IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 523–551. DOI:10.46586/tches.v2025.i1.523-551

Random Probing Security with Precomputation

Bohan Wang[1,2,3], Fanjie Ji[1,3], Yiteng Sun[1,3] and Weijia Wang[2,1,3](✉)

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
wjwang@sdu.edu.cn
[2] Quan Cheng Laboratory, Jinan, China
[3] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, China

Abstract. At Eurocrypt 2014, Duc, Dziembowski and Faust proposed the random probing model to bridge the gap between the probing model proposed at Crypto 2003 and the noisy model proposed at Eurocrypt 2013. Compared with the probing model whose noise in the leakages should (linearly) increase with the number of shares, the random probing model allows each variable leak its value with a probability p, which reflects the physical reality of side channels much better. In Crypto 2020, Belaïd et al. proposed the Random Probing Expandability (RPE) security ensuring the random probing security for arbitrary order masking algorithms with constant leakage probability. However, the complexity of existing RPE algorithms is much higher than that of the probing secure algorithms, which is short of practical usage. In this paper, we investigate the random probing security with precomputation, where a masked cryptographic implementation can be divided into two phases. The first phase, called preprocessing, takes random bits and returns a number of precomputed values. The second phase, called online computation, takes input (e.g., plaintext and shares of secret) and precomputed values to calculate output (e.g., ciphertext) efficiently. We describe a random probing secure precomputable scheme, which transforms an arbitrary circuit compiler with tolerant leakage probability p into a precomputable one by adding a public (but random) share that is calculated in the online phase and the tolerant leakage probability of the new compiler is $\min\{p, 2^{-5.01}\}$. Then, we apply the new scheme to the bitsliced AES. Notably, the implementation under ARM Cortex M architecture shows that the performance of the online phase is significantly improved and even comparable to masking schemes only secure in the probing model.

Keywords: Masking · Random probing model · Precomputation

1 Introduction

Cryptographic algorithms are usually secure against black-box attacks where the adversary is able to get the knowledge of the inputs and outputs. However, the side-channel attack [Koc96, KJJ99] (SCA) allows the adversary to obtain information about intermediate variables by exploiting the physical leakage of the underlying devices, such as the execution time, device temperature, power consumption, etc. Those attacks have posed an important threat to cryptographic devices. Masking [CJRR99, GP99] is one of the most investigated countermeasure to counteract side-channel attacks. A masking scheme splits each intermediate variable (say, x) into n shares (say, $x_1, \ldots, x_n$), satisfying $x = x_1 * \cdots * x_n$ where $*$ is some operation. Particularly, the masking is called a Boolean one if operation $*$ is defined as XOR $\oplus$, which is probably the most widely used case.

To describe the provable security of masking, Ishai, Sahai and Wagner propose the probing security model in [ISW03], which assumes the adversary can get t intermediate

Licensed under Creative Commons License CC-BY 4.0.
Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

variables from an algorithm. Furthermore, the algorithm is t-private secure if the adversary can not get any information of the secret (unsplit variable) from the t variables. Thanks to its (relatively) simplicity in proof, the probing security is widely used in numerous literature (see, e.g., [RP10, CPRR13, Cor14, BBP+16, CGZ20] for an incomplete list).

In [PR13], Prouff and Rivain propose the noisy model, which nicely captures the reality of the embedded devices by assuming all intermediate variables are leaked with a noise. However, the security of the noisy model is believed to be hard to be proved. As a result, Duc et al. proved that the security in the noisy model can be reduced to the security in the probing model [DDF14]. However, one requirement of the reduction is that the noise in the leakages should (mostly linearly) increase with the number of shares, largely restricting the side-channel security in the low-noise scenario. Another security notion for masking, called the random probing model, is proposed as an intermediate leakage model in reducing the noisy model to the probing model. It allows each variable leaks its value with a probability p.

More precisely, it has been proven in [DDF14] that, to maintain the exponential security against SCA, the tolerant leakage probability of a t-probing secure masking algorithm (especially for multiplication) decreases as $\mathcal{O}(\frac{t}{|G|})$, where $|G|$ is the size of the algorithm. Intuitively, the lower the environmental noise is, the higher the leakage probability is. As a result, practically, high-order masking implementations are more likely to be insecure in low-noise cases, even if they are provable security, especially in software implementations. In contrast, schemes in the random probing model with a constant leakage probability maintain security regardless of the number of shares.

In [AIS18], Ananth, Ishai and Sahai provide an expansion strategy on top of the multi-party computation protocol. Their work tolerates a constant leakage probability of 2^{-26} with complexity $\mathcal{O}(\kappa^{8.2})$ where κ is the security parameter of the failure probability $2^{-\kappa}$ of the global circuit [BCP+20]. Then Belaïd et al. extend this expansion strategy to any circuits at [BCP+20], and the improved gadgets achieve the constant tolerant leakage probability of 2^{-8} with the complexity $\mathcal{O}(\kappa^{7.5})$.

Considering the costly complexity of masking algorithms, the precomputation paradigm was proposed to improve the performance of masked implementation/protocol in practice. The first precomputable masking scheme can be traced back to Chari et al. [CJRR99], known as the table-based masking. The other precompution paradigm, which has been widely used for secure multi-party computation [BDOZ11, DPSZ12], has recently emerged in the field of masking [VV21, WGY+22]. It considers a paradigm where the computation can be divided into two phases: precomputation and online phase. The precomputation is independent of the input variables, and it produces some precomputed variables to be temporarily stored in the RAM, which can be performed in the devices' idle time. Then the online phase takes all the input and precomputed variables to calculate the outputs more efficiently. The merit of the above precomputation paradigm is that the precomputation can be done in the idle time of the cryptographic device, significantly accelerating some cryptographic protocols implemented with masking, see the challenge-response protocols described in [WGY+22] for a typical example. More precisely, in these protocols, where one user (Alice) presents a challenge and waits for the other user (Bob) to provide a valid response for authentication, both Alice and Bob can precompute most of the intermediate values and output shares in advance. They only need to compute the final share once the complete challenge or response has been received.

1.1 Our Contribution

To the best of our knowledge, we propose the first precomputable addition, copy and multiplication gadgets with random probing security in this paper, which reduces the online

complexity from $\mathcal{O}(n^{2.77})$ [BCP+20] to $\mathcal{O}(n)$. [1] More precisely, there are two contributions.

First, we propose a scheme that transforms an arbitrary circuit compiler into a precomputable one by adding a public (but random) share that is calculated in the online phase. We show that the gadgets in the scheme satisfy the random probing composability (RPC) [BCP+20] with tolerant leakage probability $\min\{p, 2^{-5.01}\}$ and linear memory usage in the execution, where p is the tolerant leakage probability of the initial circuit compiler. We illustrate the concept in Figure 1 and compare it with the state-of-the-art works in Table 1.

Table 1: Comparison in leakage probability and online complexity between the state-of-the-art works and ours.

Works	[BCP+20]	[BRT21]	[BRTV21]	[CFOS21]	Ours
Leakage probability	Constant	Constant	Constant	Not constant	Constant[3]
Online complexity	$\mathcal{O}(n^{2.77})$	$\mathcal{O}(n^{2.47})^1$	$\mathcal{O}(n^{2.55})^2$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Code size (pre/online)	–	–	–	–	22.7/79.9 KB

1 Similar to [BCP+20], the complexity is $\mathcal{O}(15^k)$ for 3^k-share gadgets, i.e. $\mathcal{O}(15^k) = \mathcal{O}\big((3^k)^{2.47}\big) = \mathcal{O}(n^{2.47})$ for n-share gadgets.

2 Similar to [BCP+20], the complexity is $\mathcal{O}\big((18\log 3 - 12)^k\big)$ for 3^k-share gadgets, i.e. $\mathcal{O}\big((18\log 3 - 12)^k\big) = \mathcal{O}\big((3^k)^{2.55}\big) = \mathcal{O}(n^{2.55})$ for n-share gadgets.

3 The failure probability remains constant if the leakage probability of the initial circuit compiler is constant, and vice versa.

Then, we apply the above gadgets in bitsliced AES, and compare the result of cycle counts, random cost and memory usage between the state-of-the-art works of precomputation and random probing security and our work. Our work makes the random probing security more practical. The implementation under ARM Cortex M architecture shows that the performance of the online phase is significantly improved and even comparable to masking schemes only secure in the probing model. For example, when $n = 9$, the online computation of our random probing secure scheme runs in 578 000 cycles, while the well-known result of the scheme in the probing model is 404 500 cycles.

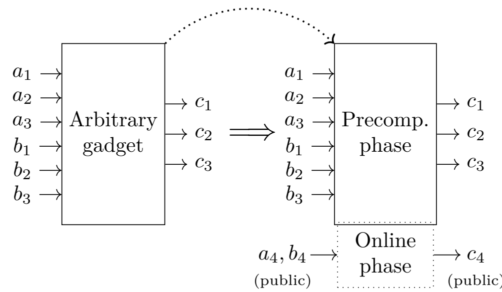


Figure 1: Illustration of the precomputable scheme secure in the random probing model with $n = 3$, which transforms an arbitrary gadget into a precomputable one by adding a public (but random) share.

1.2 Roadmap

We recall the definitions of the random probing model and propose a quantitative criterion of precomputation in Section 2. Section 3 introduces the precomputable gadgets with public share and shows their low memory usage and Section 4 shows their security. Then,

[1] To be exact, the complexity of works in [BCP+20] is $\mathcal{O}(21^k)$ for 3^k-share gadgets, namely $\mathcal{O}(21^k) = \mathcal{O}\big((3^k)^{2.77}\big) = \mathcal{O}(n^{2.77})$ for n-share gadgets.

we implement the bitsliced AES with our new gadgets and compare them with other works in Section 5.1. Finally, we summarize our works in Section 6.

2 Preliminaries

2.1 Notations

Along the paper, $\mathbb{K}$ shall denote a finite field, and we define $+$ as the field addition and $\cdot$ as the field multiplication over $\mathbb{K}$. For any $n \in \mathbb{N}$, we shall denote $[n] = [1, n] \cap \mathbb{Z}$. For any tuple $\hat{x} = (x_1, \ldots, x_n) \in \mathbb{K}^n$ and any set $I \subseteq [n]$, we shall denote $x_{|I} = (x_i)_{i \in I}$. For simplicity, we define $a_{[n]} = a_{|[n]}$, and $a_{[n]} + b_{[n]} = \{a_i + b_i | i \in [n]\}$. For any variable r, $r \leftarrow \$$ means that r is a uniformly random variable. Any two probability distributions D_1 and D_2 are said *ϵ-close*, denoted as $D_1 \approx_\epsilon D_2$, if their statistical distance is upper bounded by ϵ, that is

$$\frac{1}{2} \sum_x |p_{D_1}(x) - p_{D_2}(x)| \leqslant \epsilon \ ,$$

where $p_{D_1}(\cdot)$ and $p_{D_2}(\cdot)$ denote the probability mass functions of D_1 and D_2.

2.2 Circuit and Circuit Compiler

An arithmetic circuit on a field $\mathbb{K}$ is a labeled directed acyclic graph whose edges are wires and vertices are arithmetic gates processing operations on $\mathbb{K}$ [BRT21]. In this paper, we consider circuits composed of addition gates

$$G_{\text{add}} : \mathbb{K}^2 \to \mathbb{K} \text{ and } G_{\text{add}}(x_1, x_2) = x_1 + x_2 \ ,$$

multiplication gates

$$G_{\text{mult}} : \mathbb{K}^2 \to \mathbb{K} \text{ and } G_{\text{mult}}(x_1, x_2) = x_1 \cdot x_2 \ ,$$

and copy gates

$$G_{\text{copy}} : \mathbb{K} \to \mathbb{K}^2 \text{ and } G_{\text{copy}}(x) = (x, x) \ .$$

A randomized arithmetic circuit is equipped with an additional random gate that outputs a fresh uniform random value of $\mathbb{K}$ [BRT21].

Definition 1 (Circuit Compiler [BCP+20])**.** A circuit compiler is a triplet of algorithms $(\mathsf{CC}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows:

- CC (circuit compilation) is a deterministic algorithm that takes as input an arithmetic circuit C and outputs a randomized arithmetic circuit $\hat{C}$.
- Enc (input encoding) is a probabilistic algorithm that maps an input $\mathbf{x} \in \mathbb{K}^\alpha$ to an encoded input $\hat{\mathbf{x}} \in \mathbb{K}^{\alpha'}$ called sharing.
- Dec (output decoding) is a deterministic algorithm that maps an encoded output $\hat{\mathbf{y}} \in \mathbb{K}^{m'}$ to a plain output $\mathbf{y} \in \mathbb{K}^m$.

These three algorithms satisfy the following properties:

- **Correctness** : For every arithmetic circuit C of input length α, and for every $\mathbf{x} \in \mathbb{K}^\alpha$, we have

$$\Pr\Big(\mathsf{Dec}\big(\hat{C}(\hat{x})\big) = C(x) | \hat{x} \leftarrow \mathsf{Enc}(x)\Big) = 1, \ \ \textit{where } \hat{C} = \mathsf{CC}(C) \ .$$

- **Efficiency** : For some security parameter $\lambda \in \mathbb{N}$, the running time of $\mathsf{CC}(C)$ is $\text{poly}(\lambda, |C|)$, the running time of $\mathsf{Enc}(x)$ is $\text{poly}(\lambda, |x|)$ and the running time of $\mathsf{Dec}(\hat{y})$ is $\text{poly}(\lambda, |\hat{y}|)$, where $\text{poly}(\lambda, q)=O(\lambda^{k_1} q^{k_2})$ for some constants k_1, k_2.

