IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 286–309. DOI:10.46586/tches.v2023.i2.286-309

Efficient Private Circuits with Precomputation

Weijia Wang[1,2,3], Fanjie Ji[1], Juelin Zhang[1] and Yu Yu[4,5,6]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
wjwang@sdu.edu.cn
[2] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, China
[3] Quan Cheng Shandong Laboratory, Jinan, China
[4] Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China yuyu@yuyu.hk
[5] Shanghai Qi Zhi Institute, Shanghai, China
[6] Shanghai Key Laboratory of Privacy-Preserving Computation, Shanghai, China

Abstract. At CHES 2022, Wang et al. described a new paradigm for masked implementations using private circuits, where most intermediates can be precomputed before the input shares are accessed, significantly accelerating the online execution of masked functions. However, the masking scheme they proposed mainly featured (and was designed for) the cost amortization, leaving its (limited) suitability in the above precomputation-based paradigm just as a bonus. This paper aims to provide an efficient, reliable, easy-to-use, and precomputation-compatible masking scheme. We propose a new masked multiplication over the finite field $\mathbb{F}_q$ suitable for the precomputation, and prove its security in the composable notion called Probing-Isolating Non-Inference (PINI). Particularly, the operations (e.g., AND and XOR) in the binary field can be achieved by assigning $q = 2$, allowing the bitsliced implementation that has been shown to be quite efficient for the software implementations. The new masking scheme is applied to leverage the masking of AES and `SKINNY` block ciphers on ARM Cortex M architecture. The performance results show that the new scheme contributes to a significant speed-up compared with the state-of-the-art implementations. For `SKINNY` with block size 64, the speed and RAM requirement can be significantly improved (saving around 45% cycles in the online-computation and 60% RAM space for precomputed values) from AES-128, thanks to its smaller number of AND gates. Besides the security proof by hand, we provide formal verifications for the multiplication and T-test evaluations for the masked implementations of AES and `SKINNY`. Because of the structure of the new masked multiplication, our formal verification can be performed for security orders up to 16.

Keywords: Side-Channel Attack · Masking · Precomputation · Bitsliced Implementation · Formal Verification

1 Introduction

Side-channel attacks are able to extract secrets from a cryptographic device using leakages such as power consumption and electromagnetic radiation. A masking scheme, sometimes called a *private circuit compiler* is a well-known countermeasure against side-channel attacks. It can be regarded as a compiler to compile a cryptographic algorithm into the masked implementation. Concretely, it randomly splits each secret-dependent variable into $d + 1$ shares and transforms every elementary operation into the masked correspondence called *gadgets*. The basic requirement of the masked implementation, called *d-private*

Licensed under Creative Commons License CC-BY 4.0.
Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

security or *d-probing security*, is that the joint distribution of any d intermediate variables is independent of the secret.

Though it can provide provable and configurable protection against the side-channel attack, higher-order masking usually brings about a significant overhead. For example, the computational complexity of the well-known scheme proposed by Ishai, Sahai, and Wagner (a.k.a., the ISW scheme) [ISW03] and its numerous variants exhibit quadratic growth in the security order d. It is still challenging to apply the masking scheme in practice, especially in the resource-constraint environment.

In this paper, we investigate the masking scheme with precomputation (a.k.a, preprocessing) that has been widely applied in the field of secure multi-party computation, for example, [BDOZ11, DPSZ12]. The calculation is split into two phases: precomputation and online-computation. The precomputation randomly computes some precomputed values (required to be stored in the memory) independent of the input shares. We can regard the precomputation as a *one-time obfuscation* that transforms the cryptographic algorithm into a *one-time program* before each time of running. The online-computation takes the input shares and precomputed values and calculates the output shares more efficiently (e.g., exhibits a linear growth in the security order) than the case without precomputation. Wang et al. [WGY$^+$22] illustrated that the masking scheme with precomputation could enjoy a vast range of applications. They also present a challenge-response authentication protocol as a typical example, where Alice sends a random value to Bob as the challenge, and Bob encrypts the random value and sends the ciphertext back to Alice. At last, Alice checks if the decryption of the ciphertext produces the original random value. The precomputation of masked encryption and decryption can be performed during the idle time of Alice and Bob. Known schemes compatible with precomputation fall into two categories that we recall as follows.

Table-Based Masking. It can generate all masked tables (e.g., S-boxes) in precomputation. Its main merit is the ability to mask any look-up tables in cryptographic algorithms. On the downside, this approach requires a large RAM space since one has to generate different masked tables for different look-up tables. The table-based masking can be traced back to the first-order scheme proposed by Chari et al. in [CJRR99]. Then, second-order schemes were proposed by Schramm et al. in [SP06] and Rivain et al. in [RDP08]. Coron et al. proposed the first higher-order table-based masking [Cor14]. Their scheme requires $\approx 2^k k'(2d+1)$ bits of RAM for each (k,k')-table, and thus a cipher with ℓ (k,k')-S-boxes requires $\approx 2^k k'(2d+1)\ell$ bits of RAM [1]. An improved scheme that can save the RAM overhead by a factor of 2 is given in [CRZ18]. At CHES 2021, Valiveti et al. [VV21] proposed a method that can reduce the RAM size to $2^{k+1}k\ell + O\big(k'k(d+1)^2\big)$ bits by applying the technique of masking with pseudorandom generators (PRGs) [CGZ20]. However, the complexity of online-computation of this method grows to $\tilde{O}(\ell d^2)$, due to the running of PRGs.

Circuit-based Masking. Wang et al. [WGY$^+$22] presented at CHES 2022 a circuit-based approach that can be regarded as the other category. They provided new multiplication and addition gadgets that are compatible with the precomputation-based paradigm. That is, the precomputation calculates many intermediate values (that can be designed to be independent of the input shares) of the gadget, and the online-computation can evaluate the gadgets more efficiently. However, the scheme was specifically designed for the cost amortization across different gadgets, and its suitability for precomputed paradigm was only regarded as a bonus. Concretely, the scheme can be adopted in a way that enjoys cost amortization at the cost that all the operations are in the finite field, where a large Maximum Distance Separable (MDS) matrix is required. Or, one can abandon the cost amortization but requires many calls of refreshing and a large amount of randomness. In

[1] One can generate masked tables on-the-fly, and thus only $2^{k+1}k'(2d+1)$ bits are sufficient for a full cipher. But this strategy is obviously not applicable to the precomputation-based paradigm.

addition, the multiplication over the finite field is not directly supported in microprocessors and is still time-consuming, even using look-up tables. At the same time, the bitsliced implementation has been shown to be quite efficient for ciphers in software. For example, Goudarzi et al. [GR17] compared the masked AES of bitsliced implementation with those using field operations in software, showing that the former performs better than the latter.

1.1 Our Contributions

In this paper, we follow the line of work on the circuit-based masking with precomputation. We propose a masked multiplication algorithm over $\mathbb{F}_q$ suitable to the precomputation-based paradigm, where $q = p^m$ with any prime p and integer m. It requires $(d+1)d/2$ random elements in $\mathbb{F}_q$ (whereas the masked multiplication in [WGY+22] requires $3d^2$ elements). Particularly, operations in the binary field can be achieved by assigning $q = 2$. This allows the bitsliced implementation that has been shown to be quite efficient for software implementations. Our new algorithm is secure in the composable security notion called Probe-Isolating Non-Inference (PINI) that enables trivial composition with trivial masked linear operations (e,g., addition). Then, we describe the precomputation and online-computation of our scheme. Particularly, for a circuit consisting of ℓ_{mul} multiplications and ℓ_{add} linear operations, the precomputation produces at most $\approx 3d\ell_{\text{mul}}$ precomputed elements with computational complexity $O(d^2\ell_{\text{mul}} \log^2 q + d\ell_{\text{add}} \log q)$, and the online-computation runs in $O(d\ell_{\text{mul}} \log^2 q + \ell_{\text{add}} \log q)$. We also make a comparison of operation counts between the quasilinear-complexity masking scheme [GJR18] and ours .

We apply our countermeasure to the block ciphers AES-128 and SKINNY, and provide masked implementations on the ARM Cortex M architecture. The performance results show that the new masked implementation gains a significant speed-up beyond the state-of-the-art ones. Moreover, compared to the AES and SKINNY variant with block size 128, we exhibit that the smaller number of AND operations in SKINNY variant with block size 64 not only benefits the fast speed of the online-computation, but also significantly reduces the size of RAM for precomputed values.

Thanks to its structure, the new masked multiplication has fewer variables (i.e., $O(d)$) to be checked in the formal verification. This significantly reduces its verification time and provides the formal verification for security orders up to 16. While, to the best of our knowledge, the known maximum security order that can be checked for a multiplication gadget is 10 [BK21]. We also provide a verification using the existing tool SILVER [KSM20]. At last, a T-test evaluation for our implementation is conducted to validate the security order in practice. The source code in this paper is available on `https://github.com/wjwangcrypto/maskingwithprecomputation`.

1.2 Organization

We first present notations and backgrounds in Section 2. We describe the new masking scheme and provide security proof and formal verification in Section 3. Section 4 presents the applications to AES and SKINNY, and shows their T-test evaluations. Finally, Section 5 concludes the paper.

2 Preliminaries

2.1 Notations

In this paper, we use $\mathbb{F}_q$ to denote a finite field with $q = p^m$ for any prime p and positive integer m[2]. An element in $\mathbb{F}_q$ is denoted by a lowercase letter. We use $\oplus$ and $\ominus$ to denote

[2]Our scheme is applied for $\mathbb{F}_q$, but in the applications, we are mainly interested in the binary numbers ($q = 2$) for bitsliced implementations case as it is known to provide the best implementation performance.

addition and subtraction over $\mathbb{F}_q$ respectively, and use $\odot$ to denote the field multiplication. Particularly, for any field with characteristic two (i.e., $p = 2$), $\oplus$ and $\ominus$ are identical. For $x, y \in \mathbb{F}_q$, we usually abbreviate $x \odot y$ as xy. Let calligraphies (e.g., $\mathcal{I}$) be sets, and $|\mathcal{I}|$ denote the cardinality of the set $\mathcal{I}$. For any integers $i \leq j$, we denote $[i : j]$ the set of integers $\{i, \ldots, j\}$, and for any $x_i, \ldots, x_j \in \mathbb{F}_q$, we denote $x_{i:j} \overset{\text{def}}{=} \{x_i, \ldots, x_j\}$, and denote $\bigoplus_{i'=i}^{j} x_{i'} \overset{\text{def}}{=} x_i \oplus \ldots \oplus x_j$. A set of variables (say, $\mathcal{X}$) in $\mathbb{F}_q$ are independently distributed of the other set of variables (say, $\mathcal{Y}$), if $\Pr(\mathcal{X} = \alpha, \mathcal{Y} = \beta) = \Pr(\mathcal{X} = \alpha)\Pr(\mathcal{Y} = \beta)$ for any value α of $\mathcal{X}$ and any value β of $\mathcal{Y}$.

