IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 273–301. DOI:10.46586/tches.v2024.i3.273-301

Efficient Table-Based Masking with Pre-processing

Juelin Zhang[1,2,3], Taoyun Wang[1,3], Yiteng Sun[1,3], Fanjie Ji[1,3], Bohan Wang[1,3], Lu Li[1,3], Yu Yu[4,5,6] and Weijia Wang[1,2,3](✉)

[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
[4] Shanghai Jiao Tong University, Shanghai, China
[5] Shanghai Qi Zhi Institute, Shanghai, China
[6] Shanghai Key Laboratory of Privacy-Preserving Computation, Shanghai, China.

Abstract. Masking is one of the most investigated countermeasures against side-channel attacks. In a nutshell, it randomly encodes each sensitive variable into a number of shares, and compiles the cryptographic implementation into a masked one that operates over the shares instead of the original sensitive variables. Despite its provable security benefits, masking inevitably introduces additional overhead. Particularly, the software implementation of masking largely slows down the cryptographic implementations and requires a large number of random bits that need to be produced by a true random number generator. In this respect, reducing the overhead of masking is still an essential and challenging task. Among various known schemes, Table-Based Masking (TBM) stands out as a promising line of work enjoying the advantages of generality to any lookup tables. It also allows the pre-processing paradigm, wherein a pre-processing phase is executed independently of the inputs, and a much more efficient online (using the precomputed tables) phase takes place to calculate the result. Obviously, practicality of pre-processing paradigm relies heavily on the efficiency of online phase and the size of precomputed tables.

In this paper, we investigate the TBM scheme that offers a combination of linear complexity (in terms of the security order, denoted as d) during the online phase and small precomputed tables. We then apply our new scheme to the AES-128, and provide an implementation on the ARM Cortex architecture. Particularly, for a security order $d = 8$, the online phase outperforms the current state-of-the-art AES implementations on embedded processors that are vulnerable to the side-channel attacks. The security order of our scheme is proven in theory and verified by the T-test in practice. Moreover, we investigate the speed overhead associated with the random bit generation in our masking technique. Our findings indicate that the speed overhead can be effectively balanced. This is mainly because that the true random number generator operates in parallel with the processor's execution, ensuring a constant supply of fresh random bits for the masked computation at regular intervals.

Keywords: Side-Channel Attack · Table-Based Masking · Pre-processing · AES

1 Introduction

Side-channel attacks have posed an important threat to cryptographic implementations. They exploit the extra information (a.k.a., side-channel leakage) such as timing, power con-

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

sumption and electromagnetic radiation obtained from the cryptographic device. Masking is one of the most investigated countermeasures against side-channel attacks. Its principle is to randomly encode each secret dependent variable into a number of shares such that the distribution of any d shares are independent of the secret input, where d is usually called the security order. The key advantages of masking include provable security and configurability. Informally speaking, the difficulty of any side-channel attacks against a masked cryptographic implementation grows exponentially with the security order d.

Despite the seemingly straightforward principle, designing a scheme that can transform any cryptographic algorithm into an efficient masked implementation is nontrivial. Masking views a cryptographic algorithm as a circuit that is a composition of elemental operations, and transforms each elemental operation into the masked implementation known as a gadget. The transformation of nonlinear operations, such as multiplication or S-boxes used in many ciphers, is the most challenging part. Methods for addressing this challenge can be divided into two main categories. The first one follows the well-known scheme by Ishai et al. [ISW03], who proposed a multiplication gadget with complexity $O(d^2)$. The other approach, called table-based masking, aims to directly provide the masked implementation of lookup tables (e.g., the S-box). The method follows Chari et al.'s work [CJRR99] that is based on the table recomputations. For input shares $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d]$ (In our notation, $\hat{\boldsymbol{x}}$ represents a vector and $\hat{\boldsymbol{x}}[i]$ denotes the i^{th} element for any integer i.), the method starts with the original lookup table (say S), and recomputes the table successively using $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d-1]$, resulting in a masked table $\hat{\mathsf{S}}$, and finally, the output shares can be achieved by $\hat{\mathsf{S}}(\hat{\boldsymbol{x}}[d])$. Both approaches bring large overhead and usually require a large number of random bits, making the employment of masking countermeasures challenging in software, particularly when speed is critical.

Recently, several works [VV21, WGY+22] on the pre-processing of masking have emerged. These works introduce a paradigm in which the computation can be divided into two distinct phases: pre-processing and online phase. The pre-processing is independent of the input variables, and produces some precomputed variables to be temporarily stored in, e.g., the RAM. It can be performed in device's idle time, such as when the IoT device is waiting for the data of authentication or response. The subsequent online phase combines all the input and precomputed variables to calculate the outputs more efficiently. This two-phase approach has been successfully applied in the field of secure multi-party computation, as demonstrated in [BDOZ11, DPSZ12]. Wang et al. [WGY+22] further highlighted the significant performance improvements that can be achieved by leveraging precomputation compatibility in various scenarios, such as challenge-response authentication protocols. They also provided a new multiplication gadget that was compatible with pre-processing. In the recent study conducted by Wang et al. [WJZY23], the research extends the exploration of circuit-based masking with precomputation. It presents an efficient bitsliced implementation suitable for software deployments.

Superficially, TBM is perfectly compatible with the pre-processing paradigm: the masked table can be generated in pre-processing, and the online phase is simply accessing the masked tables. However, adapting TBMs for the pre-processing still poses significant challenges, primarily due to the constraint that each masked table can only be utilized once. That is, for a cryptographic algorithm involving ℓ calls of lookup tables, the pre-processing needs to generate ℓ masked tables. For example, for AES-128 involving 160 calls of S-boxes, the pre-processing needs to generate 160 masked tables, and as a result, the scheme necessitates a relatively large amount of RAM. The above situation caused a demand of small size of the masked table.

The first higher-order TBM was proposed by Schramm and Paar in [SP06] who extend the work of Chari et al. [CJRR99]. In the scheme, each entry of a masked table is a masked variable whose masks are the same across different entries. This scheme was broken in 2007 [CPR07], and, as a result, it is only secure with $d = 1$. Coron then fixed the security

issue and proposed a higher-order secure TBM [Cor14], in which the size of the masked table linearly increases with d. In the scheme, each entry of a masked table is composed of $d+1$ shares, in order to guarantee the probing security. Then, an improved scheme was proposed at CHES 2018 [CRZ18], where computation cost and size of random bits are saved by 50% (while still linear increasing with d).

1.1 Our Contributions

In this paper, we continue investigating the TBM, and our contributions are threefold.

A new construction. We provide a pre-processing compatible TBM offering two key features:

1. A small number of precomputed variables: This allows for wider deployment of the scheme on devices with limited RAM capacity.
2. Linear complexity (pertaining to the security order d) of the online phase: This results in a highly efficient scheme in terms of speed.