Note that the circuit compiler was first introduced in [ISW03], and we use a clearer expression proposed in [BCP+20]. In the following, the n-linear decoding mapping, denoted as $\mathsf{Dec} : \mathbb{K}^n \to \mathbb{K}$, is defined as

$$\mathsf{Dec}(x_1, \ldots, x_n) = x_1 + \cdots + x_n$$

for every $n \in \mathbb{N}$ and $(x_1, \ldots, x_n) \in \mathbb{K}^n$. We shall further consider that, for every $n, \alpha \in \mathbb{N}$, on input $(\hat{x}_1, \ldots, \hat{x}_\alpha) \in (\mathbb{K}^n)^\alpha$ the n-linear decoding mapping acts as

$$\mathsf{Dec}(\hat{x}_1, \ldots, \hat{x}_\alpha) = \big(\mathsf{Dec}(\hat{x}_1), \ldots, \mathsf{Dec}(\hat{x}_\alpha)\big) \ .$$

Thanks to these mappings, we shall define gadgets in the following, which was proposed in [BCP+20].

Definition 2 (Gadget [BCP+20])**.** An *n-share, α-to-m gadget* is denoted by a randomized arithmetic circuit that maps an input $\hat{\mathbf{x}} \in (\mathbb{K}^n)^\alpha$ to an output $\hat{\mathbf{y}} \in (\mathbb{K}^n)^m$ such that $\mathbf{x} = \mathsf{Dec}(\hat{\mathbf{x}}) \in \mathbb{K}^\alpha$ and $\mathbf{y} = \mathsf{Dec}(\hat{\mathbf{y}}) \in \mathbb{K}^m$ satisfy $\mathbf{y} = g(\mathbf{x})$ for some function g.

Generally, CC works by replacing each gate of the input circuit with a corresponding gadget, and in the rest of the paper these gadgets are called *base gadgets*. In addition, we define the *linear transformation gadget* in Definition 3, in which linear operations such as multiplication with constants and bitshift are contained.

Definition 3 (Linear Transformation Gadget)**.** Let $\mathrm{L} : \mathbb{K}^n \to \mathbb{K}^n$ be an n-share gadget with input $a_{[n]}$. Then L is a linear transformation gadget if for any $\mathrm{L}(a_{[n]}) = b_{[n]}$, there is $\mathrm{L}(a_i) = b_i$ for $i \in [n]$.

2.3 Random Probing Security

In [BCP+20], Belaïd et al. proposed the formal random probing security, which assumes every wire in the circuit leaks with a probability p. Compared with the probing model proposed in [ISW03], the random probing model is closer to the SCA in the real world. Moreover, we will introduce it in this section. Figure 2 shows the security proof flow of the paper.

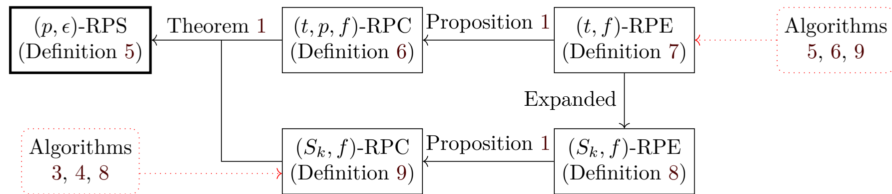


Figure 2: Overview of the security proof flow.

We start with the random probing leakage proposed in [BCP+20], which describes the leakage formally.

Definition 4 (Random Probing Leakage [BCP+20])**.** The p-random probing leakage of a randomized arithmetic circuit C on input $\mathbf{x} \in \mathbb{K}$ is the distribution $\mathcal{L}_p(C, \mathbf{x})$ obtained by composing the leaking-wires and assign-wires samplers as

$$\mathcal{L}_p(C, \mathbf{x}) \stackrel{id}{=} \mathsf{AssignWires}\big(C, \mathsf{LeakingWires}(C, p), \mathbf{x}\big)$$

with

- the leaking-wires sampler

$$\mathcal{W} \leftarrow \mathsf{LeakingWires}(C, p) \enspace ,$$

  where $\mathcal{W}$ is constructed by including each wire label from the circuit C with probability p to $\mathcal{W}$ (where all the probabilities are mutually independent).

- the assign-wires sampler takes as input a randomized arithmetic circuit C, a set of wire labels $\mathcal{W}$ (subset of the wire labels of C), and an input $\mathbf{x}$, and it outputs a $|\mathcal{W}|$-tuple $\mathbf{w} \in (\mathbb{K} \cup \{\bot\})^{|\mathcal{W}|}$, denoted as

$$\mathbf{w} \leftarrow \mathsf{AssignWires}(C, \mathcal{W}, \mathbf{x}) \enspace ,$$

  where $\mathbf{w}$ corresponds to the assignments of the wires of C with label in $\mathcal{W}$ for an evaluation on input $\mathbf{x}$.

Definition 5 ((p, ϵ)-RPS (Random Probing Secure) [BCP+20])**.** A randomized arithmetic circuit C with $\alpha \cdot n \in \mathbb{N}$ input gates is (p, ϵ)-random probing secure with respect to encoding Enc if there exists a simulator Sim such that for every $x \in \mathbb{K}^\alpha$:

$$\mathsf{Sim}(C) \approx_\epsilon \mathcal{L}_p(C, \mathsf{Enc}(x)) \enspace .$$

Although Random Probing Security (RPS) is a quantifiable security notion for random probing security, it is rarely used directly in security proofs for masking implementations against random probing adversaries. This is because the number of verified sets in the proof increases exponentially with the circuit size. Belaïd et al. introduced Random Probing Composability (RPC) security to address this issue. Thanks to the composability of RPC, we can claim the RPS security of a circuit by dividing the circuit into several gadgets and proving the RPC security of each gadget. This approach significantly reduces the verification workload.

Definition 6 (Random Probing Composability [BCP+20])**.** Let $n, \alpha, m \in \mathbb{N}$. An n-share gadget $G : (\mathbb{K}^n)^\alpha \to (\mathbb{K}^n)^m$ is (t, p, ϵ)-random probing composable (RPC) for some $t \in \mathbb{N}$ and $p, \epsilon \in [0, 1]$ if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $\hat{x} \in (\mathbb{K}^n)^\alpha$ and for every set collection $J_1 \subseteq [n], \ldots, J_m \subseteq [n]$ of cardinals $|J_1| \leqslant t, \ldots, |J_m| \leqslant t$, the random experiment

$$\begin{aligned} &\mathcal{W} \leftarrow \mathsf{LeakingWires}(G, p) \enspace , \\ &\mathbf{I} \leftarrow \mathsf{Sim}_1^G(\mathcal{W}, \mathbf{J}) \enspace , \\ &out \leftarrow \mathsf{Sim}_2^G(\hat{x}_{|\mathbf{I}}) \end{aligned}$$

yields

$$\Pr\big((|I_1| > t) \vee \cdots \vee (|I_\alpha| > t)\big) \leqslant \epsilon$$

and

$$out \stackrel{id}{=} \big(\mathsf{AssignWires}(G, \mathcal{W}, \hat{x}), \hat{y}_{|\mathbf{J}}\big) \enspace ,$$

where $\mathbf{I} = (I_1, \ldots, I_\alpha), \mathbf{J} = (J_1, \ldots, J_m)$ and $\hat{y} = G(\hat{x})$. Let $f : \mathbb{R} \to \mathbb{R}$. The gadget G is (t, f)-RPC if it is $(t, p, f(p))$-RPC for every $p \in [0, 1]$.

Theorem 1 introduces the compositional security of RPC gadgets. Compared to the original theorem in [BCP+20], we add the conclusion that the composition of RPC gadgets is also an RPC gadget. This conclusion can be directly proven from the proof of Theorem 1 in [BCP+20].

Theorem 1 ([BCP+20], adapted)**.** *Let $t \in \mathbb{N}, p, \epsilon \in [0,1]$ and CC be a standard circuit compiler with (t,p,ϵ)-RPC base gadgets. For every (randomized) arithmetic circuit C composed of $|C|$ gadgets, the compiled circuit $\mathsf{CC}(C)$ is $(t,p,|C|\cdot\epsilon)$-RPC and $(p,|C|\cdot\epsilon)$-RPS. Equivalently, the standard circuit compiler CC is (p,ϵ)-RPS.*

Although RPC gadgets are composable, the calculation of ϵ is quite complicated (intuitively, it is more challenging than the verification proof of probing security). In [AIS18], Ananth, Ishai and Sahai propose a modular approach to build an RPS circuit compiler from a secure multiparty protocol. This approach was later formalized as the notion of the expanding compiler [BCP+20], and it is proven to achieve RPC if the base gadgets verify a property called random probing expandability (RPE) [BCP+20]. Intuitively, for a (t,f)-RPE gadget, there are two requirements for its leakage simulation:

- If there are no more than t output leaking, the failure (i.e., the size of either input indices set for simulation is bigger than t) probability of the simulation for any leakage is $f(p)$;
- If there are more than t output leaking, the failure probability of the simulation for any internal leakage and $n-1$ output wires selected on the internal leakage is $f(p)$.

The first requirement ensures the composability of gadgets, while the second ensures expandability (as reflected in the proof of Proposition 1, seen in Appendix C of [BCP+20]).

Definition 7 ((t,f)-RPE [BCP+20])**.** Let $f:\mathbb{R}\to\mathbb{R}$. An n-share gadget $\mathrm{G}:(\mathbb{K}^n)^2\to\mathbb{K}^n$ is (t,f)-RPE for some $p\in[0,1]$, if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $(\hat{x},\hat{y})\in(\mathbb{K}^n)^2$ and for every set $J\subseteq[n]$, the random experiment

$$\begin{aligned}
&\mathcal{W}\leftarrow \mathsf{LeakingWires}(G,p)\ ,\\
&(I_1,I_2,J')\leftarrow \mathsf{Sim}_1^G(\mathcal{W},J)\ ,\\
&out\leftarrow \mathsf{Sim}_2^G(\mathcal{W},J',\hat{x}_{|I_1},\hat{y}_{|I_2})
\end{aligned}$$

ensures that

1. the failure events $\mathcal{F}_1\equiv(|I_1|>t)$ and $\mathcal{F}_2\equiv(|I_2|>t)$ verify

   $$\Pr(\mathcal{F}_1)=\Pr(\mathcal{F}_2)=\epsilon \ \ and \ \ \Pr(\mathcal{F}_1\wedge\mathcal{F}_2)=\epsilon^2$$

   with $\epsilon=f(p)$ (in particular $\mathcal{F}_1$ and $\mathcal{F}_2$ are mutually independent),
2. J' is such that $J'=J$ if $|J|\leqslant t$ and $J'\subseteq[n]$ with $|J'|=n-1$ otherwise,
3. the output distribution satisfies

   $$out \overset{id}{=} \Big(\mathsf{AssignWires}\big(G,\mathcal{W},(\hat{x},\hat{y})\big),\hat{z}_{|J'}\Big)\ ,$$

   where $\hat{z}=G(\hat{x},\hat{y})$.

Moreover, the failure probability of RPE gadgets can be estimated by the expansion times k. Generally, unmasked circuits are compiled once in the probing model. However, according to the modular approach [AIS18], they would be compiled several times, meaning

the operations where the masking gadgets replace gates in the compiled circuits would repeat multiple times. We refer to the number of compilations as the expansion times of gadgets. Intuitively, a gadget is RPC and expandable if it is RPE, and its security level (described by the failure probability against the random probing adversary) increases exponentially with the expansion times while maintaining a constant leakage probability.

For simplicity, let $\tilde{G}(a_{[n^k]}, b_{[n^k]}, k)$ be the k-time expansion of the n-share gadget G with input sharings $a_{[n]}, b_{[n]}$ and $\tilde{G}(k)$ for short. The gadgets with upper tilde also occur in Section 3.2 and they are the same format as $\tilde{G}(a_{[n^k]}, b_{[n^k]}, k)$. For a (t, f)-RPE gadget G, the security of $\tilde{G}(k)$ is also introduced in [BCP+20], called $(S_k, f^{(k)})$-RPE, where

$$f^{(k)}(p) = \underbrace{f \circ f \circ \cdots \circ f}_{\text{k times}}(p) \ .$$

Definition 8 ((S_k, f)-RPE [BCP+20])**.** Let $f : \mathbb{R} \to \mathbb{R}$ and $k \in \mathbb{N}$. An n^k-share gadget $G : \mathbb{K}^{n^k} \to \mathbb{K}^{n^k}$ is (S_k, f)-RPE if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $(\hat{x}, \hat{y}) \in \mathbb{K}^{n^k} \times \mathbb{K}^{n^k}$, for every set $J \in S_k \cup [n^k]$ and for every $p \in [0, 1]$, the random experiment

$$\begin{aligned} &\mathcal{W} \leftarrow \mathsf{LeakingWires}(G, p) \ , \\ &(I_1, I_2, J') \leftarrow \mathsf{Sim}_1^G(\mathcal{W}, J) \ , \\ &out \leftarrow \mathsf{Sim}_2^G(\mathcal{W}, J', \hat{x}_{|I_1}, \hat{y}_{|I_2}) \end{aligned}$$

ensures that

1. the failure events $\mathcal{F}_1 \equiv (I_1 \notin S_k)$ and $\mathcal{F}_2 \equiv (I_1 \notin S_k)$ verify

$$\Pr(\mathcal{F}_1) = \Pr(\mathcal{F}_2) = \epsilon \ \text{and} \ \Pr(\mathcal{F}_1 \wedge \mathcal{F}_2) = \epsilon^2$$

   with $\epsilon = f(p)$ (in particular $\mathcal{F}_1$ and $\mathcal{F}_2$ are mutually independent),

2. J' is such that $J' = J$ if $J \in S_k$ and $J' = [n^k]/\{j^\star\}$ for some $j^\star \in [n^k]$ otherwise,

3. the output distribution satisfies

$$out \overset{id}{=} \Big(\mathsf{AssignWires}\big(G, \mathcal{W}, (\hat{x}, \hat{y})\big), \hat{z}_{|J'}\Big) \ ,$$

   where $\hat{z} = G(\hat{x}, \hat{y})$,

where

$$\begin{aligned} &S_1 = \{I \in [n], |I| \leqslant t\} \ , \\ &S_k = \{(I_1, \ldots, I_n) \in (S_{k-1} \cup [n^{k-1}])^n, I_j \in S_{k-1} \ \forall j \in [n] \text{ except at most } t\} \ . \end{aligned}$$

Next, we introduce the formal expression of the relationship between RPE and RPC, as well as the relationship between normal RPE gadgets and expanded RPE gadgets, in Proposition 1.