2.2 Private Circuits

We typically represent a circuit manipulating sensitive variables in $\mathbb{F}_q$ as a sequence of operations (i.e., additions, linear transformations, and multiplications). A randomized circuit is a circuit involving random variables. Variables inside a circuit (including inputs and outputs) are usually called intermediate variables. A probe to a circuit is an intermediate variable assumed to be leaked. We denote $\mathcal{Y} = \mathsf{C}(\mathcal{X})$ as the functionality of circuit C taking a set $\mathcal{X}$ of variables as input and returning a set $\mathcal{Y}$ of variables. Similarly, $\mathsf{C}_{\mathcal{P}}(\mathcal{X})$ returns the values of probes $\mathcal{P}$ with input $\mathcal{X}$.

Definition 1 (Private circuit compiler [ISW03])**.** A private circuit compiler for a circuit C with input in $\mathbb{F}_q^n$ and output in $\mathbb{F}_q^{n'}$ is defined by a triple $(\mathsf{I}, \mathsf{T}, \mathsf{O})$ where

- $\mathsf{I} : \mathbb{F}_q \to \mathbb{F}_q^{d+1}$, is an encoder that randomly maps each input $x \in \mathbb{F}_q$ to a sharing consisting of shares $x_{1:d+1}$ such that $\bigoplus_{i=1}^{d+1} x_i = x$. For a share x_i, we call i the index of x_i.
- T is a circuit transformation whose input is circuit C, and output is a randomized circuit C' with $n \times (d + 1)$ shares as the input, and $n' \times (d + 1)$ shares as the output.
- $\mathsf{O} : \mathbb{F}_q^{d+1} \to \mathbb{F}_q$, is a decoder that maps output shares $z_{1:d+1}$ to the corresponding output $z \in \mathbb{F}_q$, i.e., $z \leftarrow \bigoplus_{i=1}^{d+1} z_i = z$.

We say that $(\mathsf{I}, \mathsf{T}, \mathsf{O})$ is a private circuit compiler and C' is a *d-private circuit* (or *d-probing secure*) if the following requirements hold:

- Correctness: for any input $\mathcal{X} \in \mathbb{F}_q^n$, $\mathsf{O}^\circ\big(\mathsf{C}'(\mathsf{I}^\circ(\mathcal{X}))\big) = \mathsf{C}\big(\mathcal{X}\big)$, where I° (resp., O°) is a canonical encoder (resp., decoder) that encodes (resp., decodes) each element of input secrets $\mathcal{X}$ (resp., each set of $d + 1$ shares of $n' \times (d + 1)$ output shares) by repeatedly calling I (resp., O).
- Privacy: for any input $\mathcal{X}$ and any set of probes $\mathcal{P}$ such that $|\mathcal{P}| \leq d$ and $\mathsf{C}'_{\mathcal{P}}\big(\mathsf{I}(\mathcal{X})\big)$ are independent of the input $\mathcal{X}$, where d is called the security order.

The circuit transformation T is realized by independently transforming every gate into their masked correspondence called gadget (usually denoted as G). A gadget is defined as a circuit whose inputs and outputs are sharings. Note that, the composed gadget is a gadget, and thus a recursive composition of gadgets is also a gadget.

2.3 Composable Security Notions

The naive method of proving the probing security is to enumerate the probes within the masked circuit, and check if the distribution of every tuple of probes is independent of the secret input. It makes the complexity of the proof grow exponentially with the circuit size. The natural (but flawed) solution is to prove the probing security of each small gadget, and expect the composition of the gadgets to be still secure.

However, the probing security of each small gadget does not imply the probing security of the composition. Fortunately, it has been shown by Barthe et al. [BBD+16] that there exist composable security notions supporting the deduction from the security of small gadgets to the composed one. Then, Cassiers et al. [CS20] proposed a new composable security notion called Probing-Isolating Non-Inference (PINI) that supports a more trivial composition, which will be used in this paper.

We first describe the definition of simulatability introduced in [BBP+16].

Definition 2 (Simulatability [BBP+16])**.** Let $\mathcal{P}$ be a set of probes of a gadget G with input shares $\mathcal{X}$. Let $\mathcal{S} \subseteq \mathcal{X}$ be a subset of input shares. A simulator is a randomized function $\mathsf{Sim}\colon \mathbb{F}_q^{|\mathcal{S}|} \to \mathbb{F}_q^{|\mathcal{P}|}$. Probes $\mathcal{P}$ can be simulated with input shares $\mathcal{S}$ if and only if there exists a simulator Sim such that for any input shares $\mathcal{X}$, the distributions of $\mathsf{G}_{\mathcal{P}}(\mathcal{X})$ and $\mathsf{Sim}(\mathcal{S})$ are identical.

Then, we give the definition of PINI as follows. The internal probes are probes excluding output shares, and the output probes are probes of output shares.

Definition 3 (PINI [CS20])**.** Let G be a gadget over $d+1$ shares, and its input and output sharings are $x^{(1)}_{1:d+1}, \ldots, x^{(n)}_{1:d+1}$ and $z^{(1)}_{1:d+1}, \ldots, z^{(n')}_{1:d+1}$ respectively. Let $\mathcal{P}_{int}$ be a set of t_{int} internal probes to G and $\mathcal{O} \subseteq [1:d+1]$ be a set of t_{out} indices. G is PINI if and only if for any $\mathcal{P}_{int}$ and $\mathcal{O}$ such that $t_{int} + t_{out} \leq d$, there exists a set $\mathcal{I} \subseteq [1:d+1]$ of at most t_{int} indices such that probes in $z^{(1)}_{\mathcal{O}} \cup \ldots \cup z^{(n')}_{\mathcal{O}} \cup \mathcal{P}_{int}$ can be simulated with shares in $x^{(1)}_{\mathcal{I}\cup\mathcal{O}} \cup \ldots \cup x^{(n)}_{\mathcal{I}\cup\mathcal{O}}$.

Lemma 1 bridges PINI to the probing security.

Lemma 1 (PINI implies probing security [CS20])**.** *A PINI gadget is d-probing secure if any d input shares are independently distributed of the secret input.*

Lemma 2 describes the trivial composability of PINI.

Lemma 2 (Composability of PINI [CS20])**.** *Any composition of PINI gadgets is PINI.*

The above lemmas allow our work to investigate PINI gadgets (with precomputation) for different operations and leave the rest to the trivial composition. Another famous composable security notion supporting trivial composition is Strong Non-Inference (SNI) proposed by Barthe et al. [BBD+16]. However, the masked linear operations (see the definition in Section 2.4) are PINI but not SNI. This makes a PINI gadget to be more useful for composition with trivial implementations. Besides, it is usually more difficult to construct a PINI multiplication. For example, the ISW multiplication [ISW03] is SNI but not PINI. Thus, we only consider the security in PINI.

2.4 Different Types of Gadgets

In this subsection, we discuss the types of gadgets that are necessary to protect a cryptographic algorithm.

Gadget 1 presents the addition gadget that implements a linear function in the masked domain. We denote such a gadget as *the trivial masked addition*, since it can be correctly constructed by adopting additions on the shares with the same index independently. Gadget 2 presents the gadget that implements the linear transformation in masked domain. In this paper, we particularly consider the linear transformation $\mathsf{L} : \mathbb{F}_q \to \mathbb{F}_q$, such that for any variables x and y, $\mathsf{L}(x \oplus y) = \mathsf{L}(x) \oplus \mathsf{L}(y)$. Similarly, it can be correctly constructed by adopting linear transformation on the shares with the same index independently, and thus we denote such a gadget as *the trivial masked linear transformation*.

As $\mathsf{TrivAdd}$ and $\mathsf{TrivLin}_{\mathsf{L}}$ manipulate the input shares with different indexes separately, we can directly draw the conclusions that both of them are PINI, and any output share

Gadget 1 TrivAdd

```
Input: shares $x_{1:d+1}$ and $y_{1:d+1}$
Output: shares $z_{1:d+1}$ such that $\bigoplus_{i=1}^{d+1} z_i = \bigoplus_{i=1}^{d+1} x_i \oplus \bigoplus_{i=1}^{d+1} y_i$.
1: $z_i \leftarrow x_i \oplus y_i$, for $i \in [1 : d+1]$
```

Gadget 2 $\mathsf{TrivLin_L}$

```
Input: shares $x_{1:d+1}$
Output: shares $z_{1:d+1}$ such that $\bigoplus_{i=1}^{d+1} z_i = \mathsf{L}\big(\bigoplus_{i=1}^{d+1} x_i\big)$
1: $z_i \leftarrow \mathsf{L}(x_i)$, for $i \in [1 : d+1]$
```

with index i is determined by input shares with the same index. The latter one naturally allows the precomputation. In this paper, we call addition and linear transformation as *linear operations*, and the corresponding trivial implementation in the masked domain as *the trivial masked linear operations.*

Unlike the linear operations, the field multiplication in the masked domain cannot be trivially implemented since the encoder is usually not a homomorphism over nonlinear functions. The complexity of most multiplication gadgets in $\mathbb{F}_q$ is $O(d^2 \log^2 q)$ and requires $O(d^2)$ random variables. Besides, a PINI multiplication gadget compatible with the precomputed paradigm is challenging and will be mainly investigated in the rest of this paper. The last type is the refreshing gadget that is also known as the refreshing. This functionality is to re-randomize the input sharings. In some cases, refreshing gadget is inserted between two gadgets that cannot be composed directly.

3 New Masking Scheme

3.1 Constructions of New Multiplication Gadgets

In this subsection, we present our new multiplication gadget suitable to the precomputation-based paradigm. The main part Mul_k with $k \leq d+1$ takes a part of input shares $x_{1:k}$ and $y_{1:k}$, and returns a part of output shares $z_{1:k}$ such that $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$. The gadget Mul_k is a recursive one. That is, it first calls Mul_{k-1} to compute $u_{1:k-1}$ by $x_{1:k-1}$ and $y_{1:k-1}$, and then generates random variables $r_{1:k-1}$ which are used as a part of output shares $z_{1:k-1}$, and finally calculates the other part of output share z_k by $r_{1:k-1}$, $u_{1:k-1}$, $x_{1:k}$ and $y_{1:k}$. We also illustrate the procedure of the gadget in Figure 1. The masked multiplication Mul_{d+1} (taking all input shares $x_{1:d+1}$ and $y_{1:d+1}$, and returning all output shares $z_{1:d+1}$) can be achieved by assigning $k = d+1$.