In a nutshell, our scheme is to encode all entries of the (unmasked) lookup table and d random variables by a Maximum Distance Separable (MDS) encoder, and the codeword comprises the masked table, which is smaller than that in Coron's work. Thanks to the MDS encoder, the probing security is guaranteed, ensuring that any d elements in the codeword are distributed independently of the unmasked entries. For a lookup table with m-bit input and m-bit output, we summarize the performance of our scheme and other known designs of TBM in Table 1, where $\tilde{m} \stackrel{\text{def}}{=} \lceil \log_2(2^m + d) \rceil$ is very close to m. Our scheme is to encode all entries of the (unmasked) lookup table and d random variables by an MDS encoder, and the codeword is the masked table such that any d elements in the codeword are distributed independently of the unmasked entries. Our scheme is tailored for deployment on devices with limited RAM, achieved through reduced precomputed bits complexity. However, the two phases of preprocessing and online processing are slightly inferior to [Cor14, CRZ18]. But our random bits' complexity is better than theirs.

A discussion on the overhead of the random bits generation. A masked implementation usually requires the generation of random bits, which is often regarded as a significant timing overhead in software. However, we found that this consideration may not always be true in practice, particularly in the case of microprocessors equipped with a True Random Number Generator (TRNG). This is because, in such systems, the generation of random bits and the execution of the program can occur in parallel. Concretely, when the masked implementation requires random bits at regular intervals, the timing overhead can be significantly minimized by employing a buffering mechanism. By continuously generating random bits and storing them in a buffer, the masked implementation can efficiently retrieve the required random bits from the buffer, thereby reducing the impact on overall execution time.

Application, implementation, and practical verification. We apply the masking to protect the AES-128. We instantiate our masking to the lookup table $\mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$, and implement the masked AES on the ARM Cortex architecture. For a security order of $d = 8$, the online phase of our implementation runs in 47.81 Kcycles. The pre-processing takes 622.77 Mcycles to produce 43.76 Kbytes of precomputed variables. The result also confirms the discussion on the overhead of random number generation. At last, we collect the power traces of the masked AES, and perform the T-test to verify the security order in practice.

1.2 Related Works

There also exist some TBM schemes that aim at reducing the size of the masked table. At CHES 2021, Valiveti et al. proposed a new scheme with an almost constant size

Table 1: Asymptotic complexities of various TBMs regarding any m-bit to m-bit lookup table.

	Pre-processing	Precomputed bits	Online phase	Random bits
[Cor14, CRZ18]*	$O(2^m md^2)$	$O(2^m md)$	$O(md)$	$O(2^m md^2)$
[VV21]	$O(2^m m^3 d^3)$	$O(2^m m + m^2 d^2)$	$O(m^3 d^2)$	$O(m^2 d^2)$
Ours	$O(2^m \tilde{m}^2 d^3)$	$O(2^m \tilde{m} + \tilde{m} d)$	$O(\tilde{m}^2 d)$	$O(\tilde{m} d^3)$

* Although the latter approach demonstrates improved performance in practical applications, it is important to note that the asymptotic complexities of the schemes proposed in [Cor14] and [CRZ18] remain the same.

(independent of d) of the masked table [VV21]. They applied the technique of masking with pseudorandom random number generators (PRGs). However, the online computation of this approach is quadratic in d, due to the running of PRGs. Alexander et al. recently proposed a very efficient TBM scheme that also supports the pre-processing functionality [AVV23].

1.3 Organization

In the rest of the paper, we first present notations and backgrounds in Section 2, and present the technical overview of our new scheme in Section 3. We formally describe the new masking scheme and provide security proof in Section 4. Section 5 gives the discussion regarding the random bit generation. Section 6 presents the applications to AES, gives the performance results, and validates the security in practice using T-test evaluations. Finally, Section 7 concludes the paper.

2 Preliminaries

2.1 Notations

In the rest of this paper, we use $\mathbb{F}_q$ to denote a finite field with $q = p^m$, where p is a prime and m is a positive integer. We denote elements in $\mathbb{F}_q$ by lowercase letters. Let calligraphies (e.g., $\mathcal{I}$) be sets, and $|\mathcal{I}|$ denote the cardinality of the set $\mathcal{I}$. Let bold lower cases (e.g., $\boldsymbol{x}$) be the vectors over $\mathbb{F}_q^{|\boldsymbol{x}|}$, where $|\boldsymbol{x}|$ denotes the length of the vector, $\boldsymbol{x}[i]$ denotes the i^{th} element of the vector $\boldsymbol{x}$, and $\boldsymbol{x}[i\!:\!j]$ denotes the vector made up of i^{th} to j^{th} elements of vector $\boldsymbol{x}$. Unless otherwise noted, we assume the vectors are row vectors, and the column vectors are denoted as $\boldsymbol{x}^{\mathrm{T}}$. Let bold capital letters (e.g., $\boldsymbol{A}$) be the matrices in $\mathbb{F}_q^{r\times c}$ (or $r \times c$ matrix), for row and column counts being r and c respectively. Let $\boldsymbol{A}[i, *]$ (resp., $\boldsymbol{A}[*, j]$) be the i^{th} row (resp., j^{th} column) of $\boldsymbol{A}$.

We use $\oplus$ and $\ominus$ to denote addition and subtraction over $\mathbb{F}_q$, and use $\cdot$ to denote the multiplication. Particularly, for $x, y \in \mathbb{F}_q$ we usually abbreviate $x \cdot y$ as xy. Moreover, these operations $\oplus$ and $\ominus$ extend naturally to element-wise addition and subtraction for vectors and matrices. We use $\odot$ to denote element-wise multiplication for vectors or matrices. When performing multiplication between a matrix (say, $\boldsymbol{A}$) and a column vector (say, $\boldsymbol{a}^{\mathrm{T}}$), we denote it as $\boldsymbol{A} \cdot \boldsymbol{a}^{\mathrm{T}}$, which can also be expressed concisely as $\boldsymbol{A}\boldsymbol{a}^{\mathrm{T}}$. We use $\sum \boldsymbol{x}$ to denote the summation of all the elements of vector $\boldsymbol{x}$. As the symbol encompasses all the elements, we omit writing explicit bounds for the sake of brevity. We also have $\sum_{k=i}^{j} \boldsymbol{x}[k] = \sum \boldsymbol{x}[i\!:\!j] = \boldsymbol{x}[i] \oplus \ldots \oplus \boldsymbol{x}[j]$ for any integers $i < j$. Similarly, we use $\sum \boldsymbol{A}$ to denote the summation of all the rows of matrix $\boldsymbol{A}$, resulting in a vector.