Proposition 1 ([BCP+20])**.** *Let $f : \mathbb{R} \to \mathbb{R}$ and $n \in \mathbb{N}$. Let G be an n-share gadget. If G is (t, f)-RPE, then*

- *G is $(t, 2 \cdot f)$-RPC.*
- *$\tilde{G}(k)$ is $(S_k, f^{(k)})$-RPE.*

Correspondingly we have the definition of (S_k, f)-RPC to describe the composable security of $\tilde{G}(k)$ with the (t, f)-RPE G.

Definition 9 ((S_k, f)-RPC)**.** Let $n, \alpha, m, k \in \mathbb{N}$. An n^k-share gadget $G : (\mathbb{K}^{n^k})^\alpha \to (\mathbb{K}^{n^k})^m$ is (S_k, f)-RPC for some $p, f(p) \in [0,1]$ if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $\hat{x} \in (\mathbb{K}^{n^k})^\alpha$ and for every set collection $J_1 \in S_k, \ldots, J_m \in S_k$, the random experiment

$$\begin{aligned} &\mathcal{W} \leftarrow \mathsf{LeakingWires}(G, p) \ , \\ &\mathbf{I} \leftarrow \mathsf{Sim}_1^G(\mathcal{W}, \mathbf{J}) \ , \\ &out \leftarrow \mathsf{Sim}_2^G(\hat{x}_{|\mathbf{I}}) \end{aligned}$$

yields

$$\Pr\big((I_1 \notin S_k) \vee \cdots \vee (I_\alpha \notin S_k)\big) \leqslant f(p)$$

and

$$out \overset{id}{=} \big(\mathsf{AssignWires}(g, \mathcal{W}, \hat{x}), \hat{y}_{|\mathbf{J}}\big) \ ,$$

where $\mathbf{J} = (J_1, \ldots, J_m)$ and $\hat{y} = G(\hat{x})$.

Then according to Proposition 1, $\tilde{G}(k)$ is $(S_k, 2 \cdot f^{(k)})$-RPC. Nevertheless, Theorem 1 also works with (S_k, f)-RPC gadgets. By the definition of RPE, [BCP+20] provides the method to get a compiled circuit with any failure probability $2^{-\kappa}$:

1. Construct (t, f)-RPE addition gadget Add, copy gadget Copy and multiplication gadget Mul.
2. Generate $\widetilde{\mathsf{Add}}(k), \widetilde{\mathsf{Copy}}(k)$ and $\widetilde{\mathsf{Mul}}(k)$ with failure probability $f^{(k)}$.
3. Replace the gates of the original circuit G with $\widetilde{\mathsf{Add}}(k), \widetilde{\mathsf{Copy}}(k)$ and $\widetilde{\mathsf{Mul}}(k)$ such that
$$|G| \cdot 2 \cdot f^{(k)} \leqslant 2^{-\kappa} \ .$$

To quantitate the efficiency of RPE gadgets, [BCP+20] defines the amplification order d of the failure probability $\epsilon = f(p)$ of these gadgets. As an intuition, higher amplification order refers to better security.

Definition 10 (Amplification order [BCP+20])**.**

- Let $f : \mathbb{R} \to \mathbb{R}$ which satisfies
$$f(p) = c_d p^d + \mathcal{O}(p^{d+\epsilon})$$
as p tends to 0, for some $c_d > 0$ and $\epsilon > 0$. Then d is called the amplification order of f.
- Let $t > 0$ and G a gadget. Let d be the maximal integer such that G achieves (t, f)-RPE or (t, f)-RPC for $f : \mathbb{R} \to \mathbb{R}$ of amplification order d. Then d is called the amplification order of G (with respect to t).

As shown in [BCP+20], the complexity of the expanded gadgets relates to the (minimum) amplification order of the three gadgets used in the base compiler CC. Suppose it achieves (t, f)-RPE with an amplification order d. In that case, the expanding compiler achieves $(p, 2^{-\kappa})$-RPS with a complexity blowup of $\mathcal{O}(\kappa^e)$ for an exponent e satisfying

$$e = \frac{\log N_{\max}}{\log d}$$

with

$$N_{\max} = \max\left(N_{m,m}, \mathsf{eigenvalues}\left(\begin{pmatrix} N_{a,a} & N_{c,a} \\ N_{a,c} & N_{c,c} \end{pmatrix}\right)\right) \ ,$$

where $N_{x,y}$ denotes the number of gates "x" in a gadget "y", with "m" meaning multiplication, "a" meaning addition, and "c" meaning copy. They are used in Section 3.5.

Naturally, achieving the upper bound of the amplification order refers to the smallest complexity. In [BRT21], the generic upper bound on the amplification order is given.

Lemma 1 (Upper bound of the amplification order for (t, f)-RPE [BRT21])**.** *Let $f : \mathbb{R} \to \mathbb{R}, n \in \mathbb{N}$ and $\alpha, m \in \{1, 2\}$. Let $G : (\mathbb{K}^n)^\alpha \to (\mathbb{K}^n)^m$ be an α-to-m n-share complete gadget achieving (t, f)-RPE. Then its amplification order d is upper bounded by*

$$\min\big((t+1), (3-\alpha)\cdot(n-t)\big) \ .$$

3 Precomputation with Public Shares

In this section, we provide the precomputation method with public shares, in which we can use any gadgets with n precomputation shares and the additional online share is *public* (i.e., treated as constant). Besides, the security of the n-share gadgets and the correctness of the $(n+1)$-share operation are kept.

3.1 Construction for the RPC Gadgets with Public Shares

We introduce the overview of the method in the following and let $\ell = 3^k$ in the rest of the paper.

1. First, we generate the ℓ-share private circuit with the circuit compiler ($\mathsf{Add}, \mathsf{Copy}, \mathsf{Mul}$), where $\mathsf{Add}, \mathsf{Copy}, \mathsf{Mul}$ are (S_k, f)-RPC. Besides, Copy satisfies $\sum_{i\in[\ell]} a_i = \sum_{i\in[\ell]} b_i = \sum_{i\in[\ell]} c_i$ with input sharing $a_{[\ell]}$ and output sharings $b_{[\ell]}, c_{[\ell]}$, Add satisfies $\sum_{i\in[\ell]} a_i + \sum_{i\in[\ell]} b_i = \sum_{i\in[\ell]} c_i$ with input sharings $a_{[\ell]}, b_{[\ell]}$ and output sharing $c_{[\ell]}$, and Mul satisfies $\sum_{i\in[\ell]} a_i \cdot \sum_{i\in[\ell]} b_i = \sum_{i\in[\ell]} c_i$ with input sharings $a_{[\ell]}, b_{[\ell]}$ and output sharing $c_{[\ell]}$.
2. Then we bring into the public shares with index $\ell + 1$.
   - We calculate output sharing $c_{[\ell]}$ for Add with precomputation sharings $a_{[\ell]}, b_{[\ell]}$, and $c_{\ell+1} \leftarrow a_{\ell+1} + b_{\ell+1}$ as the public output directly since the addition of $a_{\ell+1}$ and $b_{\ell+1}$ need not satisfy any RPE or RPC security. We call the composed addition gadget Add_p.
   - Let $b_{\ell+1}, c_{\ell+1} \leftarrow a_{\ell+1}$ with public outputs $b_{\ell+1}, c_{\ell+1}$ and public input $a_{\ell+1}$ for the precomputable gadget Copy_p whose precomputed outputs $b_{[\ell]}$ and $c_{[\ell]}$ are generated by Copy with input $a_{[\ell]}$.
   - As for Mul, we compose it with other gadgets such that the composed gadget (called Mul_p) satisfies $\sum_{i\in[\ell+1]} a_i \cdot \sum_{i\in[\ell+1]} b_i = \sum_{i\in[\ell+1]} c_i$.

In Figure 3, we show the general constructions of Add_p, Copy_p and Mul_p. We stress that the addition of $a_{\ell+1}$ and $b_{\ell+1}$ in Add_p (resp. $a_{\ell+1}$ in Copy_p) is not an RPE or RPC addition (resp. copy) but a normal one.

As for Mul_p, we note that the gadgets composed with Mul are all linear transformation gadgets, and the detailed construction is stated in the next section. Considering that Add_p and Mul_p are composable, we show their (S_k, f)-RPC security in Section 3.2.

3.2 Circuit Compiler with Public Shares

In this section, we introduce the gadgets described in Section 3.1, i.e., $\mathsf{Add}_p, \mathsf{Copy}_p$ and Mul_p. Additionally, we provide Lin_p to ensure the security of linear transformations in

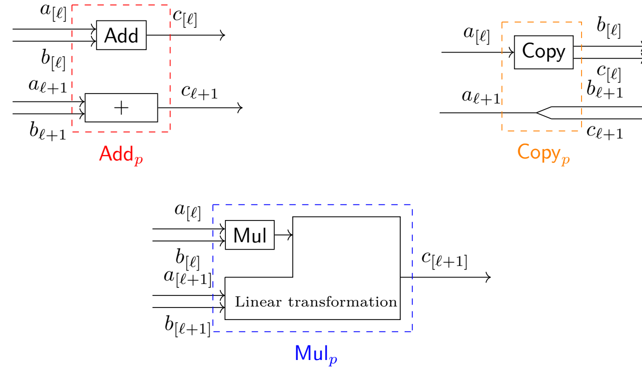


Figure 3: Brief introduction of Add_p, Copy_p and Mul_p mentioned above.

Algorithm 1 Addition Gadget $\mathsf{Add}_p(a_{[\ell+1]}, b_{[\ell+1]})$

Input: input sharings $a_{[\ell]}, b_{[\ell]}$ and the public inputs $a_{\ell+1}, b_{\ell+1}$
Output: output sharing $c_{[\ell]}$ and the public output $c_{\ell+1}$ where $\sum_{i\in[\ell+1]}(a_i + b_i) = \sum_{i\in[\ell+1]} c_i$

Precomputation Phase

Input: $a_{[\ell]}, b_{[\ell]}$
Output: $c_{[\ell]}$
1: $c_{[\ell]} \leftarrow \mathsf{Add}(a_{[\ell]}, b_{[\ell]})$
Precomputed storage: Null

Online Phase

Input: $a_{\ell+1}, b_{\ell+1}$
Output: $c_{\ell+1}$
1: $c_{\ell+1} \leftarrow a_{\ell+1} + b_{\ell+1}$

Algorithm 2 Copy Gadget $\mathsf{Copy}_p(a_{[\ell+1]}, b_{[\ell+1]})$

Input: input sharing $a_{[\ell]}$ and the public input $a_{\ell+1}$
Output: output sharings $b_{[\ell]}, c_{[\ell]}$ and the public outputs $b_{\ell+1}, c_{\ell+1}$ where $\sum_{i\in[\ell+1]} a_i = \sum_{i\in[\ell+1]} b_i = \sum_{i\in[\ell+1]} c_i$

Precomputation Phase

Input: $a_{[\ell]}$
Output: $b_{[\ell]}, c_{[\ell]}$
1: $c_{[\ell]} \leftarrow \mathsf{Copy}(a_{[\ell]}, b_{[\ell]})$
Precomputed storage: Null

Online Phase

Input: $a_{\ell+1}$
Output: $b_{\ell+1}, c_{\ell+1}$
1: $b_{\ell+1}, c_{\ell+1} \leftarrow a_{\ell+1}$

---

the masking implementation. First, we present $\mathsf{Add}_p, \mathsf{Copy}_p$ in Algorithms 1, 2. Notably, neither algorithm uses memory, and their online complexity is $\mathcal{O}(1)$.