We give the correctness and security of Mul_k in Theorem 1.

Theorem 1. *For* Mul_k *with* $k \leq d+1$*, we have:*

- ***Correctness***. $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$.
- ***Security***. Mul_k *is PINI.*

Proof (Correctness). If $k = 1$, then obviously $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$. For $k > 1$, we assume $\bigoplus_{i=1}^{k-1} z_i = \bigoplus_{i=1}^{k-1} x_i \bigoplus_{i=1}^{k-1} y_i$, and attempt to prove that $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$.

Gadget 3 Mul_k

Input: Shares $x_{1:k}$ and $y_{1:k}$.
Output: Shares $z_{1:k}$ such that $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$

```
1:  if $k = 1$ then
2:      $z_1 \leftarrow x_1 y_1$
3:      Exit
4:  end if
5:  $u_{1:k-1} \leftarrow \mathsf{Mul}_{k-1}(x_{1:k-1}, y_{1:k-1})$    ▷ We have $\bigoplus_{i=1}^{k-1} u_i = \bigoplus_{i=1}^{k-1} x_i \bigoplus_{i=1}^{k-1} y_i$
6:  Generate random variables $r_1, \ldots, r_{k-1}$
7:  for $i := 1$; $i \le k-1$; $i{+}{+}$ do
8:      $\tilde{r}_i \leftarrow u_i \ominus r_i$    ▷ $\tilde{r}_i$ is determined by $x_{1:i}$, $y_{1:i}$ and random variables
9:  end for
10: for $i := 1$; $i \le k-1$; $i{+}{+}$ do
11:     $s_i \leftarrow (x_i \oplus \tilde{r}_i) y_k \oplus (1 \ominus y_k)\tilde{r}_i$    ▷ We have $s_i = x_i y_k \oplus \tilde{r}_i$
12:     $t_i \leftarrow (y_i \oplus s_i) x_k \oplus (1 \ominus x_k) s_i$    ▷ We have $t_i = x_i y_k \oplus x_k y_i \oplus \tilde{r}_i$
13: end for
14: $z_k \leftarrow x_k y_k \oplus \bigoplus_{i=1}^{k-1} t_i$    ▷ We have $z_k = x_k y_k \oplus \bigoplus_{i=1}^{k-1}(x_i y_k \oplus x_k y_i \oplus u_i \ominus r_i)$
15: $z_{1:k-1} \leftarrow r_{1:k-1}$
```

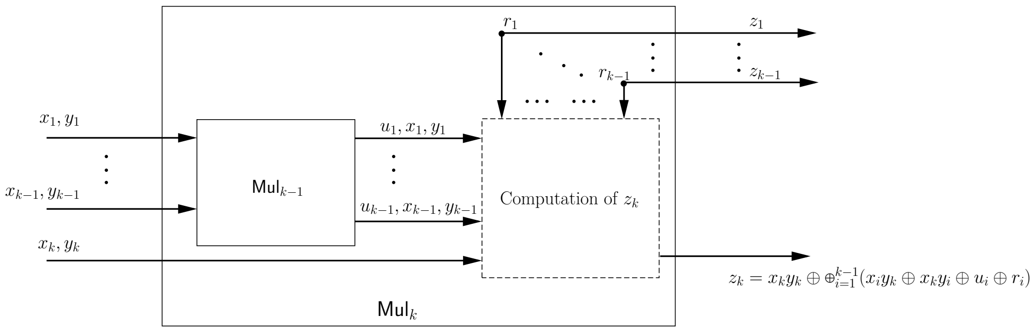


Figure 1: Structure of Mul_k: $\bigoplus_{i=1}^{k} z_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i$ and $\bigoplus_{i=1}^{k-1} u_i = \bigoplus_{i=1}^{k-1} x_i \bigoplus_{i=1}^{k-1} y_i$

By the instruction of Mul_k, we have

$$\begin{aligned}
\bigoplus_{i=1}^{k} z_i = z_k \oplus \bigoplus_{i=1}^{k-1} z_i &= x_k y_k \oplus \bigoplus_{i=1}^{k-1} t_i \oplus \bigoplus_{i=1}^{k-1} r_i \\
&= x_k y_k \oplus \bigoplus_{i=1}^{k-1} (x_k y_i \oplus y_k x_i \oplus \tilde{r}_i) \oplus \bigoplus_{i=1}^{k-1} r_i \\
&= x_k y_k \oplus \bigoplus_{i=1}^{k-1} (x_k y_i \oplus y_k x_i \oplus u_i \ominus r_i) \oplus \bigoplus_{i=1}^{k-1} r_i \qquad (1) \\
&= x_k y_k \oplus \bigoplus_{i=1}^{k-1} (x_k y_i \oplus y_k x_i) \oplus \bigoplus_{i=1}^{k-1} u_i \\
&= x_k y_k \oplus \bigoplus_{i=1}^{k-1} (x_k y_i \oplus y_k x_i) \oplus \bigoplus_{i=1}^{k-1} x_i \bigoplus_{i=1}^{k-1} y_i = \bigoplus_{i=1}^{k} x_i \bigoplus_{i=1}^{k} y_i \ .
\end{aligned}$$

□

Proof (Security). If $k = 1$, then obviously Mul_k is PINI. For $k > 1$, assuming Mul_{k-1} is PINI, we attempt to prove that Mul_k is PINI as well. All the probes $\mathcal{P}$ are partitioned into different subsets based on the types of variables:

- Internal and output probes of Mul_{k-1}: $\mathcal{P}_{\mathsf{Mul}_{k-1}}$. As Mul_{k-1} is PINI, there exists a set $\mathcal{I}_{\mathsf{Mul}_{k-1}}$ of indices with $|\mathcal{I}_{\mathsf{Mul}_{k-1}}| \leq |\mathcal{P}_{\mathsf{Mul}_{k-1}}|$ such that the probes in $\mathcal{P}_{\mathsf{Mul}_{k-1}}$ can be simulated by $x_{\mathcal{I}_{\mathsf{Mul}_{k-1}}}$ and $y_{\mathcal{I}_{\mathsf{Mul}_{k-1}}}$.
- Probes to random variables $r_{1:k-1}$: $\mathcal{P}_R$. Note that, $\mathcal{P}_R$ is also the set of the output shares $z_{1:k-1}$, and thus $\mathcal{P}_R \subseteq z_{\mathcal{O}}$. Let $\mathcal{I}_R$ be the indices corresponding to $\mathcal{P}_R$, and we have $\mathcal{I}_R = \mathcal{O}/\{k\}$ and $|\mathcal{I}_R| \leq |\mathcal{P}_R|$.
- Probes to input shares: $\mathcal{P}_{input}$. Let $\mathcal{I}_{input}$ be the indices corresponding to $\mathcal{P}_{input}$, and we have $|\mathcal{I}_{input}| \leq |\mathcal{P}_{input}|$.
- Probes to $\tilde{r}_{1:k-1}$: $\mathcal{P}_{\tilde{r}}$. Let $\mathcal{I}_{\tilde{r}}$ be the indices corresponding to $\mathcal{P}_{\tilde{r}}$, and we have $|\mathcal{I}_{\tilde{r}}| \leq |\mathcal{P}_{\tilde{r}}|$.
- Probes to the variables within the computation of s_i and t_i for $i \in [1 : k-1]$: $\mathcal{P}_{st}$, which is made up of $(x_i \oplus \tilde{r}_i)$, $(x_i \oplus \tilde{r}_i)y_k$, $(1 \ominus y_k)$, $(1 \ominus y_k)\tilde{r}_i$, $(x_i \oplus \tilde{r}_i)y_k \oplus (1 \ominus y_k)\tilde{r}_i$, $(x_i \oplus s_i)$, $(y_i \oplus s_i)x_k$, $(1 \ominus x_k)$, $(1 \ominus x_k)s_i$ and $(y_i \oplus s_i)x_k \oplus (1 \ominus y_k)s_i$ for $i \in [1 : k-1]$. Let $\mathcal{I}_{st}$ be the indices corresponding to $\mathcal{P}_{\tilde{r}}$ excluding k, and we have $|\mathcal{I}_{st}| \leq |\mathcal{P}_{st}|$.
- Probes to the variable within the computation of $z_k = x_k y_k \oplus \bigoplus_{i=1}^{k-1} t_i$: $\mathcal{P}_{\oplus}$. Note that the variable z_k (as an output probe) is excluded.
- The probe to z_k, $\mathcal{P}_z$. If $|\mathcal{P}_z| = 1$, then $k \in \mathcal{O}$, otherwise, $k \notin \mathcal{O}$.

The internal probes are $\mathcal{P}_{int} = \mathcal{P}_{\mathsf{Mul}_{k-1}} \cup \mathcal{P}_{input} \cup \mathcal{P}_{\tilde{r}} \cup \mathcal{P}_{st} \cup \mathcal{P}_{\oplus}$ and the output probes are $\mathcal{P}_O = \mathcal{P}_R \cup \mathcal{P}_z$. We then analyze the simulation of the probes in the rest of the proof.

Some intuitions. The most challenging part of the simulation is on $\mathcal{P}_{st}$, $\mathcal{P}_{\oplus}$ and $\mathcal{P}_z$, since each of other probes only relates to at most one input share. For every probe in $\mathcal{P}_{st}$, as the variable $t_i = x_i y_k \oplus x_k y_i \oplus \tilde{r}_i$ is masked by the random variable r_i, two probes in the computation of t_i (e.g., t_i and $\tilde{r}_i$) can be simulated with input shares whose indices are i and k (i.e., two indices); meanwhile, by the process of the calculation (lines 11 and 12), one probe can be simulated with input shares whose index is i or k (i.e., one index). For every probe in $\mathcal{P}_{\oplus} \cup \mathcal{P}_z$ (say, p), if no previous variable relating to i is probed (say r_i), then p can be simulated without x_i or y_i.

Let $\mathcal{I}' \overset{\text{def}}{=} \mathcal{I}_{\mathsf{Mul}_{k-1}} \cup \mathcal{I}_R \cup \mathcal{I}_{input} \cup \mathcal{I}_{\tilde{r}} \cup \mathcal{I}_{st}$ and $\mathcal{P}' \overset{\text{def}}{=} \mathcal{P} \setminus (\mathcal{P}_{\oplus} \cup \mathcal{P}_z) = \mathcal{P}_{\mathsf{Mul}_{k-1}} \cup \mathcal{P}_R \cup \mathcal{P}_{input} \cup \mathcal{P}_{\tilde{r}} \cup \mathcal{P}_{st}$. We have $|\mathcal{I}'| \leq |\mathcal{P}'|$. We separate the analysis into two cases as follows.