We define the integer representation of a field element $e = \alpha_{m-1}x^{m-1} + \alpha_{m-2}x^{m-2} + \ldots + \alpha_0$ as the integer $\alpha_{m-1}p^{m-1} + \alpha_{m-2}p^{m-2} + \ldots + \alpha_0$. For a field $\mathbb{F}_q$, we also define the field representation of an integer $i = \alpha_{m-1}p^{m-1} + \alpha_{m-2}p^{m-2} + \ldots + \alpha_0$ as the field element

$e = \alpha_{m-1}x^{m-1} + \alpha_{m-2}x^{m-2} + \ldots + \alpha_0$. For a vector $\boldsymbol{v}$ (resp., a matrix $\boldsymbol{M}$) and any field element $e \in \mathbb{F}_q$, we define $\boldsymbol{v}[e] \stackrel{\text{def}}{=} \boldsymbol{v}[i]$ with i the integer representation of e. This definition can be generalized for the index of the matrix. For example, for an element $e \in \mathbb{F}_q$ and a matrix $\boldsymbol{M}$, $\boldsymbol{M}[e, *] \stackrel{\text{def}}{=} \boldsymbol{M}[i, *]$ with i the integer representation of e. Moreover, for an integer i and a field element e, we define $i \oplus e \stackrel{\text{def}}{=} e' \oplus e$ with e' the field representation of i.

For two fields $\mathbb{F}_{q_1}$ and $\mathbb{F}_{q_2}$ sharing the same characteristic p with $q_1 = p^{m_1}$, $q_2 = p^{m_2}$ and $m_1 \geq m_2$, a function $\underset{\bar{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}} : \mathbb{F}_{p^{m_1}} \to \mathbb{F}_{p^{m_2}}$ is defined as follows: for any $e = \alpha_{m_1-1}p^{m_1-1} + \alpha_{m_1-2}p^{m_1-2} + \ldots + \alpha_0$, $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e) = \alpha_{m_2-1}p^{m_2-1} + \alpha_{m_2-2}p^{m_2-2} + \ldots + \alpha_0$. We can see that $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}$ is an additive-homomorphic function: for any $e, e' \in \mathbb{F}_{q_1}$, $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e \oplus e') = \underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e) \oplus \underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e')$ and $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e \ominus e') = \underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e) \ominus \underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(e')$. For example, for $e \in \mathbb{F}_{2^9}$, $\underset{2^9\to2^8}{\mathsf{fieldMap}}(e)$ returns the element in $\mathbb{F}_{2^8}$ by simply removing the highest significant bit of e. Additionally, $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}$ can be generalized for the case of a vector: $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{FieldMap}} : \mathbb{F}^{\ell}_{p^{m_1}} \to \mathbb{F}^{\ell}_{p^{m_2}}$ as follows: for any $\boldsymbol{e} \in \mathbb{F}^{\ell}_{q_1}$, $\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{FieldMap}}(\boldsymbol{e}) = \Big(\underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(\boldsymbol{e}[0]), \ldots, \underset{\mathsf{q}_1\to\mathsf{q}_2}{\mathsf{fieldMap}}(\boldsymbol{e}[\ell-1])\Big)$.

2.2 Maximum Distance Separable (MDS) Matrix.

Our construction uses the MDS matrix with some diffusion properties. We provide its definition along with a crucial property as follows.

Definition 1 (MDS matrix)**.** A matrix $\boldsymbol{A}$ is MDS, if and only if all the sub-determinants of $\boldsymbol{A}$ are non-zero.

Lemma 1. *Consider an $\ell \times d$ MDS matrix denoted as $\boldsymbol{A}$ and an $d \times d'$ matrix represented by $\boldsymbol{R}$. Specifically, let the first n rows of $\boldsymbol{R}$ be uniformly distributed. We then have any n rows of $\boldsymbol{AR}$ are uniformly distributed.*

Proof. Without loss of generality, we assume that $\boldsymbol{R}[\mathcal{I}, *]$ is uniformly distributed with $|\mathcal{I}| = n$ and $\mathcal{I} \subseteq \{0, \ldots, d-1\}$. Let $\bar{\mathcal{I}} \stackrel{\text{def}}{=} \{0, \ldots, d-1\} \setminus \mathcal{I}$. Then, we can express $\boldsymbol{AR} = \boldsymbol{A}[*, \mathcal{I}]\boldsymbol{R}[\mathcal{I}, *] \oplus \boldsymbol{A}[*, \bar{\mathcal{I}}]\boldsymbol{R}[\bar{\mathcal{I}}, *]$. As $\boldsymbol{A}$ is an MDS matrix, for any $\mathcal{I}' \subseteq \{0, \ldots, d-1\}$ with $|\mathcal{I}'| = n$, $\boldsymbol{A}[\mathcal{I}', \mathcal{I}]$ is nonsingular. This property ensures that both $\boldsymbol{A}[\mathcal{I}', \mathcal{I}]\boldsymbol{R}[\mathcal{I}, *]$ and $\boldsymbol{A}[\mathcal{I}', *]\boldsymbol{R}$ are uniformly distributed. Therefore, any n rows of $\boldsymbol{AR}$ are uniformly distributed. □

An MDS matrix can be constructed using the Vandermonde matrix built from $a_0, \ldots, a_{n-1} \in \mathbb{F}_q, \ldots, \mathbb{F}_q$ as follows:

$$\boldsymbol{A}' = \mathsf{van}_d(a_0, \ldots, a_n) = \begin{pmatrix} 1 & a_0 & a_0^2 & \ldots & a_0^{d-1} \\ 1 & a_1 & a_1^2 & \ldots & a_1^{d-1} \\ 1 & a_2 & a_2^2 & \ldots & a_2^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{n-1} & a_{n-1}^2 & \ldots & a_{n-1}^{d-1} \end{pmatrix} \text{ with } n > d \ .$$

$\boldsymbol{A}'$ is the generating matrix of the linear code that achieves the singleton bound, if $q \geq n$. Then, the $(n-d) \times d$ matrix $\boldsymbol{A} = \boldsymbol{A}'[d : n-1, *]\boldsymbol{A}'[0 : d-1, *]^{-1}$ is MDS. We can thus construct an $\ell \times d$ MDS matrix over $\mathbb{F}_q$ if $q \geq d + \ell$. On the other hand, this is also known as the MDS conjecture [Seg55] postulating that there exists no $d \times \ell$ MDS matrix over $\mathbb{F}_q$ if $q < d + \ell$.

2.3 The Concept of Masking

A cryptographic algorithm manipulates sensitive internal variables through a sequence of operations, which can be represented as a directed acyclic graph where sensitive variables and operations are vertices and edges respectively, and usually called a circuit. The general goal of the masking scheme is to randomly split every sensitive variable of a cryptographic algorithm into $d+1$ shares such that the joint distribution of any d variables in the masked implementation is independent of all sensitive variables. We call the security of such a masked implementation as d-private security (a.k.a., d-probing security) [ISW03], as defined in Definition 2. To achieve this, a masking scheme first randomly splits input variables into $d+1$ shares, then complies every operation into its masked correspondence named gadget. Usually, we group the $d+1$ shares corresponding to one sensitive variable as a sharing. *Then, the input and output of a gadget are sharings corresponding to the input and output of the operation.*

Definition 2 (d-probing security [ISW03])**.** A gadget is d-probing security if any t internal probes with $t \leq d$ can be simulated without any secrets.

In this paper, we consider the additive sharing. For a variable x, the additive sharing $\hat{\boldsymbol{x}}$ of $x \in \mathbb{F}_q$ is a set of $d+1$ shares $\{\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d]\} \in \mathbb{F}_q^{d+1}$, such that $x = \sum \hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}[0] \oplus \ldots \oplus \hat{\boldsymbol{x}}[d]$.

