{0,1} \oplus r^{0,2}$,
  - $\tilde{z}_1 = x_1y_0 \oplus r^{1,1} \oplus r^{1,2}$ where $(r^{0,1}, r^{0,2}, r^{1,1}, r^{1,2}) \leftarrow rand(0,1)$

  Step-2(a): Calculate the intermediate values $r_{j,0}$ which include the reconstruction of the values x_0 and y_0:
  - $r_{1,0} = \tilde{x}_1(\tilde{x}_0y_1 \oplus r^{0,1}\tilde{y}_0) \oplus \tilde{y}_1(\tilde{y}_0x_1 \oplus r^{1,1}\tilde{x}_0) \oplus r^{1,1}(r^{0,1} \oplus r^{0,2})$,
  - $r_{2,0} = \tilde{x}_1(\tilde{x}_0y_2 \oplus r^{0,2}\tilde{y}_0) \oplus \tilde{y}_1(\tilde{y}_0x_2 \oplus r^{1,2}\tilde{x}_0) \oplus r^{1,2}(r^{0,1} \oplus r^{0,2})$.

  Step-2(b): Calculate the intermediate values $r_{j,0}$ which do not include the reconstruction of the values x_0 and y_0:
  - $r_{1,2} \leftarrow$ `rand`$(0,1)$,
  - $r_{2,1} = (r_{1,2} \oplus x_1y_2) \oplus x_2y_1$.

  Step-3: Finally, calculate the rest of the shares:
  - $z_1 = x_1y_1 \oplus r_{1,0} \oplus r_{1,2}$,
  - $z_2 = x_2y_2 \oplus r_{2,0} \oplus r_{2,1}$.
- `RefreshMask`$(\overline{x}) = (\tilde{x}_0, \tilde{x}_1, x_1, x_2)$
  1. First, calculate the non-linear components of the output share x_0:
     - $\tilde{x}_0 = \tilde{x}_0 \oplus \tilde{r}_0$,
     - $\tilde{x}_1 = \tilde{x}_1 \oplus \tilde{r}_1$ where $(\tilde{r}_0, \tilde{r}_1) \leftarrow rand(0,1)$
  2. Calculate the rest the linear masks:
     - $x_1 = x_1 \oplus r_1$,
     - $x_2 = x_2 \oplus r_1$ where $r_1 \leftarrow rand(0,1)$
  3. Select a random bit $r_0 \leftarrow rand(0,1)$ and calculate the intermediate variable with $\mathcal{W}$ and $\mathcal{R}$:
     - $\mathcal{W} = \tilde{r}_0(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_1(\tilde{x}_0 \oplus r_0)$ and $\mathcal{R} = (\tilde{r}_0 \oplus r_0)(\tilde{r}_1 \oplus r_0) \oplus r_0$ where

- $x_2 = x_2 \oplus \mathcal{W} \oplus \mathcal{R}$

Example 4. Example: $n = 2$, $d = 2$

Here is an example construction for $(2, 2)$-masking scheme:

- `Encode`$(x, x_1, \tilde{x}_0, \tilde{x}_1, \tilde{x}_2) = (\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, x_1, x_2)$ where $x_2 = \tilde{x}_0\tilde{x}_1\tilde{x}_2 \oplus x_1 \oplus x$.
- `Decode`$(\overline{x}) = \tilde{x}_0\tilde{x}_1\tilde{x}_2 \oplus x_1 \oplus x_2$.
- `Xor`$(\overline{x}, \overline{y}) = (\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, z_1, z_2)$ such that $z = x \oplus y$
  - $\tilde{z}_i = \tilde{x}_i \oplus \tilde{y}_i$ for $i = \{0, 1, 2\}$
  - $z_1 = x_1 \oplus y_1$
  - $z_2 = x_2 \oplus y_2 \oplus \tilde{x}_1(\tilde{x}_2\tilde{y}_0 \oplus \tilde{y}_2(\tilde{x}_0 \oplus \tilde{y}_0)) \oplus \tilde{y}_1(\tilde{x}_2\tilde{y}_0 \oplus \tilde{x}_0(\tilde{x}_2 \oplus \tilde{y}_2))$
- `And`$(\overline{x}, \overline{y}) = (\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, z_1, z_2)$ such that $z = xy$

  Step-1: First, calculate the multiplicative representations of the output share z_0:
  - $\tilde{z}_0 = \tilde{x}_0\tilde{y}_1 \oplus r^{0,1} \oplus r^{0,2}$,
  - $\tilde{z}_1 = \tilde{x}_1\tilde{y}_2 \oplus r^{1,1} \oplus r^{1,2}$,
  - $\tilde{z}_2 = \tilde{x}_2\tilde{y}_0 \oplus r^{2,1} \oplus r^{2,2}$ where $r^{i,j} \leftarrow rand(0, 1)$ for $i = \{0, 1, 2\}$, $j = \{1, 2\}$.