- **Case 1**: $\mathcal{P}_{\oplus} \cup \mathcal{P}_z = \emptyset$ and $|\mathcal{P}'| = |\mathcal{I}'|$. Case 1 conveys that $|\mathcal{I}_{\mathsf{Mul}_{k-1}}| = |\mathcal{P}_{\mathsf{Mul}_{k-1}}|$, $|\mathcal{I}_R| = |\mathcal{P}_R|$, $|\mathcal{I}_{input}| = |\mathcal{P}_{input}|$, $|\mathcal{I}_{\tilde{r}}| = |\mathcal{P}_{\tilde{r}}|$ and $|\mathcal{I}_{st}| = |\mathcal{P}_{st}|$. Intuitively, in this case, each probe can be simulated with input shares relating to only one index. We build an indices' set $\mathcal{I}$ and run a simulator that proceeds by the following steps.

  1. Initiate the set $\mathcal{I}$ to be empty.
  2. The probes in $\mathcal{P}_R$ can be simulated by sampling from uniform distributions.
  3. For the probes in $\mathcal{P}_{input}$, put the corresponding indices $\mathcal{I}_{input}$ into $\mathcal{I}$, and the probes can be simulated with $x_{\mathcal{I}}$ and $y_{\mathcal{I}}$. Now, we have $|\mathcal{I}| = |\mathcal{P}_{input}|$.
  4. For the probes in $\mathcal{P}_{\mathsf{Mul}_{k-1}}$, put $\mathcal{I}_{\mathsf{Mul}_{k-1}}$ into $\mathcal{I}$, then $\mathcal{P}_{\mathsf{Mul}_{k-1}}$ can be simulated with $x_{\mathcal{I}}$ and $y_{\mathcal{I}}$. Now, we have $|\mathcal{I}| = |\mathcal{P}_{input}| + |\mathcal{P}_{\mathsf{Mul}_{k-1}}|$.
  5. For each probe in $\mathcal{P}_{\tilde{r}}$ with index i, say $p = u_i \ominus r_i$. As $|\mathcal{P}'| = |\mathcal{I}'|$, r_i is not simulated before. Also note that, the variable is added with r_i and thus can be simulated by sampling from uniform distributions.

6. For each probe in $\mathcal{P}_{st}$, put k into $\mathcal{I}$. As $|\mathcal{P}'| = |\mathcal{I}'|$, this probe is the only one related to index i. Then, our analysis is based on the type of the probe:
   - $x_i \oplus \tilde{r}_i = x_i \oplus u_i \oplus r_i$: as r_i is not simulated before, it can be simulated by sampling from uniform distribution.
   - $(x_i \oplus \tilde{r}_i)y_k$: it can be simulated by sampling $(x_i \oplus \tilde{r}_i)$ from uniform distribution and $y_{\mathcal{I}}$
   - $(1 \ominus y_k)\tilde{r}_i$: it can be simulated by sampling $\tilde{r}_i$ from uniform distribution and $y_{\mathcal{I}}$
   - $s_i = (x_i \oplus \tilde{r}_i)y_k \oplus (1 \ominus y_k)\tilde{r}_i = x_i y_k \oplus \tilde{r}_i$: it can be simulated by sampling from uniform distribution.
   - $y_i \oplus s_i$: it can be simulated by sampling from uniform distribution.
   - $(y_i \oplus s_i)x_k$: it can be simulated by sampling $(y_i \oplus s_i)$ from uniform distribution and $x_{\mathcal{I}}$
   - $(1 \ominus x_k)s_i$: it can be simulated by sampling s_i from uniform distribution and $x_{\mathcal{I}}$
   - $t_i = (y_i \oplus s_i)x_k \oplus (1 \ominus x_k)s_i = x_i y_k \oplus y_i x_k \oplus \tilde{r}_i$: it can be simulated by sampling from uniform distribution.

Now, if $|\mathcal{P}_{st}| \geq 1$ we have $|\mathcal{I}| = |\mathcal{P}_{input}| + |\mathcal{P}_{\mathsf{Mul}_{k-1}}| + 1$, otherwise we have $|\mathcal{I}| = |\mathcal{P}_{input}| + |\mathcal{P}_{\mathsf{Mul}_{k-1}}|$, which conveys that $|\mathcal{I}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\mathsf{Mul}_{k-1}}| + |\mathcal{P}_{st}|$. Then, all the probes are simulated with input shares $x_{\mathcal{I}\cup\mathcal{O}} \cup y_{\mathcal{I}\cup\mathcal{O}}$ such that $|\mathcal{I}| \leq |\mathcal{P}_{int}|$.

- **Case 2**: $|\mathcal{P}_{\oplus} \cup \mathcal{P}_z| > 0$ or $|\mathcal{P}'| > |\mathcal{I}'|$. Case 2 conveys that $|\mathcal{P}| = |\mathcal{P}' \cup \mathcal{P}_{\oplus} \cup \mathcal{P}_z| \geq |\mathcal{I}' \cup \{k\}|$. By the instruction of Mul_k, the probes in $\mathcal{P}'$ can be simulated by $x_{\mathcal{I}'\cup\{k\}}$, $y_{\mathcal{I}'\cup\{k\}}$ and $r_{\mathcal{I}'}$, and they have no relation with $r_{[1:k-1]\setminus\mathcal{I}'}$. Then, We build a set $\mathcal{I}$ of indices and run a simulator that proceeds by the following steps.
  - Put indices in $\mathcal{I}' \setminus \mathcal{O}$ into $\mathcal{I}$. If $z_k \in \mathcal{P}_O$, then put k into $\mathcal{I}$. Now, we have $|\mathcal{I}| \leq |\mathcal{P} \setminus \mathcal{P}_O| = |\mathcal{P}_{int}|$ and $k \in \mathcal{I} \cup \mathcal{O}$.
  - The probes in $\mathcal{P}'$ can be directly simulated by $x_{\mathcal{I}\cup\mathcal{O}}$, $y_{\mathcal{I}\cup\mathcal{O}}$.
  - We then analyze the probes in $\mathcal{P}_{\oplus}$ and $\mathcal{P}_z$. Let $\bar{\mathcal{I}} \overset{\text{def}}{=} [1 : k-1] \setminus (\mathcal{I} \cup \mathcal{O})$. Each probe in $\mathcal{P}_{\oplus}$ and $\mathcal{P}_z$ can be represented as $p = g(x_k y_k, t_{\mathcal{I}\cup\mathcal{O}}, t_{\bar{\mathcal{I}}})$ with g a function. We separate the analysis as follows.
    * $x_k y_k$ can be simulated by $x_{\mathcal{I}\cup\mathcal{O}}$ and $y_{\mathcal{I}\cup\mathcal{O}}$, since $k \in \mathcal{I} \cup \mathcal{O}$.
    * $t_{\mathcal{I}\cup\mathcal{O}}$ can be simulated by $x_{\mathcal{I}\cup\mathcal{O}}$ and $y_{\mathcal{I}\cup\mathcal{O}}$, since for any $i \in \mathcal{I} \cup \mathcal{O}$, $t_i = x_i y_k \oplus x_k y_i \oplus u_i \ominus r_i$, $k \in \mathcal{I} \cup \mathcal{O}$ and u_i can be simulated with x_i and y_i.
    * For any $i \in \bar{\mathcal{I}}$, we have $t_i = x_i y_k \oplus x_k y_i \oplus u_i \ominus r_i$, which can be safely replaced by r_i, since r_i has no relation with probes in $\mathcal{P}'$. Thus, we can simulate $t_{\bar{\mathcal{I}}}$ by sampling from the uniform distributions.

  Therefore, probes in $\mathcal{P}_{\oplus}$ and $\mathcal{P}_z$ can be simulated by $x_{\mathcal{I}\cup\mathcal{O}}$ and $y_{\mathcal{I}\cup\mathcal{O}}$.

Now, all the probes are simulated with input shares $x_{\mathcal{I}\cup\mathcal{O}} \cup y_{\mathcal{I}\cup\mathcal{O}}$ such that $|\mathcal{I}| \leq |\mathcal{P}_{int}|$ with $\mathcal{P}_{int}$ internal probes, indicating that Mul_k is PINI. □

The main advantage of Mul_{d+1} is its suitability to the precomputation-based paradigm. That is, it can be divided into two parts: precomputation and online-computation. The precomputation takes the input $x_{1:d}$ and $y_{1:d}$ and produces output shares $z_{1:d}$ and some precomputed values. The precomputed values include $\tilde{r}_{1:d}$, $x_{1:d}$ and $y_{1:d}$. Note that, if the gadget is an output one (of the composed gadget), we should additionally regard the output shares with indices in $[1 : d]$ (i.e., $z_{1:d}$) as precomputed values, for the complete output shares. Then, the online-computation takes the input x_{d+1}, y_{d+1} and precomputed values to compute the output share z_{d+1}. We present the procedure in Gadget 4. It should be noted that the shares corresponding to the non-input sensitive variable should be the output shares of the preceding gadget and can be precomputed. At the same time, the shares (say $x_{1:d+1}$) corresponding to the input secret should be firstly refreshed (using Gadget 5 given in Section 3.3) to $x'_{1:d+1}$ such that $x'_{1:d}$ is determined by the random variable in the refreshing, making $x'_{1:d}$ able to be precomputed. More details can be found in Section 3.3.

Gadget 4 Mul_{d+1} with precomputation

```
Input: $x_{1:d+1}$, $y_{1:d+1}$
Output: $z_{1:d+1}$ such that $\bigoplus_{i=1}^{d+1} z_i = \bigoplus_{i=1}^{d+1} x_i \bigoplus_{i=1}^{d+1} y_i$

Precomputation
Input: $x_{1:d}$, $y_{1:d}$
Output: $\tilde{r}_{1:d}$ and $z_{1:d}$
1: $u_{1:d} \leftarrow \mathsf{Mul}_d(x_{1:d}, y_{1:d})$
2: Generate random variables $r_1, \ldots, r_d$
3: $z_{1:d} \leftarrow r_{1:d}$
4: for $i := 1$; $i \leq d$; $i{+}{+}$ do
5:     $\tilde{r}_i \leftarrow u_i \ominus r_i$
6: end for
   Precomputed values for non-output gadget: $\tilde{r}_{1:d}$, $x_{1:d}$ and $y_{1:d}$.
   Precomputed values for output gadget: $z_{1:d}$, $\tilde{r}_{1:d}$, $x_{1:d}$ and $y_{1:d}$ .

Online computation
Input: $\tilde{r}_{1:d}$, $x_{1:d+1}$ and $y_{1:d+1}$
Output: $z_{d+1}$
1: for $i := 1$; $i \leq d$; $i{+}{+}$ do
2:     $s_i \leftarrow (x_i \oplus \tilde{r}_i) y_{d+1} \oplus (1 \ominus y_{d+1}) \tilde{r}_i$
3:     $t_i \leftarrow (y_i \oplus s_i) x_{d+1} \oplus (1 \ominus x_{d+1}) s_i$        ▷ We have $t_i = x_{d+1} y_i \oplus y_{d+1} x_i \oplus \tilde{r}_i$
4: end for
5: $z_{d+1} \leftarrow x_{d+1} y_{d+1} \oplus \bigoplus_{i=1}^{d} t_i$
```

3.2 Formal Verification of the New Multiplication Gadget

In general, checking that a gadget is secure in the probing model or the relevant composable security model is an error-prone process. This motivates the need for formal verification of higher-order masking, allowing the designers and engineers to analyze and verify the designs. To approve the security proof in Section 3.1, we provide a formal verification of the new multiplication gadget. The basic idea of the formal verification for masking is to exhaustively enumerate all the tuples of probes attempting to find one that does not satisfy the security notions (e.g., PINI). The space of enumeration rapidly increases with the number of variables. Thus, a formal verification tool can only verify masking with constraint security orders.