2.4 Composable Security Notions

Although the definition of private security nicely includes side-channel attacks, it is not trivial to directly prove large circuits (such as the AES block cipher) to be private secure. The difficulty arises from the need to enumerate the probes within the circuit, a task that becomes increasingly intricate as the circuit size grows. The natural solution is to use the composition method. By doing so, the focus can be directed towards each individual gadget, and the overall d-private security is then ensured through composition. It is Barthe et al. that were the first to introduce composable security notions [BBD+16] for (small) gadgets that are sufficient to result in provable probing security. We first describe the definition of simulatability introduced in [BBP+16]:

Definition 3 (Simulatability [BBP+16])**.** Let $\mathcal{P}$ be a set of probes of a circuit C with input shares $\mathcal{X}$. Let $\mathcal{S} \subseteq \mathcal{X}$ be a subset of input shares. A simulator is a randomized function Sim: $\mathbb{F}_q^{|\mathcal{S}|} \to \mathbb{F}_q^{|\mathcal{P}|}$. 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 $\mathsf{C}_{\mathcal{P}}(\mathcal{X})$ and $\mathsf{Sim}(\mathcal{S})$ are identical, where $\mathsf{C}_{\mathcal{P}}(\mathcal{X})$ returns the values of probes in $\mathcal{P}$ by feeding the input $\mathcal{X}$.

Based on the definition of simulatability, we can recall the composable security notions called Non-Inference (NI) and Strong Non-Inference (SNI) as follows. Here, the internal probes to a circuit refer to the probes directed at variables other than the outputs, and the output probes are defined as the probes directed at the outputs.

Definition 4 (NI/SNI [BBD+16])**.** A gadget is NI (resp., SNI), if any t_{int} internal probes and t_{out} output probes with $t_{int} + t_{out} \leq d$ can be simulated with $t_{int} + t_{out}$ (resp., t_{int}) shares of each input sharing.

As the goal of the composable security notions is to establish probing security, Lemma 2 serves as a bridge between SNI/NI and probing security.

Lemma 2 (SNI $\Rightarrow$ NI $\Rightarrow$ d-probing security [BBP+16])**.** *An SNI gadget is NI, and an NI gadget is d-probing secure if any d input shares needed for the simulation are independently distributed of the secrets.*

The following lemma illustrates the composability of NI and SNI gadgets.

Lemma 3 (Composability of NI and SNI gadgets [BBP+16]). *A composition of gadgets is NI if all gadgets are NI or SNI, based on the following composition rule: each sharing is used at most once as input of a non-SNI gadget. Moreover, a composition of gadgets is SNI if it is NI and the output sharings are from SNI gadgets.*

2.5 Linear Gadget

We briefly introduce the linear gadget that is designed to implement the additive-homomorphic function in the masked domain. An additive-homomorphic function is a function $\mathsf{L} : \mathbb{F}_{q_1}^{\ell} \to \mathbb{F}_{q_2}^{\ell'}$ taking ℓ input and ℓ' output variables. It possesses the property that for any $(x_1, \ldots, x_\ell) \in \mathbb{F}_{q_1}^{\ell}$ and $(y_1, \ldots, y_\ell) \in \mathbb{F}_{q_2}^{\ell}$, it holds that $\mathsf{L}(x_1 \oplus y_1, \ldots, x_\ell \oplus y_\ell) = \mathsf{L}(x_1, \ldots, x_\ell) \oplus \mathsf{L}(y_1, \ldots, y_\ell)$. In the masked domain, the input and output variables are encoded into *sharings* $(\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_\ell)$, and the output should likewise be *sharings* $(\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_{\ell'})$ such that $(\sum \hat{\boldsymbol{y}}_1, \ldots, \sum \hat{\boldsymbol{y}}_{\ell'}) = \mathsf{L}(\sum \hat{\boldsymbol{x}}_1, \ldots, \sum \hat{\boldsymbol{x}}_\ell)$. Particularly, we have

$$\mathsf{L}(\sum \hat{\boldsymbol{x}}_1, \ldots, \sum \hat{\boldsymbol{x}}_\ell) = \sum_{i=0}^{d} \big(\mathsf{L}(\hat{\boldsymbol{x}}_1[i], \ldots, \hat{\boldsymbol{x}}_\ell[i])\big).$$

It conveys that the masked linear operation is evaluated by $(\hat{\boldsymbol{y}}_1[i], \ldots, \hat{\boldsymbol{y}}_{\ell'}[i]) \leftarrow \mathsf{L}(\hat{\boldsymbol{x}}_1[i], \ldots, \hat{\boldsymbol{x}}_\ell[i])$ for $i \in \{0 \ldots d\}$. That is, linear operations can be executed separately over the shares with distinct indices, leading to the following conclusions:

1. It is NI.
2. Each output share corresponding to index i is exclusively determined by the input shares bearing the same index, allowing pre-processing.

We present the linear gadget in Gadget 1, which is separated into two phases: pre-processing and online. The pre-processing (resp., online phase) manipulates the shares with indices in $\{0 \ldots d-1\}$ (resp., index d).

Gadget 1 $\mathsf{MaskedLin}_{\mathsf{L}}$

Input: sharings $\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_\ell$
Output: sharings $\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_{\ell'}$ such that $(\sum \hat{\boldsymbol{y}}_1, \ldots, \sum \hat{\boldsymbol{y}}_{\ell'}) = \mathsf{L}(\sum \hat{\boldsymbol{x}}_1, \ldots, \sum \hat{\boldsymbol{x}}_\ell)$.

Pre-processing

1: $(\hat{\boldsymbol{y}}_1[i], \ldots, \hat{\boldsymbol{y}}_{\ell'}[i]) \leftarrow \mathsf{L}(\hat{\boldsymbol{x}}_1[i], \ldots, \hat{\boldsymbol{x}}_\ell[i])$, for $i \in \{0, \ldots, d-1\}$

Online-computation

2: $(\hat{\boldsymbol{y}}_1[d], \ldots, \hat{\boldsymbol{y}}_{\ell'}[d]) \leftarrow \mathsf{L}(\hat{\boldsymbol{x}}_1[d], \ldots, \hat{\boldsymbol{x}}_\ell[d])$

2.6 On the Precomputation Paradigm of Masking

We describe the conditions under which the pre-processing can be adopted for masking. The pre-processing is a computationally intensive stage of the masked implementation, outputting some precomputed variables/tables. Then, the online phase is performed efficiently using the precomputed variables/tables. To achieve this optimization, we consider the precomputation paradigm introduced by Wang et al. [WGY+22] in which for each internal sharing (say, $\hat{\boldsymbol{x}}$), shares $\hat{\boldsymbol{x}}[0{:}d-1]$ are calculated in pre-processing, while only $\hat{\boldsymbol{x}}[d]$ is calculated in the online phase. To illustrate this concept more concretely:

- In pre-processing, the gadgets are evaluated sequentially in a specific order. For each gadget, the output shares with indices in $[0{:}d-1]$ are calculated. Besides, the precomputed variables/tables required for online phase are also calculated.