Then, we introduce Mul_p at Algorithm 3 illustrated in Figure 4. Figure 3 illustrates

that Mul_p calls an RPC multiplication gadget to generate the first ℓ output shares through some precomputable operations. Additionally, the input sharings $a_{[\ell]}, b_{[\ell]}$, along with some other precomputed variables, are used to calculate the public output share $c_{\ell+1}$ during the online phase with the public input shares $a_{\ell+1}, b_{\ell+1}$. Moreover, all the online operations of Mul_p are contained in $\mathsf{Submult}$ (i.e., Algorithm 4), so we do not explicitly distinguish the online and precomputation phases in Algorithm 3.

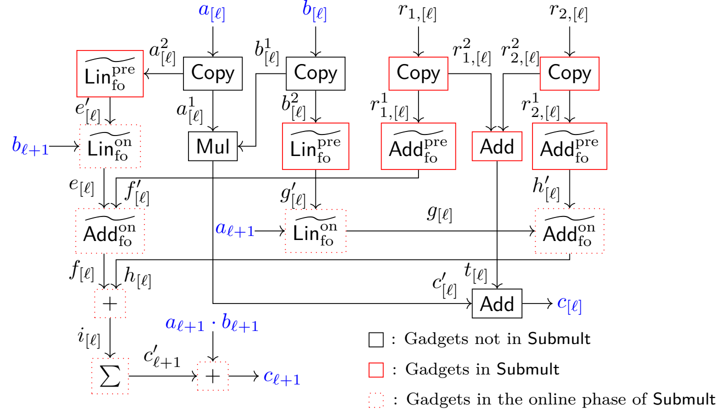


Figure 4: Illustration of Mul_p. The blue sharings are the inputs and output of Mul_p, including both the secret ones and the public ones.

Besides, we stress that the calculations for $i_{[\ell]}$ and $c_{\ell+1}$ (namely steps 6, 7 and 8 of the online phase in $\mathsf{Submult}$) are trivially expanded. More precisely, there are directly $i_j \leftarrow f_j + h_j$ for $j \in [\ell]$ and $c_{\ell+1} \leftarrow \sum_{j\in[\ell]} i_j + a_{\ell+1} \cdot b_{\ell+1}$, without any additional randoms and tricks.

Algorithm 3 Multiplication Gadget $\mathsf{Mul}_p(a_{[\ell+1]}, b_{[\ell+1]})$

Input: the input sharings $a_{[\ell]}, b_{[\ell]}$ and the public inputs $a_{\ell+1}, b_{\ell+1}$
Output: output sharing $c_{[\ell]}$ and the public output $c_{\ell+1}$ where $\sum_{i\in[\ell+1]} a_i \cdot \sum_{i\in[\ell+1]} b_i = \sum_{i\in[\ell+1]} c_i$

1: $(a^1_{[\ell]}, a^2_{[\ell]}) \leftarrow \mathsf{Copy}(a_{[\ell]})$
2: $(b^1_{[\ell]}, b^2_{[\ell]}) \leftarrow \mathsf{Copy}(b_{[\ell]})$
3: $c'_{[\ell]} \leftarrow \mathsf{Mul}(a^1_{[\ell]}, b^1_{[\ell]})$
4: $(t_{[\ell]}, c_{\ell+1}) \leftarrow \mathsf{Submult}(a^2_{[\ell]}, b^2_{[\ell]}, a_{\ell+1}, b_{\ell+1})$
5: $c_{[\ell]} \leftarrow \mathsf{Add}(t_{[\ell]}, c'_{[\ell]})$

Considering that the expansion of the invoked gadgets in $\mathsf{Submult}$ is different from the present constructions (because one of their input sharing would be all online shares), we introduce Add_{fo} and Lin_{fo} for the expansion in $\mathsf{Submult}$. Meanwhile, the precomputation phases of $\mathsf{Add}_{\text{fo}}, \mathsf{Lin}_{\text{fo}}$ are called $\mathsf{Add}^{\text{pre}}_{\text{fo}}, \mathsf{Lin}^{\text{pre}}_{\text{fo}}$ respectively, while the online phases are called $\mathsf{Add}^{\text{on}}_{\text{fo}}, \mathsf{Lin}^{\text{on}}_{\text{fo}}$ respectively.

Besides, we introduce Lin_p as the linear transformation gadget in Algorithm 8.

3.3 The Memory Usage for the Gadgets with Public Shares

In this section, we introduce the memory usage and online complexity for the gadgets in Section 3.1. Considering that there is no memory usage for $\mathsf{Add}_p, \mathsf{Copy}_p$ and Lin_p, we provide the memory usage of Mul_p and its related online gadgets e.g., Add_{fo} and Lin_{fo}.

Algorithm 4 $\mathsf{Submult}(a_{[\ell+1]}, b_{[\ell+1]})$

Input: input sharings $a_{[\ell]}, b_{[\ell]}$ and the public inputs $a_{\ell+1}, b_{\ell+1}$
Output: output sharing $t_{[\ell]}$ and the public output $c_{\ell+1}$

Precomputation Phase

Input: $a_{[\ell]}, b_{[\ell]}$
Output: $e'_{[\ell]}, f'_{[\ell]}, g'_{[\ell]}, h'_{[\ell]}$ and $t_{[\ell]}$

1: $r_{[2],[\ell]} \leftarrow \$$
2: $(r^1_{1,[\ell]}, r^2_{1,[\ell]}) \leftarrow \mathsf{Copy}(r_{1,[\ell]})$
3: $(r^1_{2,[\ell]}, r^2_{2,[\ell]}) \leftarrow \mathsf{Copy}(r_{2,[\ell]})$
4: $t_{[\ell]} \leftarrow \mathsf{Add}(r^2_{1,[n]}, r^2_{2,[\ell]})$
5: $e'_{[\ell]} \leftarrow \widetilde{\mathsf{Lin}}^{\mathrm{pre}}_{\mathrm{fo}}(a_{[\ell]}, k)$
6: $f'_{[\ell]} \leftarrow \widetilde{\mathsf{Add}}^{\mathrm{pre}}_{\mathrm{fo}}(r^1_{1,[\ell]}, k)$
7: $g'_{[\ell]} \leftarrow \widetilde{\mathsf{Lin}}^{\mathrm{pre}}_{\mathrm{fo}}(b_{[\ell]}, k)$
8: $h'_{[\ell]} \leftarrow \widetilde{\mathsf{Add}}^{\mathrm{pre}}_{\mathrm{fo}}(r^1_{2,[\ell]}, k)$
Precomputed storage: $e'_{[\ell]}, f'_{[\ell]}, g'_{[\ell]}, h'_{[\ell]}$

Online Phase

Input: $e'_{[\ell]}, f'_{[\ell]}, g'_{[\ell]}, h'_{[\ell]}$ and $a_{\ell+1}, b_{\ell+1}$
Output: $c_{\ell+1}$

1: $c_{\ell+1} \leftarrow 0$
2: $e_{[\ell]} \leftarrow \widetilde{\mathsf{Lin}}^{\mathrm{on}}_{\mathrm{fo}}(b_{\ell+1}, e'_{[\ell]}, k)$
3: $f_{[\ell]} \leftarrow \widetilde{\mathsf{Add}}^{\mathrm{on}}_{\mathrm{fo}}(e_{[\ell]}, f'_{[\ell]}, k)$
4: $g_{[\ell]} \leftarrow \widetilde{\mathsf{Lin}}^{\mathrm{on}}_{\mathrm{fo}}(a_{\ell+1}, g'_{[\ell]}, k)$
5: $h_{[\ell]} \leftarrow \widetilde{\mathsf{Add}}^{\mathrm{on}}_{\mathrm{fo}}(g_{[\ell]}, h'_{[\ell]}, k)$
6: $i_{[\ell]} \leftarrow f_{[\ell]} + h_{[\ell]}$
7: $c'_{\ell+1} \leftarrow c'_{\ell+1} + \sum_{j\in[\ell]} i_j$
8: $c_{\ell+1} \leftarrow c'_{\ell+1} + a_{\ell+1} \cdot b_{\ell+1}$

Algorithm 5 Addition Gadget $\mathsf{Add}_{\mathrm{fo}}(a_{[n]}, b_{[n]})$ for Iteration

Input: input sharings $a_{[n]}, b_{[n]}$ where $b_{[n]}$ are unavailable in precomputation phase
Output: output sharing $c_{[n]}$ where $\sum_{i\in[n]}(a_i + b_i) = \sum_{i\in[n]} c_i$

Precomputation Phase

Input: $a_{[n]}$
Output: $c''_{[n]}$ for the online phase

1: $c'_{[n]} \leftarrow \mathsf{Refresh}_{\mathrm{fo}}(a_{[n]})$
2: $c''_{[n]} \leftarrow \mathsf{Refresh}_{\mathrm{fo}}(c'_{[n]})$
Precomputed storage: $c''_{[n]}$

Online Phase

Input: $b_{[n]}, c''_{[n]}$
Output: $c_{[n]}$

1: $c_{[n]} \leftarrow b_{[n]} + c''_{[n]}$

Algorithm 6 Linear Transformation Gadget $\mathsf{Lin}_{\text{fo}}(e, a_{[n]})$ for Iteration

Input: input sharing $a_{[n]}$ and constant e
Output: output sharing $b_{[n]}$ where $\sum_{i\in[n]} b_i = e \cdot \sum_{i\in[n]} a_i$

Precomputation Phase

Input: $a_{[n]}$
Output: $c'_{[n]}$ for the online phase
1: $b'_{[n]} \leftarrow \mathsf{Refresh}_{\text{fo}}(a_{[n]})$
Precomputed storage: $b'_{[n]}$

Online Phase

Input: $b'_{[n]}, e$
Output: $c_{[n]}$
1: $b_{[n]} \leftarrow e \cdot b'_{[n]}$

Algorithm 7 Refresh Gadget $\mathsf{Refresh}_{\text{fo}}$

Input: input sharing $a_{[n]}$
Output: output sharing $b_{[n]}$ such that $\sum_{i\in[n]} b_i = \sum_{i\in[n]} a_i$
1: $b'_{[n]} \leftarrow 0$
2: **for** $i \leftarrow 1$ *to* n **do**
3: **for** $j \leftarrow i+1$ *to* n **do**
4: $r_{i,j} \leftarrow \$$
5: $b'_i \leftarrow b'_i + r_{i,j}$
6: $b'_j \leftarrow b'_j + r_{i,j}$
7: **end for**
8: **end for**
9: $b_{[n]} \leftarrow a_{[n]} + b'_{[n]}$

Algorithm 8 Linear Transformation Gadget Lin_p

Input: input sharing $a_{[\ell]}$ and the public input $a_{\ell+1}$
Output: output sharing $b_{[\ell]}$ and the public output $b_{\ell+1}$ with $\sum_{i\in[\ell+1]} b_i = \mathrm{L}(\sum_{i\in[\ell+1]} a_i)$ where L is the linear transformation operation

Precomputation Phase

Input: $a_{[\ell]}$
Output: $b_{[\ell]}$
1: $b_{[\ell]} \leftarrow \mathsf{Lin}(a_{[\ell]})$
Precomputed storage: Null

Online Phase

Input: $a_{\ell+1}$
Output: $b_{\ell+1}$
1: $b_{\ell+1} \leftarrow \mathrm{L}(a_{\ell+1})$

Corollary 1. *The memory usage of* Mul_p *is* 4ℓ.

Corollary 1 follows directly for $k = 1$ where the memory usage is 12 according to Algorithm 4, namely $a_{[3]}, b_{[3]}$ and $r_{[2],[3]}$. As for the situation of $k > 1$, the memory usage of Mul_p depends on Add_{fo} and Lin_{fo}.

First, we show the memory usage of $\widetilde{\mathsf{Lin}}_{\text{fo}}(k)$. Since step 1 of $\widetilde{\mathsf{Lin}}_{\text{fo}}(k)$ can be operated

Algorithm 9 Lin

Input: input sharing $a_{[n]}$, refresh gadget Refresh
Output: output sharing $c_{[n]}$ of $\mathrm{L}(a_{[n]})$
1: $b_{[n]} \leftarrow \mathrm{L}(a_{[n]})$
2: $c_{[n]} \leftarrow \mathsf{Refresh}(b_{[n]})$

in the precomputation phase, the only memory usage of $\widetilde{\mathsf{Lin}_{\mathrm{fo}}}(k)$ is $b'_{[\ell]}$, which means the memory usage of $\widetilde{\mathsf{Lin}_{\mathrm{fo}}}(k)$ is ℓ.

Then we consider $\mathsf{Add}_{\mathrm{fo}}$. We show how the memory is used in $\widetilde{\mathsf{Add}_{\mathrm{fo}}}(k)$ with $k = 1, 2$ in Figure 6. The operations contained by the red dashed rectangles in Figure 6 can be precomputed because all of the used variables are random, which results in an ℓ-share sharing. Consequently, the memory usage of an ℓ-share $\mathsf{Add}_{\mathrm{fo}}$ is 2ℓ.

So for the ℓ-share Mul_p, there is total 2ℓ memory usage for the $\widetilde{\mathsf{Lin}_{\mathrm{fo}}}(k)$ and 2ℓ memory usage for the $\widetilde{\mathsf{Add}_{\mathrm{fo}}}(k)$. Note that the memory usage of $\widetilde{\mathsf{Add}_{\mathrm{fo}}}(k)$ is not 4ℓ because one of the input sharing of $\widetilde{\mathsf{Add}_{\mathrm{fo}}}(k)$ is the output of $\widetilde{\mathsf{Lin}_{\mathrm{fo}}}(k)$. We illustrate the online phase of Mul_p in Figure 5. According to the proof of Lemma 5, the additions after the calculations of $e_{[\ell]}$ and $f_{[\ell]}$ can be operated with normal addition gates because it is secure for Mul_p even all intermediate variables at step 4 and 6 in the online phase are public. So the memory usage of the whole Mul_p is 4ℓ.