The recursive structure of our masked multiplication significantly benefits the fast verification thereof, enabling larger security orders that can be verified. We assume that Mul_{k-1} is PINI, and attempt to verify Mul_k is PINI. This strategy features the fact that the variables in Mul_{k-1} are not needed to be considered. Also note that, the intermediate variables that are in Mul_k but out of Mul_{k-1} are only a small subset of all the intermediate variables in Mul_k. That is, the multiplication gadget Mul_{d+1} is made up of Mul_d that contains $O(d^2)$ variables and the other part that contains $O(d)$ variables. We only need to check the $O(d)$ variables out of Mul_d.

Our verification method follows the language-based approach in [BBD+15, BBC+19], and so provides further evidence of the benefits of language-based approaches. As described in [BBD+15, BBC+19], the method follows a divide-and-conquer approach, embodied in two algorithms. The first algorithm checks if leakages are independent of secrets for a fixed admissible set of observations. The algorithm repeatedly applies semantic-preserving simplifications to the symbolic representation of the leakages, until it does not depend on secrets or fails. The second algorithm explores all admissible observation sets, calling the first algorithm on each of them. We separate the verification into two cases as the security proof, and verify them separately. This further accelerates the speed of verification.

Table 1 summarizes the verification outcomes of the new multiplication gadget. Our verification tool is a single-threaded one written in Python. We use a 2.5 GHz Intel Core i7-11700 with 16 GB of RAM running in Windows 10. The verification can be completed until the security order $d = 16$. While, to the best of our knowledge, known formal verification tools for masking can at best verify the multiplication gadget up to the order $d = 10$ after a significant parallel computation effort [BK21].

Table 1: Verification outcomes of Mul_{d+1} using our optimized tool.

	$d=1$	$d=2$	...	$d=14$	$d=15$	$d=16$
Verification time	<1 second		...	≈ 1 hour	<5 hours	<16 hours
Is PINI?	✓	✓	✓	✓	✓	✓

The basic probing model does not consider specific physical defaults, such as glitches, transitions, or couplings, that may occur during the processing of sensitive information on a physical device. Thus, extensions of the probing model considering specific physical defaults and formal verification methods on those more established theories have gained relevance over the last years [BGI+18, FGP+18, BBC+19, KSM20]. However, we do not investigate them in this paper, and pose the security in the probing model as the first necessary step to a provably side-channel secure implementation. Besides, we provide a T-test evaluation for our implementations in Section 4.5, which validates the security order in practice.

We emphasize that, the fast speed of verification is only because of the structure of our multiplication gadget which allows the verification of a small subset of variables and the separation of two cases. It is not related to the verification method. We developed our own tool since it can be customized for the multiplication gadget. Last but not least, to re-confirm the security, we provide verification of our multiplication (over the field $\mathbb{F}_2$) using the existing and popular tool SILVER [KSM20]. The verification can be completed until the security order $d = 5$, which is shown in Table 2. It is, to the best of our knowledge, already the largest security order in PINI to be achieved for a multiplication gadget using SILVER. However, it is still slower than the verification using our tool, due to the following two reasons. Firstly, without deeply customizing the SILVER, some optimizations (e.g., separating the verification process into two cases separately) that can accelerate the verification cannot be adopted. Secondly, SILVER is much more powerful than ours (supporting verification for the security in the extended probing model of both software and hardware designs), making it less efficient for larger designs. For instance, SILVER returns the results for both PINI and extended PINI, and the latter considers much more internal tuples to be checked, significantly elongating its running time. In contrast, our tool only focuses on security in PINI and does not check additional internal tuples related to the extended PINI.

Table 2: Verification outcomes of Mul_{d+1} using SILVER.

	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$
Verification time (in seconds)	0.005	0.037	3.29	597.5	170242
Is PINI?	✓	✓	✓	✓	✓

3.3 Precomputation-based Design Using Mul_{d+1}, TrivAdd and TrivLin

The precomputation-based design paradigm relies on the fact that the shares with indices in $[1:d]$ can be precomputed without knowing the shares with index $d+1$. Hence, we provide a refreshing algorithm in Gadget 5 whose output shares with indices $[1:d]$ are determined by the randomness. This algorithm can produce the shares (corresponding to the secret input variables) for the precomputation.

```
Gadget 5 Refresh
Input: $x_{1:d+1}$.
Output: $z_{1:d+1}$ such that $\bigoplus_{i=1}^{d+1} z_i = \bigoplus_{i=1}^{d+1} x_i$

Precomputation
Input: Empty
Output: $z_{1:d}$
1: Generate random elements $r_{1:d}$
2: $z_{1:d} \leftarrow r_{1:d}$
   Precomputed values: $r_{1:d}$.

Online-computation
Input: $x_{1:d+1}$ and $r_{1:d}$.
Output: $z_{d+1}$
1: $z_{d+1} \leftarrow x_{d+1} \oplus \bigoplus_{i=1}^{d}(x_i \ominus r_i)$
```

We give the correctness and security of Refresh in Theorem 2.

Theorem 2. *For* Refresh*, we have:*

- ***Correctness.*** $\bigoplus_{i=1}^{d+1} z_i = \bigoplus_{i=1}^{d+1} x_i$.
- ***Security.*** Refresh *is PINI.*

Proof (Correctness). By the instruction of Refresh, we have

$$\bigoplus_{i=1}^{d+1} z_i = z_{d+1} \oplus \bigoplus_{i=1}^{d} z_i = x_{d+1} \oplus \bigoplus_{i=1}^{d}(x_i \ominus r_i) \oplus \bigoplus_{i=1}^{d} r_i = x_{d+1} \oplus \bigoplus_{i=1}^{d} x_i = \bigoplus_{i=1}^{d+1} x_i$$

□

Proof (Security). We partition the probes into different subsets:

- Probes to input shares: $\mathcal{P}_{input}$.
- Probes to random elements $r_{1:d}$: $\mathcal{P}_R$. They are the output with indices in $[1:d]$.
- Probes to the variable within the computation of $z_{d+1} = x_{d+1} \oplus \bigoplus_{i=1}^{d}(x_i \ominus r_i)$: $\mathcal{P}_{\oplus}$. Note that the variable z_{d+1} (as an output probe) is excluded.
- The probe to z_{d+1}, $\mathcal{P}_z$. If $|\mathcal{P}_z| = 1$, then $d+1 \in \mathcal{O}$; otherwise, $d+1 \notin \mathcal{O}$. That is, $\mathcal{P}_z \cup \mathcal{P}_R = z_{\mathcal{O}}$.

In the rest of the proof, the most challenging part should be the probes in $\mathcal{P}_{\oplus}$. Intuitively, if r_i or x_i is probed, then we might use x_i to simulate probes in $\mathcal{P}_{\oplus}$; otherwise, the probes can be simulated without x_i. We build an indices' set $\mathcal{I}$ and run a simulator that proceeds by the following steps.

1. Initiate the set $\mathcal{I}$ to be empty.

2. For the probes in $\mathcal{P}_R$, they can be simulated by sampling from uniform distributions.

3. For the probes in $\mathcal{P}_{input}$, put the corresponding indices into $\mathcal{I}$, and the probes can be simulated with $x_{\mathcal{I}}$. Now, we have $|\mathcal{I}| \leq |\mathcal{P}_{input}|$.

4. We then consider probes in $\mathcal{P}_{\oplus}$ and $\mathcal{P}_z$. Let $\mathcal{I}' \overset{\text{def}}{=} \mathcal{I} \cup \mathcal{O} \setminus \{d+1\}$, and $\bar{\mathcal{I}}' \overset{\text{def}}{=} [1:d] \setminus \mathcal{I}'$. Each probe can be represented as $p = g\big(x_{d+1}, (x_{\mathcal{I}'} \oplus r_{\mathcal{I}'}), (x_{\bar{\mathcal{I}}'} \ominus r_{\bar{\mathcal{I}}'})\big)$ with g a function. If $|\mathcal{P}_{\oplus}| \geq 1$, we put $d+1$ into $\mathcal{I}$, we separate our analysis into two cases:

   - x_{d+1} can be simulated with $x_{\mathcal{I} \cup \mathcal{O}}$, since $d+1 \in \mathcal{I} \cup \mathcal{O}$.
   - $x_{\mathcal{I}'} \oplus r_{\mathcal{I}'}$ can be simulated with $x_{\mathcal{I} \cup \mathcal{O}}$, since $\mathcal{I}' \subseteq \mathcal{I} \cup \mathcal{O}$.
   - $r_{\bar{\mathcal{I}}'}$ is not simulated before and thus $x_{\bar{\mathcal{I}}'} \ominus r_{\bar{\mathcal{I}}'}$ can be simulated by sampling from uniform distributions.

   Now, we have $|\mathcal{I}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\oplus}|$.

Now, all the probes are simulated with shares $x_{\mathcal{I} \cup \mathcal{O}}$ such that $|\mathcal{I}| \leq |\mathcal{P}_{int}|$ with $\mathcal{P}_{int}$ internal probes, indicating that Refresh is PINI. □

We consider a circuit C with ℓ_{in} input variables and ℓ_{out} output variables, and C is made up of ℓ_{mul} multiplications, ℓ_{add} operations (including addition and transformation). Additionally, let the number of input variables for multiplications in C be ℓ', and we have $\ell' \leq 2\ell_{\text{mul}}$. We compile the circuit into a private circuit by encoding each input variable into $d+1$ shares and transforming every operation into Mul_{d+1}, TrivAdd, or $\mathsf{TrivLin}_{\mathsf{L}}$. Additionally, we refresh the $\ell_{\text{in}} \times (d+1)$ input shares using ℓ_{in} Refresh gadgets. In the following, we discuss the precomputation and online-computation separately.