- In online phase, the gadgets are also evaluated in a sequential order. For each gadget, the output shares with index d are calculated by precomputed variables/tables and input shares with index d.

To implement the aforementioned design, any gadget with input sharing $\hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_n$ and output sharing $\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_{n'}$ should satisfy one of the following two conditions.

1. *Condition 1*: $\hat{\boldsymbol{y}}_1[0{:}d{-}1], \ldots, \hat{\boldsymbol{y}}_{n'}[0{:}d{-}1]$ should be calculated by $\hat{\boldsymbol{x}}_1[0{:}d{-}1], \ldots, \hat{\boldsymbol{x}}_n[0{:}d{-}1]$ and randomness within the gadget.
2. *Condition 2*: $\hat{\boldsymbol{y}}_1[0{:}d{-}1], \ldots, \hat{\boldsymbol{y}}_{n'}[0{:}d{-}1]$ should be calculated only by the randomness within the gadget.

We can see that Condition 2 is stronger than Condition 1, i.e., a gadget satisfying Condition 2 should also satisfy Condition 1, but not vice versa. The linear gadget presented in Section 2.5 satisfies Condition 1. Our new gadget for lookup table (will be formally introduced in Section 4) satisfies Condition 2.In a masked implementation, the input gadgets should satisfy Condition 2, and the other gadgets should satisfy Condition 1.

Gadget 2 presents a refreshing gadget satisfying Condition 2. This gadget originated from the work in [IKL+13]. Coron et al. [CGZ20] proved that this gadget is secure in the security notion called Probing-Isolating Non-Inference (PINI) [CS20]. Besides, Cassiers et al. have proven that PINI is strictly stronger than NI [CS20], i.e., a gadget is NI if it is PINI. Therefore, this gadget is also NI.

Gadget 2 Refresh satisfying Condition 2

Input: sharing $\hat{\boldsymbol{x}}$.
Output: sharing $\hat{\boldsymbol{y}}$ such that $\sum \hat{\boldsymbol{x}} = \sum \hat{\boldsymbol{y}}$

——————Pre-processing——————

1: Generate random elements $r_0, \ldots, r_{d-1}$
2: $\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d{-}1] \leftarrow r_0, \ldots, r_{d-1}$

——————Online-computation——————

3: $\hat{\boldsymbol{y}}[d] \leftarrow \hat{\boldsymbol{x}}[d] \oplus (\hat{\boldsymbol{x}}[0] \ominus r_0) \oplus (\hat{\boldsymbol{x}}[1] \ominus r_1) \oplus \ldots \oplus (\hat{\boldsymbol{x}}[d-1] \ominus r_{d-1})$

3 Technical Overview of the New Scheme

In this section, we first revisit two typical TBMs (Schramm and Paar's scheme and Coron's scheme) that inspire our scheme, showing their design concepts and limitations. Then, a technical overview of our scheme is illustrated. Our description in this section specifically focuses on the m-bit to m_o-bit lookup table $\mathsf{S}(\cdot)$ with d-order security. For the sake of simplicity, we will set $m = 2$ and $d = 2$ to facilitate comprehension. In this context, we have input shares $\hat{\boldsymbol{x}}[0], \hat{\boldsymbol{x}}[1], \hat{\boldsymbol{x}}[2]$ over $\mathbb{F}_{2^m}$, and the output shares are $\hat{\boldsymbol{y}}[0], \hat{\boldsymbol{y}}[1], \hat{\boldsymbol{y}}[2]$ over $\mathbb{F}_{2^{m_o}}$, such that $\mathsf{S}\big(\sum \hat{\boldsymbol{x}}\big) = \sum \hat{\boldsymbol{y}}$.

It is important to note that the descriptions provided can be easily extended to accommodate any values of m and d. For a more detailed understanding of the two typical TBMs, readers can refer to Appendices A and B, respectively. Moreover, Section 4 will formally present our scheme for any lookup table. Our descriptions use the static single-assignment (SSA) form, where each variable is assigned exactly once, simplifying the properties of variables.

3.1 Schramm and Paar's scheme

The origins of TBM can be traced back to the work of Chari et al. [CJRR99] that provided a scheme with two shares, and was extended by Schramm and Paar in [SP06] to the

case with multiple shares. We briefly recall Schramm and Paar's scheme in Figure 1. It first generates d random values $\boldsymbol{s}[0], \ldots, \boldsymbol{s}[d-1]$ that serve as d output shares (i.e., $\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d-1]$), and constructs a vector $\boldsymbol{t}_0$ of size 2^m (i.e., the number of possible inputs of $\mathsf{S}(\cdot)$) such that $\boldsymbol{t}_0[i] \oplus \sum \boldsymbol{s} = \mathsf{S}(i)$ for any $i \in \{0, \ldots, 2^m-1\}$. Then, the elements in vector $\boldsymbol{t}_0$ are shifted for d rounds. In the k^{th} round, the elements are shifted by $\hat{\boldsymbol{x}}[k]$, i.e., for any $i \in \{0 \ldots 2^m-1\}$, $\boldsymbol{t}_k[i] \leftarrow \boldsymbol{t}_{k-1}[i \oplus \hat{\boldsymbol{x}}[k]]$, ensuring $\mathsf{S}\big(i \oplus \sum \hat{\boldsymbol{x}}[0\!:\!k]\big) = \boldsymbol{t}_k[i] \oplus \sum \boldsymbol{s}$. Finally, the last share of the output should be $\hat{\boldsymbol{y}}[d] = \boldsymbol{t}_d[\hat{\boldsymbol{x}}[d]]$.

Although this method circumvents the direct summing of input shares, Coron et al. demonstrated that it remains insecure with $d > 1$ [CPR07]. For instance, consider the scenario in Figure 1, assuming that $\hat{\boldsymbol{x}}[2] \neq 0$, it can be readily observed that the joint leakage of $\hat{\boldsymbol{y}}[2]$ and $\boldsymbol{t}_0[0]$ depends on all input shares $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d]$, since $\hat{\boldsymbol{y}}[2] \oplus \boldsymbol{t}_0[0] \oplus \mathsf{S}(0) = \mathsf{S}\big(\sum \hat{\boldsymbol{x}}\big)$. Moreover, for the case of $d > 2$ (as demonstrated in Appendix A), the joint leakage of $\boldsymbol{t}_d[0]$, $\boldsymbol{t}_d[1]$ and $\hat{\boldsymbol{x}}[d]$ depends on all input shares, since $\boldsymbol{t}_d[0] \oplus \boldsymbol{t}_d[1] = \mathsf{S}\big(\sum \hat{\boldsymbol{x}}[0\!:\!d\text{–}1]\big) \oplus \mathsf{S}\big(1 \oplus \sum \hat{\boldsymbol{x}}[0\!:\!d\text{–}1]\big)$. Those security flaws stem from the fact that the procedure employs the same masks $\boldsymbol{s}[0], \ldots, \boldsymbol{s}[d-1]$.