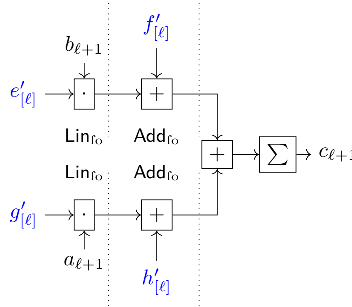


Figure 5: The online phase of Mul_p. The 4ℓ blue variables are stored. There are totally 6ℓ operations in the online phase, which means the online complexity of Mul_p is $\mathcal{O}(n)$ since Mul_p is an $(\ell+1)$-share gadget.

3.4 Online Complexity of the Circuit Compiler

Next, we discuss the online complexity of the precomputable gadgets. Given that the online complexity of $\mathsf{Add}_p, \mathsf{Copy}_p$ and Lin_p is $\mathcal{O}(1)$, we only provide the online complexity of Mul_p in Corollary 2.

Corollary 2. *The online complexity of the* $(\ell+1)$*-share* Mul_p *is* $\mathcal{O}(\ell)$*.*

We provide the online complexity matrix of $\mathsf{Add}_{\mathrm{fo}}$ and $\mathsf{Lin}_{\mathrm{fo}}$ below to prove Corollary 2. The definition of $N_{x,y}$ is described in Section 2.3. In addition,

- a_{fo} denotes addition gates where one input is provided online and the other is stored. These gates are replaced by $\mathsf{Add}_{\mathrm{fo}}$ during the expansion process;
- l_{fo} represents linear transformation gates with one online input and one stored input, which are substituted by $\mathsf{Lin}_{\mathrm{fo}}$ during the expansion.

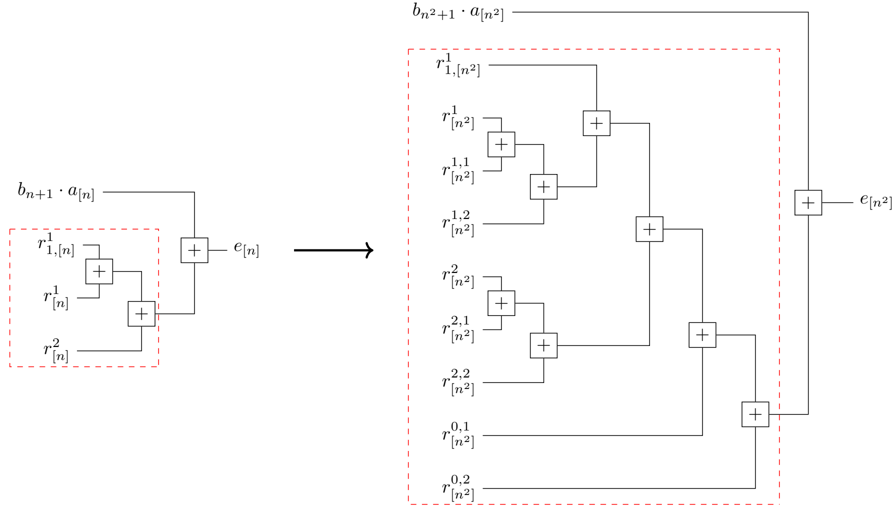


Figure 6: The calculations of $\widetilde{\mathsf{Add}}_{\mathrm{fo}}(k)$ with $k = 1, 2$ and $n = 3$ to get $e_{[n]}$ (or $e_{[n^2]}$) in Submult. Randoms $r^{[2]}_{[n]}$, $r^{[2]}_{[n^2]}$ and $\{r^{i,[2]}_{[n^2]}\}_{i\in[0,2]}$ are generated in $\mathsf{Refresh}_{\mathrm{fo}}$, namely $b'_{[n]}$ or $b'_{[n^2]}$.

$$M = \begin{pmatrix} N_{\mathsf{Add}_{\mathrm{fo}},a_{\mathrm{fo}}} & N_{\mathsf{Lin}_{\mathrm{fo}},a_{\mathrm{fo}}} \\ N_{\mathsf{Add}_{\mathrm{fo}},l_{\mathrm{fo}}} & N_{\mathsf{Lin}_{\mathrm{fo}},l_{\mathrm{fo}}} \end{pmatrix} = \begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix} .$$

Therefore, the online complexity of both $\widetilde{\mathsf{Add}}_{\mathrm{fo}}(k)$ and $\widetilde{\mathsf{Lin}}_{\mathrm{fo}}(k)$ is $\mathcal{O}(3^k) = \mathcal{O}(\ell)$. Moreover, the complexity of calculations from step 6 to step 8 of the online phase of Submult is also $\mathcal{O}(\ell)$, as there are a total of 2ℓ additions and one multiplication. Consequently, the overall online complexity of Submult is $\mathcal{O}(\ell)$.

3.5 Randomness Cost of the Circuit Compiler

First we calculate the randomness cost of $\mathsf{Add}_1, \mathsf{Copy}_1$ and Mul_1. The definition of $N_{x,y}$ are described in Section 2.3.

$$N = \begin{pmatrix} N_{a,a} & N_{c,a} & N_{m,a} & N_{r,a} \\ N_{a,c} & N_{c,c} & N_{m,c} & N_{r,c} \\ N_{a,m} & N_{c,m} & N_{m,m} & N_{r,m} \\ N_{a,r} & N_{c,r} & N_{m,r} & N_{r,r} \end{pmatrix} = \begin{pmatrix} 15 & 12 & 42 & 0 \\ 6 & 9 & 30 & 0 \\ 0 & 0 & 9 & 0 \\ 6 & 6 & 18 & 3 \end{pmatrix} .$$

Therefore, the randomness cost for $k = 1$ is $N^{(1)}_{a,r} = N^{(1)}_{c,r} = 6, N^{(1)}_{m,r} = 18$. And for $k = 2$, there is

$$\begin{cases} N^{(2)}_a = N \cdot (N^{(1)}_a)^\mathsf{T} = N \cdot (15, 6, 0, 6)^\mathsf{T} = (297, 144, 0, 144)^\mathsf{T} \\ N^{(2)}_c = N \cdot (N^{(1)}_c)^\mathsf{T} = N \cdot (12, 9, 0, 6)^\mathsf{T} = (288, 153, 0, 144)^\mathsf{T} \\ N^{(2)}_m = N \cdot (N^{(1)}_m)^\mathsf{T} = N \cdot (42, 30, 9, 18)^\mathsf{T} = (1\,368, 792, 81, 648)^\mathsf{T} \end{cases},$$

i.e., $N^{(2)}_{a,r} = N^{(2)}_{c,r} = 144, N^{(2)}_{m,r} = 648$. For $k = 3$,

$$\begin{cases} N^{(3)}_a = N \cdot (N^{(2)}_a)^\mathsf{T} = N \cdot (297, 144, 0, 144)^\mathsf{T} = (6\,183, 3\,078, 0, 3\,078)^\mathsf{T} \\ N^{(3)}_c = N \cdot (N^{(2)}_c)^\mathsf{T} = N \cdot (288, 153, 0, 144)^\mathsf{T} = (6\,156, 3\,105, 0, 3\,078)^\mathsf{T} \\ N^{(3)}_m = N \cdot (N^{(2)}_m)^\mathsf{T} = N \cdot (1\,368, 792, 81, 648)^\mathsf{T} = (33\,426, 17\,766, 243, 16\,362)^\mathsf{T} \end{cases},$$

i.e., $N^{(3)}_{a,r} = N^{(3)}_{c,r} = 3\,078, N^{(3)}_{m,r} = 16\,362$.

Addition. Now we cope with Add_p, the randomness cost of Add_p is the same as Add_1 since we use Add_1 as the RPC gadget in Add_p, i.e., $6 \times 2 = 12$ Bytes randoms for $k = 1$, $144 \times 2 = 288$ Bytes randoms for $k = 2$ and $3\,078 \times 2 = 6\,156$ Bytes randoms for $k = 3$.

Copy. Copy_p needs $6 \times 2 = 12$ Bytes randoms for $k = 1$, $144 \times 2 = 288$ Bytes randoms for $k = 2$ and $3\,078 \times 2 = 6\,156$ Bytes randoms for $k = 3$.

Multiplication. As for the randomness cost of Mul_p, we have

$$N_{\mathrm{mult}_p} = N_{\mathrm{mult}_1} + 2N_{\mathrm{add}_1} + 4N_{\mathrm{copy}_1} + 2N_{\mathrm{Lin}_{\mathrm{fo}}} + 2N_{\mathrm{add}_{\mathrm{fo}}} \ ,$$

where N_* means the randomness cost of the corresponding gadget. We also have

$$R = \begin{pmatrix} N^{\mathrm{fo}}_{a,a} & N^{\mathrm{fo}}_{a,c} & N^{\mathrm{fo}}_{a,l} & N^{\mathrm{fo}}_{a,r} \\ N^{1}_{c,a} & N^{1}_{c,c} & N^{1}_{c,l} & N^{1}_{c,r} \\ N^{\mathrm{fo}}_{l,a} & N^{\mathrm{fo}}_{l,c} & N^{\mathrm{fo}}_{l,l} & N^{\mathrm{fo}}_{l,r} \\ N_{r,a} & N_{r,c} & N_{r,l} & N_{r,r} \end{pmatrix} = \begin{pmatrix} 15 & 6 & 0 & 6 \\ 12 & 9 & 0 & 6 \\ 6 & 3 & 3 & 3 \\ 0 & 0 & 0 & 3 \end{pmatrix} \ ,$$

where the first and third line of R are the number of addition, copy, linear operations and randomness cost of Add_p and Lin_p respectively, the second line is that of Copy and the last line is that of the random gate. Let $N^{\mathrm{fo}}_* = (N^{\mathrm{fo}}_{*,a}, N^{\mathrm{fo}}_{*,c}, N^{\mathrm{fo}}_{*,l}, N^{\mathrm{fo}}_{*,r})$, and let $N^1_* = (N^1_{*,a}, N^1_{*,c}, N^1_{*,l}, N^1_{*,r})$. There is

$$\begin{cases} N'_{\mathrm{add}_{\mathrm{fo}}} = N^{\mathrm{fo}}_a \cdot N_{\mathrm{fo}} = (297, 144, 0, 144) \\ N''_{\mathrm{add}_{\mathrm{fo}}} = N^{\mathrm{fo}}_a \cdot N^2_{\mathrm{fo}} = (6\,183, 3\,078, 0, 3\,078) \end{cases}$$

and

$$\begin{cases} N'_{\mathrm{lin}_{\mathrm{fo}}} = N^{\mathrm{fo}}_l \cdot N_{\mathrm{fo}} = (144, 72, 9, 72) \\ N''_{\mathrm{lin}_{\mathrm{fo}}} = N^{\mathrm{fo}}_l \cdot N^2_{\mathrm{fo}} = (3\,078, 1\,539, 27, 1\,539) \end{cases} .$$

Hence there are $(18 + 2 \times 6 + 4 \times 6 + 2 \times 6 + 2 \times 3) \times 2 = 144$ Bytes randomness cost for $k = 1$, $(648 + 2 \times 144 + 4 \times 144 + 2 \times 144 + 2 \times 72) \times 2 = 3\,888$ Bytes randomness cost for $k = 2$ and $(16\,362 + 2 \times 3\,078 + 4 \times 3\,078 + 2 \times 3\,078 + 2 \times 1\,539) \times 2 = 88\,128$ Bytes randomness cost for $k = 3$.

Linear transformation. There are 6 Bytes random for $k = 1$, 144 Bytes random for $k = 2$ and 3 078 Bytes random for $k = 3$ since Lin_p uses one $\mathsf{Refresh}$ and Add_p uses two and all of their randoms are used by $\mathsf{Refresh}$.

4 Security of the New Compiler

In this section, we establish the security of the gadgets discussed in Section 3. We demonstrate the security of Add_p and Copy_p through the following lemmas. The proofs of these lemmas are straightforward since all additional wires in Add_p and Copy_p are public compared to Add and Copy.

Lemma 2 (Security of Add_p)**.** *Add_p is (S_k, f)-RPC with input sharings $a_{[\ell]}, b_{[\ell]}$ and output sharing $c_{[\ell]}$ if Add is (S_k, f)-RPC.*

Lemma 3 (Security of Copy_p)**.** *Copy_p is (S_k, f)-RPC with input sharings $a_{[\ell]}$ and output sharing $b_{[\ell]}, c_{[\ell]}$ if Copy is (S_k, f)-RPC.*

According to Proposition 1 and Theorem 1, we derive the following lemma. Notably, the (S_k, f)-RPC of $\mathsf{Submult}$ is established in Lemma 5.