3.3.1 Precomputation

This phase calculates the precomputed values. For each gadget (say, G) excluding the Refresh, we consider the case that the input shares with indices in $[1:d]$ can be precomputed from the preceding gadgets. The precomputed values of G (including its output shares with indices in $[1:d]$) are calculated from the input shares with indices in $[1:d]$ and all the random bits. The precomputed values will be used to compute the output shares whose index is $d+1$. Besides, for practicality, it is necessary to reduce the number of precomputed values as much as possible to save the RAM overhead. We describe the precomputation for the composition of multiple Mul_{d+1}, TrivAdd and $\mathsf{TrivLin}_{\mathsf{L}}$ as follows.

1. For the shares of each input variable, it runs the precomputation phase of a Refresh and produces the shares with indices $[1:d]$. Note that, as the output shares of precomputation of Refresh are only determined by the random variables, currently we do not need the shares of input secret.

2. It evaluates each gadget using input shares with indices in $[1:d]$, and produce the output shares with indices in $[1:d]$ as well. This can be done since for each gadget in Mul_{d+1}, TrivAdd or $\mathsf{TrivLin}_{\mathsf{L}}$, the output shares with indices in $[1:d]$ can be determined by input shares with indices in $[1:d]$.

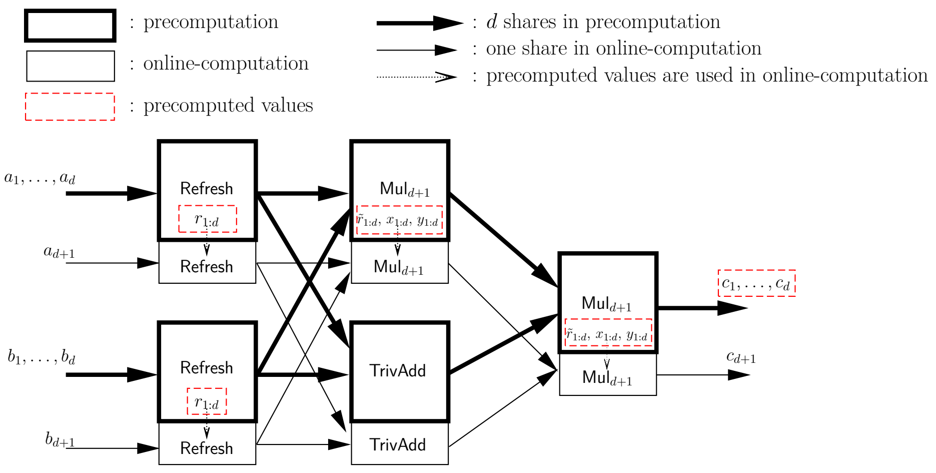


Figure 2: An example for $ab(a \oplus b)$ using Mul$_{d+1}$, TrivAdd and Refresh.

- For precomputation of each gadget Mul$_{d+1}$, it returns the values $\tilde{r}_{1:d}$, $x_{1:d}$, $y_{1:d}$ as the precomputed values. Note that, $x_{1:d}$, $y_{1:d}$ can be used as the input of other gadgets Mul$_{d+1}$.
- It return the output shares (of the composed gadget) with indices in $[1 : d]$ as the precomputed values.

The above process requires $d\ell_{\text{in}} + \ell_{\text{mul}}d(d+1)/2$ random variables in $\mathbb{F}_q$, and generates $d(\ell_{\text{in}} + \ell_{\text{out}}) + d\ell_{\text{mul}} + d\ell' \leq d(\ell_{\text{in}} + \ell_{\text{out}}) + 3d\ell_{\text{mul}}$ variables in $\mathbb{F}_q$ to be stored in the RAM.

3.3.2 Online-computation

The online-computation takes input shares with index $d + 1$ and the precomputed values, and returns the output shares with index $d + 1$. The online-computation is much more efficient than the precomputation.

1. For the shares of each input secret, it runs the online-computation of the Refresh and produces the shares with index $d + 1$.
2. It evaluates the online-computation of each gadget using input shares with index $d + 1$. If this gadget is Mul$_{d+1}$, the evaluation also requires the precomputed values $\tilde{r}_{1:d}$, $\tilde{x}_{1:d}$ and $\tilde{y}_{1:d}$. Then, it returns the output shares with index $d + 1$ as well.

In summary, the precomputation produces $d(\ell_{\text{in}} + \ell_{\text{out}}) + d\ell_{\text{mul}} + d\ell'$ precomputed values with complexity $O(d^2\ell_{\text{mul}} \log^2 q + d\ell_{\text{add}} \log q)$, and the online-computation runs in $O(d\ell_{\text{mul}} \log^2 q + d\ell_{\text{add}} \log q)$. In Figure 2, we give an example of masked $ab(a \oplus b)$, which is a composition of two instances of Mul$_{d+1}$ and one TrivAdd, and the input shares of a and b are refreshed at the very beginning.

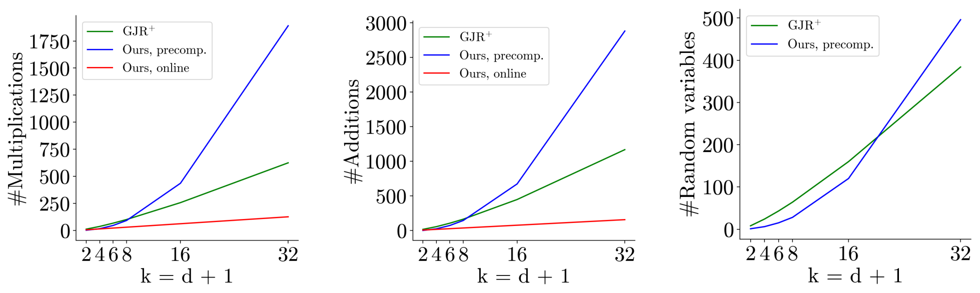
3.4 Operation Counts and Comparison with the GJR$^+$ Scheme

In this subsection, we compare our scheme with the quasilinear-complexity masking scheme that was first proposed in [GJR18] and improved in [GPRV21] by Goudarzi et al. Our comparison focuses on the multiplication gadget over fields with characteristic 2. Table 3 shows the operation counts of multiplications and additions/subtractions and required random variables, in the function of number of shares $k = d + 1$. We also provide in Figure 3 the trend of operation counts for different number of shares. We can see that, for

$k > 4$, the GJR$^+$ enjoys a smaller number of multiplications and additions/subtractions than the precomputation of our scheme. While, our online computation is faster for any number of shares.

Table 3: Operation counts for GJR$^+$ and our scheme, where $k = d + 1$.

	Multiplication	Addition/Subtraction	Random
GJR$^+$ [GPRV21]	$7k\log(k)/2 + 2k$	$k\log^2(k)/2 + 4k\log(2k)$	$2k\log(k) + 2k$
Ours, precomp.	$2k^2 - 5k + 3$	$3(k-1)^2 - 2$	$k(k-1)/2$
Ours, online	$4k - 3$	$5(k-1) + 2$	0



(a) Multiplication counts (b) Addition/Subtraction counts (c) Random variables counts

Figure 3: Operations counts for $k = \{2, 4, 8, 16, 32\}$.

GJR$^+$ and our scheme are also quite different in other aspects. In terms of the security, GJR$^+$ provided a stronger guarantee tolerating a leakage rate of $O(1/\log k)$, which can be obtained since its complexity is quasilinear and an input-output separation refresh gadget is used for the region probing security. At the same time, when working on the field of $\mathbb{F}_{2^m}$, GJR$^+$ requires that $m \geq 2$, making the bitsliced implementation quite challenging.

4 Applications to AES and SKINNY

4.1 Descriptions of Ciphers and Their Bitslicing Approaches

The bitslice strategy allows for computing several instances of a cryptographic primitive in parallel, or alternatively, all the s-boxes in parallel within an instance of the primitive. Whilst our masking scheme is suitable to any bitsliced implementations of ciphers, in this section, we only consider the former case for a single instance of a block cipher.

4.1.1 AES-128

The AES-128 block cipher is performed on 16 bytes called state. The round function is made up of four types of transformations: AddRoundKey, SubBytes, ShiftRows and MixColumns. In AddRoundKey, the state is added with the subkey (that is derived from the key) using bitwise XOR. ShiftRows and MixColumns can be regarded as linear operations over $\mathbb{F}_{2^8}$, and thus they can be implemented by a number of XOR operations. In the SubBytes transformation, a nonlinear function $\mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ called S-box is computed over each of the 16 bytes of the state. More details can be found in, e.g., [DR02].

We consider 16 binary operations (i.e., with $q = 2$) in parallel, take the same approach of bitsliced implementation as in [GR17]. That is, we use the bitslice at the S-box level that packs the i^{th} bits of 16 S-boxes' inputs, and process 16 S-boxes in parallel. It conveys that we just use the 16 bits of one register. We use the compact representation proposed by Boyar et al. in [BMP13] for the implementation of AES S-box. Their circuit is obtained by applying logic minimization techniques to the tower-field representation of Canright [CB08]. It involves 115 logic gates, including 32 logical AND. The MixColumns and ShiftRows can be evaluated using the strategy given in [GR17], which takes 43 and 144 one-cycle instructions respectively.

4.1.2 SKINNY

SKINNY is a family of lightweight tweakable block ciphers [BJK+16]. In this paper, we consider the variants with tweakey sizes 64 and 128 respectively, and consider the case that tweakey size equals the block size. That is, we consider SKINNY-64-64 and SKINNY-128-128. And, in the rest of this paper, we abbreviate SKINNY-64-64 and SKINNY-128-128 as SKINNY-64 and SKINNY-128 respectively. The internal state of SKINNY-128 (resp., SKINNY-64) can be viewed as a 4×4 matrix of bytes (resp., nibbles). The ciphering process simply consists of several applications of the round function. The round function is composed of five layers: SubCells, AddConstants, AddRoundTweakey, ShiftRows and MixColumns. SubCells uses an 8-bit (resp., 4-bit) S-box and other layers consist of linear operations.

We consider 16 binary operations (i.e., with $q = 2$) in parallel, and take the same approach as in AES which uses the bitslice at the S-box level. As instructed in [BJK+16], the 8-bit S-box consists of 8 XOR and 8 NOR, and the 4-bit S-box consists of 4 XOR and 4 NOR. As the linear layer of SKINNY is quite similar to that of AES, we can apply the strategy of AES. It results in a bitsliced implementation with ≈ 60 operations. Unlike AES, the tweakey schedule in SKINNY is linear, and thus can be implemented efficiently. As we consider the case that tweakey size equals the block size, the tweakey schedule is a permutation P_T applied on the cells (i.e., bytes/nibbles) positions of all tweakey arrays: $P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$. We use the bitslice at the cell level. Hence, the lower 8 output bytes/nibbles can be achieved by right shifting, and the higher 8 output bytes/nibbles can be achieved by left shifting and look-up tables.