$$
\begin{aligned}
&\boldsymbol{s}[0], \boldsymbol{s}[1], \begin{matrix}\boldsymbol{t}_0[0]\\ \boldsymbol{t}_0[1]\\ \boldsymbol{t}_0[2]\\ \boldsymbol{t}_0[3]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_0[0]\\ \mathsf{S}(1) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_0[1]\\ \mathsf{S}(2) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_0[2]\\ \mathsf{S}(3) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_0[3]\end{matrix}\\
&\hat{\boldsymbol{x}}[0] \longrightarrow \left|\begin{matrix}\boldsymbol{t}_1[0] \leftarrow \boldsymbol{t}_0[0 \oplus \hat{\boldsymbol{x}}[0]]\\ \boldsymbol{t}_1[1] \leftarrow \boldsymbol{t}_0[1 \oplus \hat{\boldsymbol{x}}[0]]\\ \boldsymbol{t}_1[2] \leftarrow \boldsymbol{t}_0[2 \oplus \hat{\boldsymbol{x}}[0]]\\ \boldsymbol{t}_1[3] \leftarrow \boldsymbol{t}_0[3 \oplus \hat{\boldsymbol{x}}[0]]\end{matrix}\right.\\
&\boldsymbol{s}[0], \boldsymbol{s}[1], \begin{matrix}\boldsymbol{t}_1[0]\\ \boldsymbol{t}_1[1]\\ \boldsymbol{t}_1[2]\\ \boldsymbol{t}_1[3]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_1[0]\\ \mathsf{S}(1 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_1[1]\\ \mathsf{S}(2 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_1[2]\\ \mathsf{S}(3 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{s}[0] \oplus \boldsymbol{s}[1] \oplus \boldsymbol{t}_1[3]\end{matrix}\\
&\hat{\boldsymbol{x}}[1] \longrightarrow \left|\begin{matrix}\boldsymbol{t}_2[0] \leftarrow \boldsymbol{t}_1[0 \oplus \hat{\boldsymbol{x}}[1]]\\ \boldsymbol{t}_2[1] \leftarrow \boldsymbol{t}_1[1 \oplus \hat{\boldsymbol{x}}[1]]\\ \boldsymbol{t}_2[2] \leftarrow \boldsymbol{t}_1[2 \oplus \hat{\boldsymbol{x}}[1]]\\ \boldsymbol{t}_2[3] \leftarrow \boldsymbol{t}_1[3 \oplus \hat{\boldsymbol{x}}[1]]\end{matrix}\right.\\
&\boldsymbol{s}[0], \boldsymbol{s}[1], \begin{matrix}\boldsymbol{t}_2[0]\\ \boldsymbol{t}_2[1]\\ \boldsymbol{t}_2[2]\\ \boldsymbol{t}_2[3]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{t}_2[0] \oplus \boldsymbol{s}[0] \oplus \boldsymbol{s}[1]\\ \mathsf{S}(1 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{t}_2[1] \oplus \boldsymbol{s}[0] \oplus \boldsymbol{s}[1]\\ \mathsf{S}(2 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{t}_2[2] \oplus \boldsymbol{s}[0] \oplus \boldsymbol{s}[1]\\ \mathsf{S}(3 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{t}_2[3] \oplus \boldsymbol{s}[0] \oplus \boldsymbol{s}[1]\end{matrix}\\
&\hat{\boldsymbol{x}}[2] \longrightarrow \left|\begin{matrix}\hat{\boldsymbol{y}}[0], \hat{\boldsymbol{y}}[1], \hat{\boldsymbol{y}}[2] \leftarrow \boldsymbol{s}[0], \boldsymbol{s}[1], \boldsymbol{t}_2[\hat{\boldsymbol{x}}[2]],\end{matrix}\right.\\
&\hat{\boldsymbol{y}}[0], \hat{\boldsymbol{y}}[1], \hat{\boldsymbol{y}}[2], \text{ such that } \mathsf{S}(\hat{\boldsymbol{x}}[2] \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \hat{\boldsymbol{y}}[0] \oplus \hat{\boldsymbol{y}}[1] \oplus \hat{\boldsymbol{y}}[2]
\end{aligned}
$$

Figure 1: Schramm and Paar's scheme [SP06] with $m = 2$ and $d = 2$.

3.2 Coron's Scheme

Coron addressed the above security flaws by replacing each element in $\boldsymbol{t}$ with $d+1$ shares and refreshing the shares after each shifting operation [Cor14]. As depicted in Figure 2, Coron's approach replaces the vector $\boldsymbol{t}$ by a $2^m \times (d+1)$ matrix $\boldsymbol{T}$, such that the i^{th} row $\boldsymbol{T}[i,*]$ consists of the shares of $\mathsf{S}(i \oplus \hat{\boldsymbol{x}}[0] \oplus \ldots \oplus \hat{\boldsymbol{x}}[k])$. The refreshing first generates $d+1$ random variables (say, $r_0, \ldots, r_d$) such that $r_0 \oplus \ldots \oplus r_d = 0$, and element-wisely adds them to the input vector. The refreshing enhances security as the same masks are no longer shared among different rows of the matrix.

The refreshing runs in $O(d)$ and requires $O(d)$ random variables. In total, Coron's method runs in $O(2^m d^2 m_o)$ and $O(dm_o)$ for pre-processing and online phase respectively, and demands $O(2^m d^2)$ random variables. Although this adaption results in a probing secure scheme, it imposes a significant requirement of the RAM to the precomputation paradigm. The precomputed table $\boldsymbol{T}$ contains $2^m(d+1)$ entries, which is $(d+1)$ times larger than the vector $\boldsymbol{t}$ used in Schramm and Paar's method. Consequently, the pre-processing produces d times more variables (to be stored in the RAM). For instance, in the case of AES-128, which includes $10 \times 16 = 160$ 8-bit to 8-bit Sboxes, Coron's scheme requires approximately $40(d+1)$ KBytes of memory to store all the precomputed tables. This renders the pre-processing phase impractical in many real-world applications.