Lemma 4 (Security of Mul_p)**.** *Mul_p is $(S_k, 5 \cdot f')$-RPC with $f' = \max\{f_1, f_2, f_3, f_4\}$, if $\mathsf{Add}, \mathsf{Copy}, \mathsf{Mul}, \mathsf{Submult}$ are (S_k, f_i)-RPC for $i \in [4]$ respectively.*

Lemma 5 (Security of $\mathsf{Submult}$). $\mathsf{Submult}$ *is* $(S_k, 2 \cdot f)$*-RPC with* $f \leqslant 7 \cdot \max\{f_{add}, f_{copy}, f_{add,fo}, f_{lin,fo}\}$*, where* Add, Copy, $\widetilde{\mathsf{Add}}_{fo}(k)$, $\widetilde{\mathsf{Lin}}_{fo}(k)$ *are* (S_k, f_{add})*-RPC,* (S_k, f_{copy})*-RPC,* $(S_k, f_{add,fo})$*-RPC and* $(S_k, f_{lin,fo})$*-RPC respectively.*

Proof. In this proof, we assume that $f_{[\ell]}$ and $h_{[\ell]}$ are public. As a result, the leakage of each $r^1_{1,i}$ (resp. $r^1_{2,i}$) is equivalent to that of e_i (resp. g_i) for $i \in [\ell]$. And thanks to the $(S_k, f_{\text{add,fo}})$-RPE of $\widetilde{\mathsf{Add}}_{\text{fo}}(k)$ at step 3, we have

$$\Pr(I_e \notin S_k) = \Pr\big((I_e \notin S_k) \vee (I_{r^1_1} \notin S_k)\big) = 2 \cdot f_{\text{add,fo}}$$

where $I_e, I_{r^1_1}$ are the output of Sim_1 for $e_{[\ell]}$ and $r^1_{1,[\ell]}$ in the experiment of RPC. Similarly, there is also $\Pr(I_g \notin S_k) = 2 \cdot f_{\text{add,fo}}$ for g_ℓ. We stress that $r^1_{1,[\ell]}$ and $r^1_{2,[\ell]}$ are the outputs of Copy at steps 2 and 3 of the precomputation phase, and their other outputs are the inputs of Add. Thus $I_{r^1_1} \notin S_k$ and $I_{r^1_2} \notin S_k$ only if all gadgets above do not fail. Finally, $I_a \in S_k$ and $I_b \in S_k$ unless $\widetilde{\mathsf{Lin}}_{\text{fo}}(k)$ fail at steps 2 and 4 of the online phase respectively. Therefore we have

$$\begin{aligned}
&\Pr\big((I_a \notin S_k) \vee (I_b \notin S_k)\big) \\
&= 1 - (1 - 2 \cdot f_{\text{add}}) \cdot (1 - 2 \cdot f_{\text{copy}})^2 \cdot (1 - 2 \cdot f_{\text{add,fo}})^2 \cdot (1 - 2 \cdot f_{\text{lin,fo}})^2 \\
&\leqslant 1 - (1 - 2 \cdot \max\{f_{\text{add}}, f_{\text{copy}}, f_{\text{add,fo}}, f_{\text{lin,fo}}\})^7 \\
&\leqslant 2 \cdot 7 \cdot \max\{f_{\text{add}}, f_{\text{copy}}, f_{\text{add,fo}}, f_{\text{lin,fo}}\} \ .
\end{aligned}$$

According to the definition of RPC, let $\Pr\big((I_a \notin S_k) \vee (I_b \notin S_k)\big) = 2 \cdot f$, so that $\mathsf{Submult}$ is $(S_k, 2 \cdot f)$-RPC with $f \leqslant 7 \cdot \max\{f_{\text{add}}, f_{\text{copy}}, f_{\text{add,fo}}, f_{\text{lin,fo}}\}$. □

Instead of providing formal security proofs of Add_{fo} and Lin_{fo}, we use the formal verification tool VRAPS[2] proposed in [BCP+20] to generate their failure probability for (t, f)-RPE directly with $n = 3, t = 1$. For Add_{fo},

$$\begin{aligned}
f_1 &= 4p^2 + 153p^3 + 3\,019p^4 + 3\,9645p^5 + \mathcal{O}(p^6) \\
f_2 &= 12p^2 + 404p^3 + 6\,939p^4 + 31\,806p^5 + \mathcal{O}(p^6) \\
f_{12} &= 2p^3 + 108p^4 + 2\,381p^5 + \mathcal{O}(p^6)
\end{aligned}$$

with the max tolerant leakage probability $\log p_{\max} = -5.01$, where f_1 (resp. f_2) is the failure probability of $\hat{a}$ (resp. $\hat{b}$) and f_{12} is the failure probability of both $\hat{a}$ and $\hat{b}$. For Lin_{fo},

$$f = 12p^2 + 263p^3 + 1140p^4 + 2832p^5 + \mathcal{O}(p^6)$$

with the max tolerant leakage probability $\log p_{\max} = -4.63$. Unfortunately, the amplification of Add_{fo} is $d = \frac{3}{2}$ instead of 2, so the amplification order of the k-time expanded circuit turns to $(\frac{3}{2})^k$. Nevertheless, we have shown that the Add_{fo} is memory-friendly in Section 3.3.

The security of Algorithm 9 is shown in Theorem 2. The definition of TRPE is provided in Appendix A.

Theorem 2 (Security of Lin). *Let* $\mathsf{Refresh}$ *be a* (t, f)*-TRPE* n*-share refresh gadget of amplification order* d*. Then* Lin *instantiated with* $\mathsf{Refresh}$ *is* (t, f')*-RPE of amplification order* d*.*

Proof. Since $\mathsf{Refresh}$ is (t, f)-RPE with amplification order d, we choose the simulator of $\mathsf{Refresh}$ as the simulator of Lin. We assume the leakage wire set of Lin is $\mathcal{W}$ with

[2] The construction and formal verification of VRAPS are also shown in [BCP+20].

$|\mathcal{W}| = d_1 + d_2 < d \leqslant t+1$, and $\mathcal{W}_1$ are the leakage set of $\mathsf{Refresh}$ with $|\mathcal{W}_1| = d_1$. According to Definition 3, any leakage wire in the $\mathsf{Refresh}$ can be simulated by $b_{|I'}$, where I' is generated by the simulator of $\mathsf{Refresh}$ and $|I'| = \min(t, d_1)$. Besides, we put i into I'' if a_i or b_i is leaked, and we have $|I''| \leqslant d_2$. Therefore, we can simulate all leakage wires with $a_{|I}$ where $I = I' \cup I''$ and $|I| < d$, namely $|I| \leqslant t$. So we deduce that the simulator of $\mathsf{Refresh}$ can refer to (t, f')-RPE of Lin with amplification order d. □

Lemma 6 (Security of Lin_p)**.** *Lin_p is (S_k, f)-RPC for input sharing $a_{[\ell]}$ if Lin is (S_k, f)-RPC.*

Finally, we present the security analysis of $\mathsf{Add}_p, \mathsf{Copy}_p, \mathsf{Mul}_p$ and Lin_p, along with their invoking gadgets, in Table 2 with $n = 3$. All invoking gadgets are detailed in Appendix A. Additionally, the tolerant leakage probability of $\mathsf{Add}_p, \mathsf{Copy}_p, \mathsf{Mul}_p$ and Lin_p is chosen as the minimum value among their invoking gadgets: specifically, $p^{(1)}_{\text{add}}, p^{(1)}_{\text{copy}}, p^{(1)}_{\text{copy}}, p^{(1)}_{\text{mult}}$ (i.e. the tolerant leakage probability of the initial circuit compiler) and $\min\{p^{\text{fo}}_{\text{add}}, p^{\text{fo}}_{\text{lin}}\} = 2^{-5.01}$. Thanks to the RPC security of $\mathsf{Add}_p, \mathsf{Copy}_p, \mathsf{Mul}_p$ and Lin_p, we establish the following theorem.

Theorem 3. *Let C be a circuit, let $\mathsf{Add}_p, \mathsf{Copy}_p, \mathsf{Mul}_p$ and Lin_p be the base gadgets of the circuit compiler $(\mathsf{Enc}, \mathsf{CC}, \mathsf{Dec})$, And we define $\mathsf{CC}(\cdot)$ as the operation to replace all gates of the circuit to the base gadgets. Then the compiled circuit $\mathsf{CC}(C)$ is $(p, |C| \cdot f_{\max})$-RPS where $f_{\max}$ is the maximum failure probability of the base gadgets.*

Table 2: Security and tolerant leakage probability of our circuit compiler with $n = 3$, established using $\mathsf{Add}_1, \mathsf{Copy}_1, \mathsf{Mul}_1$ as proposed in [BRT21]. The tolerant leakage probability and amplification order are obtained from their respective papers. The tolerant leakage probability and amplification order of Add_{fo} and Lin_{fo} are discussed in this section.

Additionally, all refresh gadgets without online shares invoke the ISW refresh gadget proposed in [DDF14], proven as (t, f)-TRPE in [BRT21] with amplification order $\min(t+1, n-t)$. According to [BRT21], $\mathsf{Add}_1, \mathsf{Copy}, \mathsf{Mul}_1$ are (t, f)-RPE with amplification order $\min(t+1, n-t)$ with the ISW refresh.

	RPE	RPC	Amplification order	Tolerant leakage probability
Add_1	✓	✓	2	$\log p^{(1)}_{\text{add}} = -4.51$
Copy_1	✓	✓	2	$\log p^{(1)}_{\text{copy}} = -6.10$
Mul_1	✓	✓	2	$\log p^{(1)}_{\text{mult}} = -8.24$
Add_{fo}	✓	✓	$\frac{3}{2}$	$\log p^{\text{fo}}_{\text{add}} = -5.01$
Lin_{fo}	✓	✓	2	$\log p^{\text{fo}}_{\text{lin}} = -4.63$
Add_p	×	✓	2^k	$\log p_{\text{add}} = -4.51$
Copy_p	×	✓	2^k	$\log p_{\text{copy}} = -6.10$
Mul_p	×	✓	$(\frac{3}{2})^k$	$\log p_{\text{mult}} = -8.24$[1]
Lin_p	×	✓	2^k	$\log p_{\text{lin}} = -4.39$

1 The tolerant leakage probability of Mul_p is the least tolerant leakage probability of its component gadgets, i.e., the least one among $p^{(1)}_{\text{add}}, p^{(1)}_{\text{copy}}$ and $p^{(1)}_{\text{mult}}$.

5 Application to Bitsliced AES and Results

5.1 Application and Performance

We describe AES and its bitslicing approach in Appendix C.1 and present its masked implementations in Appendix C.2. Then we consider implementations on the ARM Cortex M architecture, namely ChipWhisperer STM32F415 UFO target board with 192 KB memory. Since there are barely any implementations in the random probing model, we

target some outstanding AES implementations under the probing model instead of the random probing model. Moreover, the performance of our work is comparable to these implementations in execution with a better security. However, we emphasize that our implementation is only a proof-of-concept and the actual performance would require carefully examining the assembly code depending on the specific device targeted. In addition, the round keys are pre-extended and stored in a masked form, which is a common practice in masked implementation and helps improve the performance.

The first benchmark is the implementation of AES proposed in [WGY+22], known as the first work with such precomputation paradigm. We stress that the work in [WGY+22] implements inner product masking and hence can not be applied to the bitsliced implementations. The second benchmark is the state-of-the-art precomputable implementation of AES proposed in [WJZY23], which is the best-known implementation for the probing secure bitsliced AES in precomputation paradigm. The third benchmark is the bitsliced implementation proposed in [GR17], which is the best-known implementation for the bitsliced AES in general. The last benchmark is [BCP+20], the only AES implementation with random probing security to the best of our knowledge. Since there is no directly measured execution cycles, we use the data in [BCP+20] adapted from millisecond to cycle calculated by the CPU frequency in their execution, although the implementation in [BCP+20] is neither precomputable nor bitsliced.

The performance results for $n = 3, 9, 27$ are summarized in Table 3, where timings are given in kilos of clock cycles (Kcycles). It is shown that the efficiency of our work is higher by 9 times to 227 times compared with the other random probing secure implementation proposed in [BCP+20]. Besides, the memory cost of our work is close to that of the precomputable probing secure algorithm proposed in [WGY+22]. In addition, the data that are not mentioned in the literature (e.g., execution cycles with $n = 27$ and code size of some schemes) are marked as '$-$'. It should be noted that, our works are slower than the probing secure ones [WGY+22, WJZY23, GR17], but the random probing security are proven stronger than the probing one.

Notably, the code size of precomputation phase is lower than the online phase in our implementations, since the code of precomputation phase is written by C language while that of online phase is written by assembler one.

Table 3: Comparison of masked implementations.

	Security model	n	Kcycles for precomp.	Random bits	Memory for precomp.	Kcycles for online.	Code size (pre/online)
[WGY+22]	Probing	3	705	96 B	5.63 KB	60	–
[WJZY23]			68	2.22 KB	2.91 KB	50	–
[GR17]			–	3.75 KB	–	83.9	7.5 KB
[BCP+20]	Random probing		–	84.5 KB	–	2 087[1]	–
Our Work			5 977	103 KB	7.5 KB	231	22.7/79.9 KB
[WGY+22]	Probing	9	3 662	1.5 KB	11 KB	137	–
[WJZY23]			446	23.88 KB	11.66 KB	92.27	–
[GR17]			–	45 KB	–	404.5	7.5 KB
[BCP+20]	Random probing		–	2 760 KB	–	22 632	–
Our Work			137 436	2 610 KB	22.5 KB	578	22.7/79.9 KB
[WGY+22]	Probing	27	–	15.8 KB	26.5 KB	–	–
[WJZY23]			–	233.53 KB	37.88 KB	–	–
[GR17]			–	438.75 KB	–	2 783	7.5 KB
[BCP+20]	Random probing		–	67 499 KB	–	387 108	–
Our Work			2 957 229	57 355 KB	67.5 KB	1 704	22.7/79.9 KB

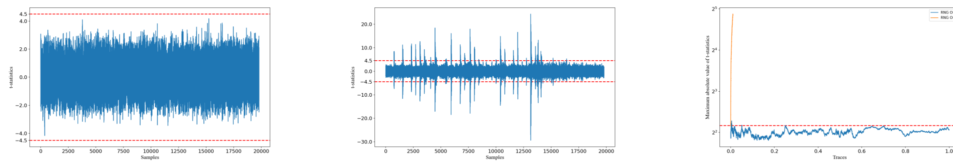
1 The executing cycles of AES in [BCP+20] are scored in milliseconds. To complete the comparison, we calculate the execution cycles by multiplying the execution time and the CPU frequency in [BCP+20].