Remarks. The bitsliced implementations shown above can be significantly improved by, e.g., using the more advanced fixslicing method [AP21]. But, to the best of our knowledge, state-of-the-art higher-order masked implementations of AES are based on the above strategies, and thus we choose them as well for fair comparisons. We aim to show that the improvements (in online-computation) come from our new design of masked multiplication rather than from the better implementation of linear layers.

4.2 Masked Implementations

After bitslicing the ciphers, we adopt the strategy presented in Section 3.3 to obtain the masked implementations for AES and SKINNY. That is, we refresh the shared subkeys and transform bitwise multiplication into Mul_{d+1}, bitwise XOR into $\mathsf{TrivAdd}$ and shifts into $\mathsf{TrivLin}_{\mathsf{L}}$. The precomputation takes the random bits and produces the precomputed values, and the online-computation takes the precomputed values and the input shares to calculate the results.

The AES-128 contains $\ell_{\text{mul}} = 32 \times 10$ bitwise multiplications with $\ell' = 33 \times 10$ different 16-bit inputs. We consider the round keys as the secret input, and thus we have $\ell_{\text{in}} = 11 \times 128 = 88 \times 16$. As the block size is 128 bits, then $\ell_{\text{out}} = 128 = 8 \times 16$. Hence, the number of precomputed 16-bit variables is $d(\ell_{\text{in}} + \ell_{\text{out}}) + d\ell_{\text{mul}} + d\ell' = d(88 + 8) + d \times 10 \times (33 + 32) = 746d$. It conveys that our implementation requires $746d \times 2/1024 = 1.457d$ Kbytes of RAM to store the precomputed intermediates. And, the precomputation requires $d\ell_{\text{in}} + \ell_{\text{mul}}d(d+1)/2 = 88d + 160d(d+1) = 160d^2 + 248d$ 16-bit random variables, which is $0.3125d^2 + 0.4844d$ Kbytes.

The SKINNY-128 contains $\ell_{\mathrm{mul}} = 8 \times 40$ bitwise multiplications with $\ell' = 11 \times 40$ different 16-bit inputs. We consider the tweakey input as secret input, and thus have $\ell_{\mathrm{in}} = 128 = 8 \times 16$. As the block size is 128 bits, then $\ell_{\mathrm{out}} = 128 = 8 \times 16$. Hence, the number of precomputed 16-bit variables is $d(\ell_{\mathrm{in}}+\ell_{\mathrm{out}})+d\ell_{\mathrm{mul}}+d\ell' = d(8+8)+d\times 40\times(11+8) = 776d$, requiring $776d \times 2/1024 = 1.5156d$ Kbytes of RAM. And, the precomputation requires $d\ell_{\mathrm{in}} + \ell_{\mathrm{mul}}d(d+1)/2 = 8d + 160d(d+1) = 160d^2 + 168d$ 16-bit random variables, which is $0.3125d^2 + 0.3281d$ Kbytes.

The SKINNY-64 contains $\ell_{\mathrm{mul}} = 4 \times 32 = 128$ bitwise multiplications with $\ell' = 5 \times 32 = 160$ different 16-bit inputs. We consider the tweakey input as secret input, and thus have $\ell_{\mathrm{in}} = 64 = 4 \times 16$. As the block size is 64 bits, then $\ell_{\mathrm{out}} = 64 = 4 \times 16$. Hence, the number of precomputed 16-bit variables is $d(\ell_{\mathrm{in}}+\ell_{\mathrm{out}})+d\ell_{\mathrm{mul}}+d\ell' = d(4+4)+d\times(160+128) = 296d$, requiring $296d \times 2/1024 = 0.5781d$ Kbytes of RAM. And, the precomputation requires $d\ell_{\mathrm{in}} + \ell_{\mathrm{mul}}d(d+1)/2 = 4d + 64d(d+1) = 64d^2 + 68d$ 16-bit random variables, which is $0.125d^2 + 0.1328d$ Kbytes.

4.3 Some Details to Prevent Transitional Leakage

A precaution of the masked implementation in practice is that, the implementation should be done carefully to avoid known implementation issues [BDF+17, DFS15, CPR07] that can make it do not align with the assumptions (typically, the independent leakage assumption) for masking proofs. To prevent the issue caused by the nonlinear leakage of bits stored within a register reported in [GMPO20], our implementation always load at most one share to a register.

In Algorithm 1, we describe the code snippet of the online-computation of Mul_{d+1} in ARM Cortex architecture. The input values of the code snippet are in 8 registers and the result will be in register $R_{z_{d+1}}$. The address of vectors $\tilde{r}_{1:d}$, $x_{1:d}$ and $y_{1:d}$ are in registers $R_{\&\tilde{r}}$, $R_{\&x}$ and $R_{\&y}$ respectively. The values of x_{d+1} are y_{d+1} in registers $R_{x_{d+1}}$ and $R_{y_{d+1}}$. The registers R_{tmp_1} and R_{tmp_2} are used for some temporary values. In the code snippet, to prevent the transitional leakage, we always clear the registers R_{tmp_1} and R_{tmp_2} by assigning zero value whenever necessary. Besides, this code snippet is also the main part used in the precomputation.

4.4 Implementation Results

We showcase the advantage of our scheme in the precomputation-based design paradigm. We consider implementations on the ARM Cortex M architecture. For the comparison with the state-of-the-art implementations of AES, we first consider the results reported in [WGY+22] as the benchmark. The work in [WGY+22] implements inner product masking that cannot be directly applied to the bitsliced implementation, and should be obviously slower than Boolean masking in software without the support of dedicated instructions. Thus, for a more fair comparison, we implement the scheme using Boolean masking and bit-slicing technology. Concretely, we set all the parameters $\boldsymbol{\alpha}$ to be ones, and thus the gadgets can be implemented only with XOR and AND operations. This inevitably deprives the scheme of the cost amortization for $d > 1$. We implement bitwise XOR and AND using the trivial masked addition and the Gadget 5 in [WGY+22], respectively, which further optimizes the original scheme for the Boolean case. As shown in Section 5.3 of [WGY+22], the case with parameters $\boldsymbol{\alpha}$=1s may suffer from transitional leakage. Thus, we carefully insert some instructions to clear registers to prevent transitional leakage, similarly to the strategy given in Algorithm 1.

Another benchmark we consider is the bit-sliced implementation given in [GR17]. To comply with the precomputation-based paradigm and accelerate the online-computation, the masking approaches in [GR17] can at best generate all the random bits and store them in RAM, resulting in a large RAM requirement. Then, the online-computation carries out

Algorithm 1 Online-computation of Mul_{d+1} in ARM Cortex M architecture

```
Input: Registers $R_{\&\tilde{r}}$, $R_{\&x}$, $R_{\&y}$, $R_{x_{d+1}}$, $R_{y_{d+1}}$, $R_{tmp_1}$, $R_{tmp_2}$
Output: Register $R_{z_{d+1}}$
 1: MOV $R_{z_{d+1}}$, #0                    ▷ Prevent transitional leakage
 2: AND $R_{z_{d+1}}$, $x_{d+1}$, $y_{d+1}$          ▷ $R_{z_{d+1}} \leftarrow x_{d+1}y_{d+1}$
 3: for $i := 1$; $i \leq d$; $i$++ do       ▷ Loop using assembler directive
 4:     MOV $R_{tmp_1}$, #0                  ▷ Prevent transitional leakage
 5:     MOV $R_{tmp_2}$, #0                  ▷ Prevent transitional leakage
 6:     LDRH $R_{tmp_1}$, [$R_{\&x}$, #$i$]
 7:     LDRH $R_{tmp_2}$, [$R_{\&\tilde{r}}$, #$i$]
 8:     EOR $R_{tmp_1}$, $R_{tmp_2}$                ▷ $R_{tmp_1} \leftarrow x_i \oplus \tilde{r}_i$
 9:     AND $R_{tmp_1}$, $R_{y_{d+1}}$              ▷ $R_{tmp_1} \leftarrow (x_i \oplus \tilde{r}_i)y_{d+1}$
10:     NVM $R_{y_{d+1}}$, $R_{y_{d+1}}$            ▷ $R_{y_{d+1}} \leftarrow (1 \ominus y_{d+1})$
11:     AND $R_{tmp_2}$, $R_{y_{d+1}}$              ▷ $R_{tmp_2} \leftarrow (1 \ominus y_{d+1})\tilde{r}_i$
12:     EOR $R_{tmp_2}$, $R_{tmp_1}$                ▷ $R_{tmp_2} \leftarrow s_i = (x_i \oplus \tilde{r}_i)y_{d+1} \oplus (1 \ominus y_{d+1})\tilde{r}_i$
13:     MVN $R_{y_{d+1}}$, $R_{y_{d+1}}$            ▷ $R_{y_{d+1}} \leftarrow y_{d+1}$
14:     MOV $R_{tmp_1}$, #0                  ▷ Prevent transitional leakage
15:     LDRH $R_{tmp_1}$, [$R_{\&y}$, #$i$]
16:     EOR $R_{tmp_1}$, $R_{tmp_2}$                ▷ $R_{tmp_1} \leftarrow y_i \oplus s_i$
17:     AND $R_{tmp_1}$, $R_{x_{d+1}}$              ▷ $R_{tmp_1} \leftarrow (y_i \oplus s_i)x_{d+1}$
18:     NVM $R_{x_{d+1}}$, $R_{x_{d+1}}$            ▷ $R_{x_{d+1}} \leftarrow (1 \ominus y_{d+1})$
19:     AND $R_{tmp_2}$, $R_{x_{d+1}}$              ▷ $R_{tmp_2} \leftarrow (1 \ominus x_{d+1})s_i$
20:     EOR $R_{tmp_2}$, $R_{tmp_1}$                ▷ $R_{tmp_2} \leftarrow t_i = (y_i \oplus s_i)x_{d+1} \oplus (1 \ominus x_{d+1})s_i$
21:     MVN $R_{x_{d+1}}$, $R_{x_{d+1}}$            ▷ $R_{x_{d+1}} \leftarrow y_{d+1}$
22:     EOR $R_{z_{d+1}}$, $R_{tmp_2}$
23: end for
```

the masked operations with the pre-generated random bits. Indeed, an implementation can generate the random bits on-the-fly, but it will largely decelerate online computation.