  Step-2(a): Calculate the intermediate values $r_{j,0}$ where the combination of random nodes are defined as; $u = (r^{1,1} \oplus r^{1,2})$ and $v = (r^{2,1} \oplus r^{2,2})$.
  - $r_{1,0} = \mathcal{F}(x_1, y_1) = \tilde{x}_0 \left[\tilde{x}_2(\tilde{x}_1 y_1 \oplus r^{0,1}\tilde{y}_0) \oplus r^{1,1}v\tilde{y}_1\right] \oplus$
    $\tilde{y}_0 \left[\tilde{y}_1(\tilde{y}_2 x_1 \oplus r^{1,1}\tilde{x}_2) \oplus r^{0,1}u\tilde{x}_2\right] \oplus$
    $\tilde{x}_0\tilde{y}_1(r^{1,1}\tilde{x}_2\tilde{y}_0 \oplus r^{2,1}\tilde{x}_1\tilde{y}_2) \oplus r^{0,1}\tilde{x}_1\tilde{y}_2(v \oplus \tilde{x}_2\tilde{y}_0) \oplus$
    $\tilde{x}_2\tilde{y}_0(r^{0,1}\tilde{x}_0 \oplus r^{1,1}\tilde{y}_1) \oplus uvr^{0,1}$.
  - $r_{2,0} = \mathcal{F}(x_2, y_2) = \tilde{x}_0 \left[\tilde{x}_2(\tilde{x}_1 y_2 \oplus r^{0,2}\tilde{y}_0) \oplus r^{1,2}v\tilde{y}_1\right] \oplus$
    $\tilde{y}_0 \left[\tilde{y}_1(\tilde{y}_2 x_2 \oplus r^{1,2}\tilde{x}_2) \oplus r^{0,2}u\tilde{x}_2\right] \oplus$
    $\tilde{x}_0\tilde{y}_1(r^{1,2}\tilde{x}_2\tilde{y}_0 \oplus r^{2,2}\tilde{x}_1\tilde{y}_2) \oplus r^{0,2}\tilde{x}_1\tilde{y}_2(v \oplus \tilde{x}_2\tilde{y}_0) \oplus$
    $\tilde{x}_2\tilde{y}_0(r^{0,2}\tilde{x}_0 \oplus r^{1,2}\tilde{y}_1) \oplus uvr^{0,2}$.

  Step-2(b): Calculate the intermediate values $r_{j,0}$ which do not include the reconstruction of the values x_0 and y_0:
  - $r_{1,2} \leftarrow \texttt{rand}(0, 1)$,
  - $r_{2,1} = (r_{1,2} \oplus x_1 y_2) \oplus x_2 y_1$.

  Step-3: Finally, calculate the rest of the shares:
  - $z_1 = x_1 y_1 \oplus r_{1,0} \oplus r_{1,2}$,
  - $z_2 = x_2 y_2 \oplus r_{2,0} \oplus r_{2,1}$.
- `RefreshMask`$(\overline{x}) = (\tilde{x}_0, \tilde{x}_1, \tilde{x}_2, x_1, x_2)$
  1. First, calculate the multiplicative representations of the output share x_0:
     - $\tilde{x}_0 = \tilde{x}_0 \oplus \tilde{r}_0$,
     - $\tilde{x}_1 = \tilde{x}_1 \oplus \tilde{r}_1$,
     - $\tilde{x}_2 = \tilde{x}_2 \oplus \tilde{r}_2$, where $(\tilde{r}_0, \tilde{r}_1, \tilde{r}_2) \leftarrow rand(0, 1)$
  2. Calculate the rest the linear shares:
     - $x_1 = x_1 \oplus r_1$,
     - $x_2 = x_2 \oplus r_1$ where $r_1 \leftarrow rand(0, 1)$

3. Select a random bit $r_0 \leftarrow rand(0,1)$ and calculate the intermediate variable with $\mathcal{W}$ and $\mathcal{R}$:
   - $\mathcal{W} = \tilde{r}_1\tilde{r}_2(\tilde{x}_0 \oplus r_0) \oplus \tilde{r}_0\tilde{r}_2(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_0\tilde{r}_1(\tilde{x}_2 \oplus r_0)\oplus$
     $\tilde{r}_2(\tilde{x}_0 \oplus r_0)(\tilde{x}_1 \oplus r_0) \oplus \tilde{r}_1(\tilde{x}_0 \oplus r_0)(\tilde{x}_2 \oplus r_0) \oplus \tilde{r}_0(\tilde{x}_1 \oplus r_0)(\tilde{x}_2 \oplus r_0),$
     $\mathcal{R} = (\tilde{r}_0 \oplus r_0)(\tilde{r}_1 \oplus r_0)(\tilde{r}_2 \oplus r_0)$
     $\tilde{r}_2 r_0(\tilde{x}_0 \oplus \tilde{x}_1) \oplus \tilde{r}_1 r_0(\tilde{x}_0 \oplus \tilde{x}_2) \oplus \tilde{r}_0 r_0(\tilde{x}_1 \oplus \tilde{x}_2).$
   - $x_2 = x_2 \oplus \mathcal{W} \oplus \mathcal{R}$