$$\begin{matrix}\boldsymbol{T}_0[0,0] & \boldsymbol{T}_0[0,1] & \boldsymbol{T}_0[0,2] \\ \boldsymbol{T}_0[1,0] & \boldsymbol{T}_0[1,1] & \boldsymbol{T}_0[1,2] \\ \boldsymbol{T}_0[2,0] & \boldsymbol{T}_0[2,1] & \boldsymbol{T}_0[2,2] \\ \boldsymbol{T}_0[3,0] & \boldsymbol{T}_0[3,1] & \boldsymbol{T}_0[3,2]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0) = \boldsymbol{T}_0[0,0] \oplus \boldsymbol{T}_0[0,1] \oplus \boldsymbol{T}_0[0,2] \\ \mathsf{S}(1) = \boldsymbol{T}_0[1,0] \oplus \boldsymbol{T}_0[1,1] \oplus \boldsymbol{T}_0[1,2] \\ \mathsf{S}(2) = \boldsymbol{T}_0[2,0] \oplus \boldsymbol{T}_0[2,1] \oplus \boldsymbol{T}_0[2,2] \\ \mathsf{S}(3) = \boldsymbol{T}_0[3,0] \oplus \boldsymbol{T}_0[3,1] \oplus \boldsymbol{T}_0[3,2]\end{matrix}$$

$$\hat{\boldsymbol{x}}[0] \longrightarrow \begin{matrix}(\boldsymbol{T}_1[0,0], \boldsymbol{T}_1[0,1], \boldsymbol{T}_1[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_0[0 \oplus \hat{\boldsymbol{x}}[0], 0], \boldsymbol{T}_0[0 \oplus \hat{\boldsymbol{x}}[0], 1], \boldsymbol{T}_0[0 \oplus \hat{\boldsymbol{x}}[0], 2]) \\ (\boldsymbol{T}_1[1,0], \boldsymbol{T}_1[1,1], \boldsymbol{T}_1[1,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_0[1 \oplus \hat{\boldsymbol{x}}[0], 0], \boldsymbol{T}_0[1 \oplus \hat{\boldsymbol{x}}[0], 1], \boldsymbol{T}_0[1 \oplus \hat{\boldsymbol{x}}[0], 2]) \\ (\boldsymbol{T}_1[0,0], \boldsymbol{T}_1[0,1], \boldsymbol{T}_1[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_0[2 \oplus \hat{\boldsymbol{x}}[0], 0], \boldsymbol{T}_0[2 \oplus \hat{\boldsymbol{x}}[0], 1], \boldsymbol{T}_0[2 \oplus \hat{\boldsymbol{x}}[0], 2]) \\ (\boldsymbol{T}_1[0,0], \boldsymbol{T}_1[0,1], \boldsymbol{T}_1[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_0[3 \oplus \hat{\boldsymbol{x}}[0], 0], \boldsymbol{T}_0[3 \oplus \hat{\boldsymbol{x}}[0], 1], \boldsymbol{T}_0[3 \oplus \hat{\boldsymbol{x}}[0], 2])\end{matrix}$$

$$\begin{matrix}\boldsymbol{T}_1[0,0] & \boldsymbol{T}_1[0,1] & \boldsymbol{T}_1[0,2] \\ \boldsymbol{T}_1[1,0] & \boldsymbol{T}_1[1,1] & \boldsymbol{T}_1[1,2] \\ \boldsymbol{T}_1[2,0] & \boldsymbol{T}_1[2,1] & \boldsymbol{T}_1[2,2] \\ \boldsymbol{T}_1[3,0] & \boldsymbol{T}_1[3,1] & \boldsymbol{T}_1[3,2]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{T}_1[0,0] \oplus \boldsymbol{T}_1[0,1] \oplus \boldsymbol{T}_1[0,2] \\ \mathsf{S}(1 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{T}_1[1,0] \oplus \boldsymbol{T}_1[1,1] \oplus \boldsymbol{T}_1[1,2] \\ \mathsf{S}(2 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{T}_1[2,0] \oplus \boldsymbol{T}_1[2,1] \oplus \boldsymbol{T}_1[2,2] \\ \mathsf{S}(3 \oplus \hat{\boldsymbol{x}}[0]) = \boldsymbol{T}_1[3,0] \oplus \boldsymbol{T}_1[3,1] \oplus \boldsymbol{T}_1[3,2]\end{matrix}$$

$$\hat{\boldsymbol{x}}[1] \longrightarrow \begin{matrix}(\boldsymbol{T}_2[0,0], \boldsymbol{T}_2[0,1], \boldsymbol{T}_2[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_1[0 \oplus \hat{\boldsymbol{x}}[1], 0], \boldsymbol{T}_1[0 \oplus \hat{\boldsymbol{x}}[1], 1], \boldsymbol{T}_1[0 \oplus \hat{\boldsymbol{x}}[1], 2]) \\ (\boldsymbol{T}_2[1,0], \boldsymbol{T}_2[1,1], \boldsymbol{T}_2[1,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_1[1 \oplus \hat{\boldsymbol{x}}[1], 0], \boldsymbol{T}_1[1 \oplus \hat{\boldsymbol{x}}[1], 1], \boldsymbol{T}_1[1 \oplus \hat{\boldsymbol{x}}[1], 2]) \\ (\boldsymbol{T}_2[0,0], \boldsymbol{T}_2[0,1], \boldsymbol{T}_2[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_1[2 \oplus \hat{\boldsymbol{x}}[1], 0], \boldsymbol{T}_1[2 \oplus \hat{\boldsymbol{x}}[1], 1], \boldsymbol{T}_1[2 \oplus \hat{\boldsymbol{x}}[1], 2]) \\ (\boldsymbol{T}_2[0,0], \boldsymbol{T}_2[0,1], \boldsymbol{T}_2[0,2]) \leftarrow \mathsf{Refresh}(\boldsymbol{T}_1[3 \oplus \hat{\boldsymbol{x}}[1], 0], \boldsymbol{T}_1[3 \oplus \hat{\boldsymbol{x}}[1], 1], \boldsymbol{T}_1[3 \oplus \hat{\boldsymbol{x}}[1], 2])\end{matrix}$$

$$\begin{matrix}\boldsymbol{T}_2[0,0] & \boldsymbol{T}_2[0,1] & \boldsymbol{T}_2[0,2] \\ \boldsymbol{T}_2[1,0] & \boldsymbol{T}_2[1,1] & \boldsymbol{T}_2[1,2] \\ \boldsymbol{T}_2[2,0] & \boldsymbol{T}_2[2,1] & \boldsymbol{T}_2[2,2] \\ \boldsymbol{T}_2[3,0] & \boldsymbol{T}_2[3,1] & \boldsymbol{T}_2[3,2]\end{matrix}, \text{ such that } \begin{matrix}\mathsf{S}(0 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{T}_2[0,0] \oplus \boldsymbol{T}_2[0,1] \oplus \boldsymbol{T}_2[0,2] \\ \mathsf{S}(1 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{T}_2[1,0] \oplus \boldsymbol{T}_2[1,1] \oplus \boldsymbol{T}_2[1,2] \\ \mathsf{S}(2 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{T}_2[2,0] \oplus \boldsymbol{T}_2[2,1] \oplus \boldsymbol{T}_2[2,2] \\ \mathsf{S}(3 \oplus \hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1]) = \boldsymbol{T}_2[3,0] \oplus \boldsymbol{T}_2[3,1] \oplus \boldsymbol{T}_2[3,2]\end{matrix}$$

$$\hat{\boldsymbol{x}}[2] \longrightarrow \hat{\boldsymbol{y}}[0], \hat{\boldsymbol{y}}[1], \hat{\boldsymbol{y}}[2] \leftarrow \boldsymbol{T}_2[\hat{\boldsymbol{x}}[2], 0], \boldsymbol{T}_2[\hat{\boldsymbol{x}}[2], 1], \boldsymbol{T}_2[\hat{\boldsymbol{x}}[2], 2],$$

$$\hat{\boldsymbol{y}}[0], \hat{\boldsymbol{y}}[1], \hat{\boldsymbol{y}}[2], \text{ such that } \mathsf{S}(\hat{\boldsymbol{x}}[0] \oplus \hat{\boldsymbol{x}}[1] \oplus \hat{\boldsymbol{x}}[2]) = \hat{\boldsymbol{y}}[0] \oplus \hat{\boldsymbol{y}}[1] \oplus \hat{\boldsymbol{y}}[2]$$

Figure 2: Coron's scheme [Cor14] with $d = 2$ and $m = 2$.