5.2 Practical Evaluations

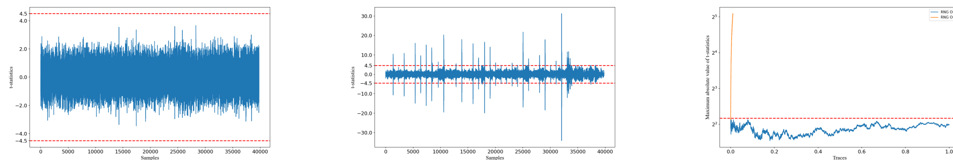
We run the AES round function on a ChipWhisperer STM32F415 UFO target board and collect its power traces with Picoscope 5244D. Besides, we perform a fixed vs. random Welch's T-test with 1 million fixed and random inputs respectively to verify the security order in practice.

Figure 7 provides the first-order T-test results for AES round function. We stress that all tested phases are the online ones since there is no secret in the precomputation phases. Moreover, Figures 7(c) and 7(f) provide the variations of the maximum absolute T-values changing with the traces' number. As the comparison, we also test the situations without randomness in Figures 7(b) and 7(e), where the absolute T-values are quite high with only 10 000 traces. In conclusion, there is no first-order leakage for the compiler of both $n = 3$ and $n = 9$.

Besides, as the two-variant second-order t-test is highly time- and memory-intensive in our case of software implementation, we perform univariate second-order T-test for our implementations, resulting in Figure 8. The versions without randomness are omitted since they have failed in the first-order T-test. Similar to Figures 7(c) and 7(f), Figures 8(b) and 8(d) provide the variations of the maximum absolute T-values in the univariate second-order T-test. In summary, it exhibits no univariate second-order leakage for the compiler of both $n = 3$ and $n = 9$.



(a) Compiler of $n=3$, RNG is on. (b) Compiler of $n=3$, RNG is off. (c) Evolution of the T-values, $n=3$



(d) Compiler of $n=9$, RNG is on. (e) Compiler of $n=9$, RNG is off. (f) Evolution of the T-values, $n=9$.

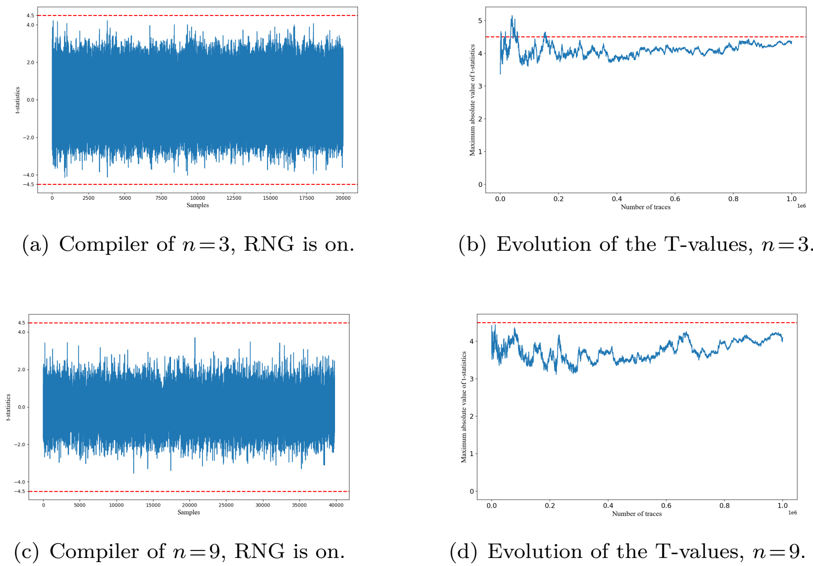
Figure 7: First-order T-test results.

6 Conclusion and Discussions

We propose the first generic precomputable random probing secure circuit compiler with constant leakage probability. Using this compiler, any given (S_k, f)-RPC circuit compiler can be transformed into a precomputable $(S_k, 5 \cdot f)$-RPC one with leakage probability $p \leqslant \min\{p, 2^{-5.01}\}$, where p is the tolerant leakage probability of the initial circuit compiler.

As for the efficiency, each precomputable ℓ-share multiplication gadget incurs a memory cost of 4ℓ, while the memory cost of each other gadgets is 1. Additionally, the online computation complexity of our compiler is $\mathcal{O}(\ell)$. To validate this, we implemented AES-128 using our compiler and compared its memory cost and execution cycles in the online phase with state-of-the-art works on masking AES-128 implementations.

It should be noted that, the usage of the precomputation paradigm faces certain limitations, such as the high complexity of the precomputation phase. This paradigm



(a) Compiler of $n=3$, RNG is on. (b) Evolution of the T-values, $n=3$.

(c) Compiler of $n=9$, RNG is on. (d) Evolution of the T-values, $n=9$.

Figure 8: Univariate second-order T-test results.

represents a trade-off between precomputation and online computation, with applications that benefit from sufficient idle time of the cryptographic device, such as challenge-response protocols, as discussed in the introduction. Additionally, situations where large amounts of data need to be encrypted urgently may render this approach inadequate. To address this issue, leakage-resistant cryptographic modes can be employed (see, e.g., [PSV15, BGP+20] for an incomplete list of literature), which require masking only a few blocks, while the majority of the operations can be executed without side-channel protection.

Acknowledgments

The authors would like to thank the reviewers for their helpful comments and suggestions. This work was supported by the National Natural Science Foundation of China (Grant No. 62372273), the National Key Research and Development Program of China (No. 2021YFA1000600), the Program of Taishan Young Scholars of the Shandong Province, the Program of Qilu Young Scholars (No. 61580082063088) of Shandong University, Department of Science & Technology of Shandong Province (SYS202201) and Quan Cheng Laboratory (QCLZD202306).

References

[AIS18] Prabhanjan Ananth, Yuval Ishai, and Amit Sahai. Private circuits: A modular approach. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 427–455. Springer, 2018.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *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, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.

[BCP+20] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random probing security: Verification, composition, expansion and new constructions. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 339–368. 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.

[BGP+20] Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Tedt, a leakage-resist AEAD mode for high physical security applications. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):256–320, 2020.

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

[BRT21] Sonia Belaïd, Matthieu Rivain, and Abdul Rahman Taleb. On the power of expansion: More efficient constructions in the random probing model. 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 313–343. Springer, 2021.

[BRTV21] Sonia Belaïd, Matthieu Rivain, Abdul Rahman Taleb, and Damien Vergnaud. Dynamic random probing expansion with quasi linear asymptotic complexity. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 157–188. Springer, 2021.

[CFOS21] Gaëtan Cassiers, Sebastian Faust, Maximilian Orlt, and François-Xavier Standaert. Towards tight random probing security. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 185–214. Springer, 2021.

[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.

[CPRR13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2013.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. 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 423–440. Springer, 2014.

[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.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, 2017.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa*

Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. 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 388–397. Springer, 1999.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 96–108. ACM, 2015.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[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.

[WJZY23] Weijia Wang, Fanjie Ji, Juelin Zhang, and Yu Yu. Efficient private circuits with precomputation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):286–309, 2023.

A Gadgets and Notions of the Related Works

Definition 11 (TRPE [BRT21]). Let $f : \mathbb{R} \to \mathbb{R}$. An n-share gadget G : $(\mathbb{K}^n)^2 \to \mathbb{K}^n$ is (t, f)-RPE for some $p \in [0, 1]$, if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $(\hat{x}, \hat{y}) \in (\mathbb{K}^n)^2$ and for every set $J \subseteq [n]$, the random experiment

$$\begin{aligned} &\mathcal{W} \leftarrow \mathsf{LeakingWires}(G, p) \ , \\ &(I_1, I_2, J') \leftarrow \mathsf{Sim}_1^G(\mathcal{W}, J) \ , \\ &out \leftarrow \mathsf{Sim}_2^G(\mathcal{W}, J', \hat{x}_{|I_1}, \hat{y}_{|I_2}) \end{aligned}$$

Algorithm 10 Addition Gadget Add_1 [BRT21]

Input: input sharings $a_{[n]}, b_{[n]}$
Output: output sharing $c_{[n]}$ of $a + b$

1: $e_{[n]} \leftarrow \mathsf{Refresh}(a_{[n]})$
2: $f_{[n]} \leftarrow \mathsf{Refresh}(b_{[n]})$
3: $c_{[n]} \leftarrow e_{[n]} + f_{[n]}$

Algorithm 11 Copy Gadget Copy_1 [BRT21]

Input: input sharing $a_{[n]}$
Output: output sharings $b_{[n]}, c_{[n]}$ fresh copies of $a_{[n]}$

1: $b_{[n]} \leftarrow \mathsf{Refresh}(a_{[n]})$
2: $c_{[n]} \leftarrow \mathsf{Refresh}(a_{[n]})$

Algorithm 12 Multiplication Gadget Mul_1 [BRT21]

Input: input sharings $a_{[n]}, b_{[n]}$, refresh gadget G_{refresh}
Output: output sharing $c_{[n]}$ of $a \cdot b$

1: $(b_{i,[n]}) \leftarrow \mathsf{Refresh}(b_{[n]})$ for $i \in [n]$
2: $r_{[n],[n]} \leftarrow \$$
3: $p_{[n],i} \leftarrow a_{[n]} \cdot b_{[n],i} + r_{[n],i}$ for $i \in [n]$
4: $(v_1, \ldots, v_n), (x_1, \ldots, x_n) \leftarrow (0, \ldots, 0), (0, \ldots, 0)$
5: $v_{[n]} \leftarrow v_{[n]} + p_{n,i}$ for $i \in [n]$
6: $x_{[n]} \leftarrow x_{[n]} + r_{i,[n]}$ for $i \in [n]$
7: $c_{[n]} \leftarrow v_{[n]} + x_{[n]}$

Algorithm 13 ISW Refresh $\mathsf{Refresh}$ [DDF14]

Input: input sharing $a_{[n]}$
Output: output sharing $b_{[n]}$ such that $b_1 + \cdots + b_n = a_1 + \cdots + a_n$

1: $b_{[n]} \leftarrow a_{[n]}$
2: **for** $i \leftarrow 1$ *to* n **do**
3:     **for** $j \leftarrow i + 1$ *to* n **do**
4:         $r_{i,j} \leftarrow \$$
5:         $b_i \leftarrow b_i + r_{i,j}$
6:         $b_j \leftarrow b_j + r_{i,j}$
7:     **end for**
8: **end for**

ensures that

1. the failure events $\mathcal{F}_1 \equiv \big(|I_1| > \min(t, |\mathcal{W}|)\big)$ and $\mathcal{F}_2 \equiv \big(|I_2| > \min(t, |\mathcal{W}|)\big)$ verify

$$\Pr(\mathcal{F}_1) = \Pr(\mathcal{F}_2) = \epsilon \;\; and \;\; \Pr(\mathcal{F}_1 \wedge \mathcal{F}_2) = \epsilon^2$$

   with $\epsilon = f(p)$ (in particular $\mathcal{F}_1$ and $\mathcal{F}_2$ are mutually independent),

2. J' is such that $J' = J$ if $|J| \leqslant t$ and $J' \subseteq [n]$ with $|J'| = n - 1$ otherwise,

3. the output distribution satisfies

$$out \stackrel{id}{=} \Big(\mathsf{AssignWires}\big(G, \mathcal{W}, (\hat{x}, \hat{y})\big), \hat{z}_{|J'}\Big) \ ,$$

   where $\hat{z} = G(\hat{x}, \hat{y})$.

B The Reused Variables in SubBytes

The bitsliced SubBytes in [GR17] is inspired from [BMP13]. In the following we show the operations in [GR17]. It involves 115 logic gates including 32 logic AND. The circuit is composed of 3 parts: the *top linear transformation* involving 23 XOR gates and mapping the 8 S-box input bits $x_p, \ldots, x_7$ to 22 new bits $x_7, y_1, \ldots, y_{21}$; the *middle non-linear transformation* involving 30 XOR gates and 32 AND gates and mapping the previous 23 bits to 18 new bits $z_p, \ldots, z_{17}$; and the *bottom linear transformation* involving 26 XOR gates and 4 XNOR gates and mapping the 18 previous bits to the 8 S-box output bits $s_p, \ldots, s_7$.