The last very important line of work that should be considered for comparison is table-based masking. Our comparison considers the schemes with higher-order security, which narrows down schemes to the ones given in [Cor14, CRZ18, VV21]. Among them, we consider the one proposed by Valiveti et al. [VV21]. It is because that, in this scheme, the amount of RAM requirement is reduced to be feasible on resource-constrained devices.

We cannot give a fair benchmark for `SKINNY`, since, to our knowledge, our implementation should be the first higher-order masked implementation of `SKINNY` block cipher in software. A possible exception might be the masked implementations in [BDM+20] for `SKINNY` AEAD, which considers the running of multiple parallel instances. Nevertheless, it is different from our case that computes all the S-boxes in parallel within an instance. Moreover, as the scheme in [WGY+22] requires a large MDS matrix over the field that the cipher relies on, the scheme in [WGY+22] is difficult to be employed for `SKINNY`.

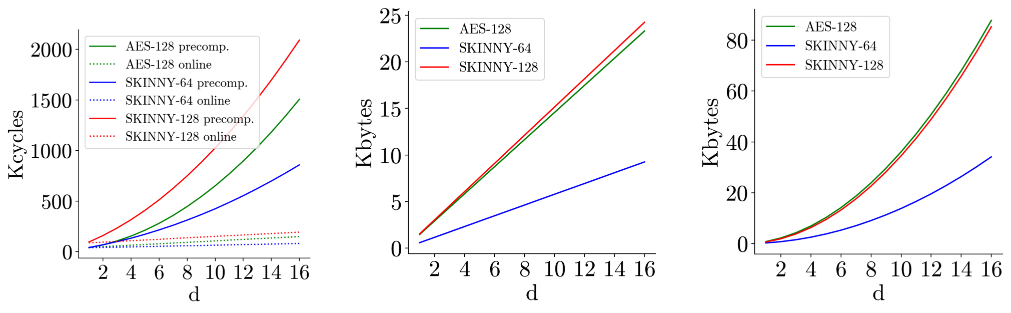
The performance results for $d = 2, 8$ are summarized in Table 4, where timings are given in kilos of clock cycles (Kcycles). To show the performance trend for $d = [1 : 16]$, we depict in Figure 4(a) the Kcycles of precomputed and online phases, in Figure 4(b) the RAM size for precomputed values, and in Figure 4(c) the random bits requirements. Compared with [WGY+22], our implementation of AES gains a significant speed-up in both precomputation and online-computation, and the requirement of RAM size for precomputed values is smaller when $d < 8$, making the scheme more practical in many embedded platforms. For instance, compared with the original scheme in [WGY+22], it saves clock cycles of the precomputation and online-computation by 8 and 1.5 times when $d = 8$. Although the scheme in [WGY+22] with the Boolean case gains a speed-up in both precomputation and online-computation beyond its version over finite field $\mathbb{F}_{2^8}$, it is still inferior to our scheme, and requires much more random bits.

Besides, we can see that our implementation of AES is much faster than the table-based masking in [VV21]. We attribute the speed-up to three reasons. First, the scheme in [VV21] applies the technique of masking with PRGs, making the complexity of its online phase to be quadratic in the security order, while ours is linear. Secondly, the scheme in [VV21] includes many field multiplications over $\mathbb{F}_{2^{16}}$, which is costly in the software implementation. Finally, we found that the work of [VV21] considers implementations in the C language, while ours are in the assembly language.

The SKINNY-64 is much faster in both precomputation and online-computation and uses less RAM space for precomputed values than ciphers (AES and SKINNY) with a block size of 128. It is thanks to its lower number of AND operations. Notably, for $d = 8$, the RAM for precomputed values for SKINNY-64 is smaller than that of AES by a factor of 2.5.

Table 4: Summary of masked implementations.

		d	Kcycles for precomp.	Random bits	RAM for precomp.	Kcycles for online.
AES	[WGY+22]	2	705	96 B	5.63 KB	60
	[WGY+22] Boolean	2	130	7.86 KB	5.12 KB	72
	[GR17]	2	–	3.75 KB	3.75 KB	83.9
	[VV21]	2	72 590	**0.011 KB**	40.1 KB	423
	Our Work	2	**67.98**	2.22 KB	**2.91 KB**	**50.03**
	[WGY+22]	8	3 662	1.5 KB	**11 KB**	137
	[WGY+22] Boolean	8	1 038	122.88 KB	16.6 KB	113
	[GR17]	8	–	45 KB	45 KB	404.5
	[VV21]	8	3 265 303	**0.56 KB**	40.8 KB	2 873
	Our Work	8	**446.34**	23.88 KB	11.66 KB	**92.27**
SKINNY -128	Our Work	2	159.28	1.91 KB	3.03 KB	75.48
	Our Work	8	749.2	22.62 KB	12.12 KB	117.72
SKINNY -64	Our Work	2	68.61	0.77 KB	1.16 KB	33.79
	Our Work	8	312.06	9.06 KB	4.62 KB	50.69



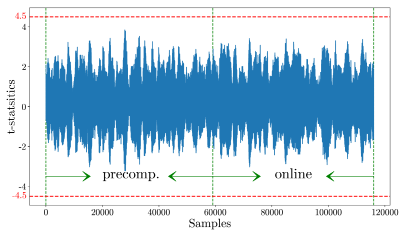
(a) Timings (b) Precomputed values (c) Random bits

Figure 4: Performance trend for $d = [1 : 16]$.

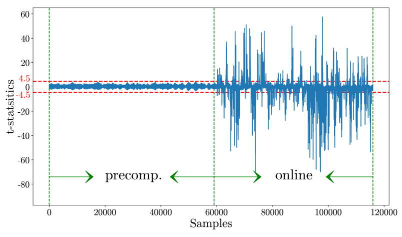
4.5 Practical Evaluations

Our implementation is ran on a ChipWhisperer STM32F303 UFO target board. The power traces of the AES round function were collected using Picoscope 5244D at sampling rate of 128 MS/s. To validate the security order in practice, we perform a fixed vs. random Welch's T-test with 50 000 fixed and random inputs respectively.

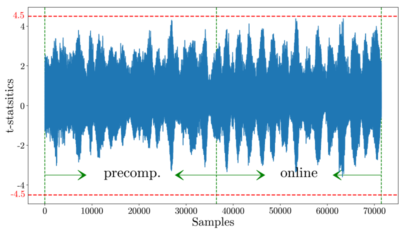
Figure 5(a) depicts the T-test results for AES-128 round function. We mark the phase of precomputation and online-computation in the figures. For comparison, we also provide in Figures 5(b) the result for the implementation when the randomness source is turned off. In Figures 5(c) and 5(d), we provide the T-test results for SKINNY. In summary, our implementations with $d = 1$ do not have any first-order leakage. Another interesting point is that, even if the randomness source is off, the precomputations have no leakage. It is because that the variables in the precomputation are all independent of the secure input, and thus are impossible to leak any information about secret.



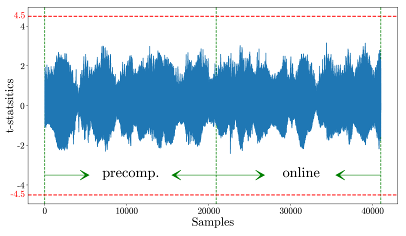
(a) AES, RNG is on



(b) AES, RNG is off



(c) SKINNY-128, RNG is on



(d) SKINNY-64, RNG is on

Figure 5: T-test results

5 Conclusion

In this paper, we continue the long line of works seeking to reduce the overhead of masking. We follow the line of work on the precomputation-based paradigm but focus on the bitsliced implementation that has been shown to be quite efficient for the software implementations. First, we propose a new masked multiplication over the field $\mathbb{F}_q$ for the precomputation-based paradigm, and prove its PINI security. Then, we apply the new masking scheme to AES and SKINNY block ciphers on ARM Cortex M architecture. For SKINNY-64, the speed and RAM requirement can be significantly improved thanks to its smaller block size. The security of our scheme is proved by hand, verified by formal verification tools, and validated by performing T-test evaluation for the masked implementation of AES and SKINNY. We believe a promising future work is to investigate application the masking schemes with precomputation to other crypto-systems such as post-quantum cryptography.

Acknowledgments

The authors would like to thank the reviewers for their helpful comments and suggestions. This work was supported by the National Key Research and Development Program of China (No. 2021YFA1000600), the National Natural Science Foundation of China (Grant Nos. 62002204, 62125204, 92270201 and 61872236), and the Program of Qilu Young Scholars (Grant No 61580082063088) of Shandong University. Yu Yu also acknowledges the support from the XPLORER PRIZE.

References

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers new bitsliced AES speed records on arm-cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021.

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 457–485, 2015.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *ACM CCS '16*, pages 116–129, 2016.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 616–648, 2016.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *EUROCRYPT 2017 (1)*, pages 535–566, 2017.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[BGI$^+$18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[BJK$^+$16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

[BK21] Nicolas Bordes and Pierre Karpman. Fast verification of masking schemes in characteristic two. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 283–312. Springer, 2021.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *J. Cryptol.*, 26(2):280–312, 2013.

[CB08] D. Canright and Lejla Batina. A very compact "perfectly masked" s-box for AES. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, 6th International Conference, ACNS 2008, New York, NY, USA, June 3-6, 2008. Proceedings*, volume 5037 of *Lecture Notes in Computer Science*, pages 446–459, 2008.

[CGZ20] Jean-Sébastien Coron, Aurélien Greuet, and Rina Zeitoun. Side-channel masking with pseudo-random generator. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 342–375. Springer, 2020.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458. Springer, 2014.

[CPR07] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side channel cryptanalysis of a higher order masking scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007.

[CRZ18] Jean-Sébastien Coron, Franck Rondepierre, and Rina Zeitoun. High order masking of look-up tables with common shares. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):40–72, 2018.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 401–429, 2015.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[FGP$^+$18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GJR18] Dahmun Goudarzi, Antoine Joux, and Matthieu Rivain. How to securely compute with noisy leakage in quasilinear complexity. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 547–574. Springer, 2018.

[GMPO20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):152–174, 2020.

[GPRV21] Dahmun Goudarzi, Thomas Prest, Matthieu Rivain, and Damien Vergnaud. Probing security through input-output separation and revisited quasilinear masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):599–640, 2021.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *EUROCRYPT 2017(1)*, pages 567–597, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003*, pages 463–481, 2003.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[RDP08] Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. Block ciphers implementations provably secure against second order side channel analysis. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2008.

[SP06] Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[VV21] Annapurna Valiveti and Srinivas Vivek. Higher-order lookup table masking in essentially constant memory. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):546–586, 2021.

[WGY$^+$22] Weijia Wang, Chun Guo, Yu Yu, Fanjie Ji, and Yang Su. Side-channel masking with common shares. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):290–329, 2022.