– top linear transformation –

$$\begin{array}{l|l|l|l}
y_{14} = x_3 + x_5 & y_1 = t_p + x_7 & y_{15} = t_1 + x_5 & y_{17} = y_{10} + y_{11} \\
y_{13} = x_p + x_6 & y_4 = y_1 + x_3 & y_{20} = t_1 + x_1 & y_{19} = y_{10} + y_8 \\
y_{12} = y_{13} + y_{14} & y_2 = y_1 + x_p & y_6 = y_{15} + x_7 & y_{16} = t_p + y_{11} \\
y_9 = x_p + x_3 & y_5 = y_1 + x_6 & y_{10} = y_{15} + t_p & y_{21} = y_{13} + y_{16} \\
y_8 = x_p + x_5 & t_1 = x_4 + y_{12} & y_{11} = y_{20} + y_9 & y_{18} = x_p + y_{16} \\
t_p = x_1 + x_2 & y_3 = y_5 + y_8 & y_7 = x_7 + y_{11} &
\end{array}$$

– middle non-linear transformation –

$$\begin{array}{l|l|l|l}
t_2 = y_{12} \cdot y_{15} & t_{23} = t_{19} + y_{21} & t_{34} = t_{23} + t_{33} & z_2 = t_{33} \cdot x_7 \\
t_3 = y_3 \cdot y_6 & t_{15} = y_8 \cdot y_{10} & t_{35} = t_{27} + t_{33} & z_3 = t_{43} \cdot y_{16} \\
t_5 = y_4 \cdot x_7 & t_{26} = t_{21} \cdot t_{23} & t_{42} = t_{29} + t_{33} & z_4 = t_{40} \cdot y_1 \\
t_7 = y_{13} \cdot y_{16} & t_{16} = t_{15} + t_{12} & z_{14} = t_{29} \cdot y_2 & z_6 = t_{42} \cdot y_{11} \\
t_8 = y_5 \cdot y_1 & y_{18} = t_6 + t_{16} & t_{36} = t_{24} \cdot t_{35} & z_7 = t_{45} \cdot y_{17} \\
t_{10} = y_2 \cdot y_7 & t_{20} = t_{11} + t_{16} & t_{37} = t_{36} + t_{34} & z_8 = t_{41} \cdot y_{10} \\
t_{12} = y_9 \cdot y_{11} & t_{24} = t_{20} + y_{18} & t_{38} = t_{27} + t_{36} & z_9 = t_{44} \cdot y_{12} \\
t_{13} = y_{14} \cdot y_{17} & t_{30} = t_{23} + t_{24} & t_{39} = t_{29} \cdot t_{38} & z_{10} = t_{37} \cdot y_3 \\
t_4 = t_3 + t_2 & t_{22} = t_{18} + t_{19} & z_5 = t_{29} \cdot y_7 & z_{11} = t_{33} \cdot y_4 \\
t_6 = t_5 + t2 & t_{25} = t_{21} + t_{22} & t_{44} = t_{33} + t_{37} & z_{12} = t_{43} \cdot y_{13} \\
t_9 = t_8 + t_7 & t_{27} = t_{24} + t_{26} & t_{40} = t_{25} + t_{39} & z_{13} = t_{40} \cdot y_5 \\
t_{11} = t_{10} + t_7 & t_{31} = t_{22} + t_{26} & t_{41} = t_{40} + t_{37} & z_{15} = t_{42} \cdot y_9 \\
t_{14} = t_{13} + t_{12} & t_{28} = t_{25} \cdot t_{27} & t_{43} = t_{29} + t_{40} & z_{16} = t_{45} \cdot y_{14} \\
t_{17} = t_4 + t_{14} & t_{32} = t_{31} \cdot t_{30} & t_{45} = t_{42} + t_{41} & z_{17} = t_{41} \cdot y_8 \\
t_{19} = t_9 + t_{14} & t_{29} = t_{28} + t_{22} & z_p = t_{44} \cdot y_{15} & \\
t_{21} = t_{17} + y_{20} & t_{33} = t_{32} + t_{24} & z_1 = t_{37} \cdot y_6 &
\end{array}$$

– bottom linear transformation –

$$\begin{array}{l|l|l|l}
t_{46} = z_{15} + z_{16} & t_{49} = z_9 + z_{10} & t_{61} = z_{14} + t_{57} & t_{48} = z_5 + z_{13} \\
t_{55} = z_{16} + z_{17} & t_{63} = t_{49} + t_{58} & t_{65} = t_{61} + t_{62} & t_{56} = z_{12} + t_{48} \\
t_{52} = z_7 + z_8 & t_{66} = z_1 + t_{63} & s_p = t_{59} + t_{63} & s_3 = t_{53} + t_{66} \\
t_{54} = z_6 + z_7 & t_{62} = t_{52} + t_{58} & t_{51} = z_2 + z_5 & s_1 = \overline{t_{64} + s_3} \\
t_{58} = z_4 + t_{46} & t_{53} = z_p + z_3 & s_4 = t_{51} + t_{66} & s_6 = \overline{t_{56} + t_{62}} \\
t_{59} = z_3 + t_{54} & t_{50} = z_2 + z_{12} & s_5 = t_{47} + t_{65} & s_7 = \overline{t_{48} + t_{60}} \\
t_{64} = z_4 + t_{59} & t_{57} = t_{50} + t_{53} & t_{67} = t_{64} + t_{65} & \\
t_{47} = z_{10} + z_{11} & t_{60} = t_{46} + t_{57} & s_2 = \overline{t_{55} + t_{67}} &
\end{array}$$

According to the circuit, we count the number of occurrences of all intermediate variables which is provided in Tables 4, 5, 6 and 7. There are totally $22 + 29 + 122 + 26 = 229$ used times for all variables in SubBytes instead of $115 \times 2 = 230$, which is bacause there is an s_3 used in the calculation of s_1 which is not counted. Therefore the Copy_p number is $229 - (8 + 21 + 68 + 18) = 114$ for a single SubBytes, and $114 \times 10 = 1\,140$ for the whole AES encryption.

Table 4: Number of occurrences for $x_{[0,7]}$

Variables	x_p	x_1	x_2	x_3	x_4	x_5	x_6	x_7	Total
Times	5	2	1	3	1	3	2	5	22

Table 5: Number of occurrences for $y_{[21]}$

Variables	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
Times	5	2	2	2	3	2	2	4
Variables	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}	y_{16}
Times	3	4	5	3	4	3	4	4
Variables	y_{17}	y_{18}	y_{19}	y_{20}	y_{21}			Total
Times	2	1	1	2	1			59

Table 6: Number of occurrences for $t_{[0,67]}$

Variables	t_p	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
Times	3	2	2	1	1	1	1	2	1	1
Variables	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}
Times	1	1	1	2	2	1	2	1	1	1
Variables	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}	t_{26}	t_{27}	t_{28}	t_{29}
Times	1	2	3	3	4	2	2	3	1	5
Variables	t_{30}	t_{31}	t_{32}	t_{33}	t_{34}	t_{35}	t_{36}	t_{37}	t_{38}	t_{39}
Times	1	1	1	6	2	1	2	4	1	1
Variables	t_{40}	t_{41}	t_{42}	t_{43}	t_{44}	t_{45}	t_{46}	t_{47}	t_{48}	t_{49}
Times	4	3	3	2	2	2	2	1	2	1
Variables	t_{50}	t_{51}	t_{52}	t_{53}	t_{54}	t_{55}	t_{56}	t_{57}	t_{58}	t_{59}
Times	1	1	1	1	1	1	1	2	2	2
Variables	t_{60}	t_{61}	t_{62}	t_{63}	t_{64}	t_{65}	t_{66}	t_{67}		Total
Times	1	1	2	2	2	2	2	2		122

Table 7: Number of occurrences for $z_{[0,17]}$

Variables	z_p	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9
Times	1	1	2	2	2	2	1	2	1	1
Variables	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}	z_{16}	z_{17}		Total
Times	2	1	2	1	1	1	2	1		26

C AES and Its Bitslicing Approach with Masking

C.1 Description of AES and its Bitslicing Approach

The AES-128 block cipher [DR02] 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.

We consider 16 binary operations in parallel, taking the same bitsliced implementation approach as in [GR17]. We use the bitslice at the S-box level that packs the ith bits of 16 S-boxes' inputs, and process 16 S-boxes in parallel. It conveys that we use the 16 bits of one register. We use the compact representation proposed in [BMP13] to implement the AES S-box. Their circuit involves 115 logic gates, including 32 logic AND. Furthermore, the MixColumns and ShiftRows can be evaluated using the strategy given in [GR17], which takes 43 and 144 one-cycle instructions respectively.

C.2 Masked Implementations

After bitslicing the cipher, we adopt the strategy presented in Section 3.1 to obtain the masked implementation for AES. Concretely, we replace the XOR gates with Add_p and the AND gates with Mul_p. Once there is a reused sharing, we add a Copy_p to ensure the difference of all sharings. The precomputation takes the random bits and produces the

precomputed values, and the online phase takes the precomputed values and input shares to calculate the results. We use Add_1, Copy_1, Mul_1, Lin and their expansions to construct Add_p, Mul_p and Lin_p in the implementation.

AddRoundKey. In AddRoundKey, there are $8 \times 11 = 88$ bitwise additions and no other gadgets.

SubBytes. In the S-boxes, it contains $32 \times 10 = 320$ bitwise multiplications and $83 \times 10 = 830$ bitwise additions, and there are $4 \times 10 = 40$ Lin_p used for the 4 XNOR gates in S-boxes. Besides, there are $114 \times 10 = 1\,140$ Copy_p needed in the S-boxes, which is shown in Appendix B.

ShiftRows. In our work, the ShiftRows is implemented as Algorithm 14. It must be applied on the bits of each vector w_k (since each nibble of w_k corresponds to a different row of the state) for $k \in [8]$. And there are totally $3 \times 3 \times 8 \times 10 = 720$ bitwise additions, $3 \times 3 \times 8 \times 10 = 720$ Copy_p and $3 \times 5 \times 8 \times 10 = 1\,200$ Lin_p for the ShiftRows among the AES.

Algorithm 14 Bitsliced ShiftRows [GR17]

Input: 16-bit bitslicing inputs $w_{[8]}$ for ShiftRows
Output: 16-bit bitslicing outputs $z_{[8]}$ for ShiftRows

1: $t \leftarrow 0$
2: **for** $i \leftarrow 1$ *to* 3 **do**
3: $\quad t \leftarrow t + 2^{4-i}$
4: $\quad x_{[8]} \leftarrow w_{[8]} \cdot \big(\mathbf{0x}0\mathrm{F} \ll (4 \times i)\big)$
5: $\quad y_{[8]} \leftarrow x_{[8]} \cdot \big(t \ll (4 \times i)\big)$
6: $\quad z_{[8]} \leftarrow x_{[8]} + y_{[8]}$
7: $\quad z_{[8]} \leftarrow (z_{[8]} \ll i) + \big(y_{[8]} \gg (4 - i)\big)$
8: $\quad v_{[8]} \leftarrow w_{[8]} \cdot \overline{\big(\mathbf{0x}0\mathrm{F} \ll (4 \times i)\big)}$
9: $\quad z_{[8]} \leftarrow z_{[8]} + v_{[8]}$
10: **end for**

MixColumns. According to [GR17], the MixColumns can be described as Algorithm 15, where $\lll$ denotes the rotate left operation on 16 bits. And there are $38 \times 9 = 342$ bitwise additions and Copy_p, and $35 \times 9 = 315$ Lin_p for the bitshift operations. So totally, there

Algorithm 15 Bitsliced MixColumns [GR17]

Input: 16-bit inputs $w_{[8]}$ for bitslicing with order of $(w_8, \dots, w_1)$
Output: 16-bit outputs $z_{[8]}$ for bitslicing with order of $(z_8, \dots, z_1)$

1: $z_1 \leftarrow w_8 + (w_8 \lll 4) + (w_1 \lll 4) + (w_1 \lll 8) + (w_1 \lll 12)$
2: $z_2 \leftarrow w_1 + (w_1 \lll 4) + w_8 + (w_8 \lll 4) + (w_2 \lll 4) + (w_2 \lll 8) + (w_2 \lll 12)$
3: $z_3 \leftarrow w_2 + (w_2 \lll 4) + (w_3 \lll 4) + (w_3 \lll 8) + (w_3 \lll 12)$
4: $z_4 \leftarrow w_3 + (w_3 \lll 4) + w_8 + (w_8 \lll 4) + (w_4 \lll 4) + (w_4 \lll 8) + (w_4 \lll 12)$
5: $z_5 \leftarrow w_4 + (w_4 \lll 4) + w_8 + (w_8 \lll 4) + (w_5 \lll 4) + (w_5 \lll 8) + (w_5 \lll 12)$
6: $z_6 \leftarrow w_5 + (w_5 \lll 4) + (w_6 \lll 4) + (w_6 \lll 8) + (w_6 \lll 12)$
7: $z_7 \leftarrow w_6 + (w_6 \lll 4) + (w_7 \lll 4) + (w_7 \lll 8) + (w_7 \lll 12)$
8: $z_8 \leftarrow w_7 + (w_7 \lll 4) + (w_8 \lll 4) + (w_8 \lll 8) + (w_8 \lll 12)$

are $88 + 830 + 720 + 342 = 1\,980$ Add_p, 320 Mul_p, $1\,140 + 720 + 342 = 2\,202$ Copy_p and $40 + 1\,200 + 315 = 1\,555$ Lin_p for the whole AES-128. Then we calculate the random bits and memory for precomputation of Add_p, Mul_p, Copy_p and Lin_p for $n = 3$ and $k \in [3]$.