3.3 A Birdeye on Our Scheme

Our scheme is briefly described in Figure 3. It begins with vectors $\boldsymbol{t}_0$ of size 2^m and $\boldsymbol{s}_0$ of size d, designed to satisfy the relationship $\big(\mathsf{S}(0), \ldots, \mathsf{S}(2^m-1)\big) = \boldsymbol{t}_0 \oplus \boldsymbol{A}\boldsymbol{s}_0^{\mathrm{T}}$, with the

constant matrix $\boldsymbol{A} = \begin{bmatrix} 2 & 4 & 6 & 2 \\ 3 & 5 & 7 & 4 \end{bmatrix}^{\mathrm{T}}$ for the case of $m = 2$ and $d = 2$. The dimensions of $\boldsymbol{A}$ vary depending on the values of m and d. We omit the precise description with respect to the field elements in $\boldsymbol{A}$ for the sake of brevity, and deter the formal description to the next section.

In the k^{th} iteration, $(\boldsymbol{s}_k, \boldsymbol{t}_k)$ undergoes a shift by $\hat{\boldsymbol{x}}[k]$ and is refreshed. This operation ensures that

$$\big(\mathsf{S}(0 \oplus \sum \hat{\boldsymbol{x}}[0{:}k{-}1]), (1 \oplus \sum \hat{\boldsymbol{x}}[0{:}k{-}1]), \ldots, \mathsf{S}(2^m{-}1 \oplus \sum \hat{\boldsymbol{x}}[0{:}k{-}1])\big) = \boldsymbol{t}_k \oplus \boldsymbol{A}\boldsymbol{s}_k^{\mathrm{T}} \ .$$

After d iterations, we obtain

$$\big(\mathsf{S}(0 \oplus \sum \hat{\boldsymbol{x}}[0{:}d{-}1]), \mathsf{S}(1 \oplus \sum \hat{\boldsymbol{x}}[0{:}d{-}1]), \ldots, \mathsf{S}(2^m{-}1 \oplus \sum \hat{\boldsymbol{x}}[0{:}d{-}1])\big) = \boldsymbol{t}_d \oplus \boldsymbol{A}\boldsymbol{s}_d^{\mathrm{T}} \ .$$

It conveys that we can calculate the output shares from $(\boldsymbol{s}, \boldsymbol{t}[\hat{\boldsymbol{x}}[d]])$ (via the ToShares operation). The precomputed table $(\boldsymbol{s}, \boldsymbol{t})$ contains $2^m + d$ variables, significantly smaller than the $\boldsymbol{T}$ in Coron's method. As a result, this new scheme is much more practical for pre-processing compared to previous TBM designs. It should be noted that (and will be proved in the next section), our scheme is secure if the matrix $\boldsymbol{A}$ is MDS.

We will provide a detailed formal description in the next section. In the following, we introduce the additional considerations and details in the formal presentation.

1. For any $k \in \{0, \ldots, d\}$, the matrix $\boldsymbol{A}$ and vector $\boldsymbol{s}_k$ should be over the same field (say, $\mathbb{F}_{2^{\tilde{m}}}$). By the MDS conjecture [Seg55], a $d \times 2^m$ MDS matrix over $\mathbb{F}_{2^{\tilde{m}}}$ must satisfy the condition $d + 2^m \leq 2^{\tilde{m}}$. It conveys that there does not exist a $d \times 2^m$ MDS matrix over $\mathbb{F}_{2^{\tilde{m}}}$ if $\tilde{m} \leq m$. Thus, the scheme requires $\tilde{m} > m$. In the case of Figure 3, the bit-length of the elements in $\boldsymbol{A}$ and $\boldsymbol{s}_k$ should surpass 2. Considering an m-bit to m_o-bit lookup table S, the vector $\boldsymbol{t}_k$ should be over $\mathbb{F}_{2^{m_o}}$ for any $k \in \{0, \ldots, d\}$. This is not a big issue if $m_o > m$ since we can simply set $\tilde{m} = m_o$. For the case that $m_o \leq m$, we additionally need an function $\underset{2^{\tilde{\mathrm{m}}} \to 2^{\mathrm{m_o}}}{\mathsf{FieldMap}}$ defined in Section 2.1 to map the elements in $\mathbb{F}_{2^{\tilde{m}}}$ to $\mathbb{F}_{2^{m_o}}$, such that for any $k \in \{0, \ldots, d\}$,
$$\big(\mathsf{S}(0 \oplus \sum \hat{\boldsymbol{x}}[0{:}k{-}1]), \ldots, \mathsf{S}(2^m{-}1 \oplus \sum \hat{\boldsymbol{x}}[0{:}k{-}1])\big) = \boldsymbol{t}_k \oplus \underset{2^{\tilde{\mathrm{m}}} \to 2^{\mathrm{m_o}}}{\mathsf{FieldMap}}(\boldsymbol{A}\boldsymbol{s}_k^{\mathrm{T}}) \ .$$
2. Furthermore, we extend our consideration to encompass a more general lookup table, mapping each element in the field $\mathbb{F}_q$ to an element in $\mathbb{F}_{q_o}$, where $q = p^m$ and $q_o = p_o^{m_o}$ with p, p_o any primes (rather than only 2).

4 New Table-based Masking Scheme

In this section, we formally present and analyze our new TBM that is compatible with the precomputation-based paradigm.

4.1 Construction

We consider the lookup table $\mathsf{S}(\cdot)$ mapping each element in the field $\mathbb{F}_q$ to an element in $\mathbb{F}_{q_o}$ such that $q = p^m$, $q_o = p_o^{m_o}$ and p, p_o are primes. The new gadget MaskedTable is equipped with the lookup table $\mathsf{S}(\cdot)$, a new field $\mathbb{F}_{\tilde{q}}$ such that $\tilde{q} = p_o^{\tilde{m}}$, $\tilde{m} \geq m_o$ and $\tilde{q} \geq q + d$ [1], and an MDS matrix $\boldsymbol{A} \in \mathbb{F}_{\tilde{q}}^{q \times d}$. It takes a sharing $\hat{\boldsymbol{x}}$, and produces the output sharing $\hat{\boldsymbol{y}}$ such that $\sum \hat{\boldsymbol{y}} = \mathsf{S}(\sum \hat{\boldsymbol{x}})$.

[1] The conditions can also be presented as $\tilde{q} \geq q_o$, $\tilde{q} \geq q + d$ and the field $\mathbb{F}_{\tilde{q}}$ shares the same characteristic as $\mathbb{F}_{q_o}$.

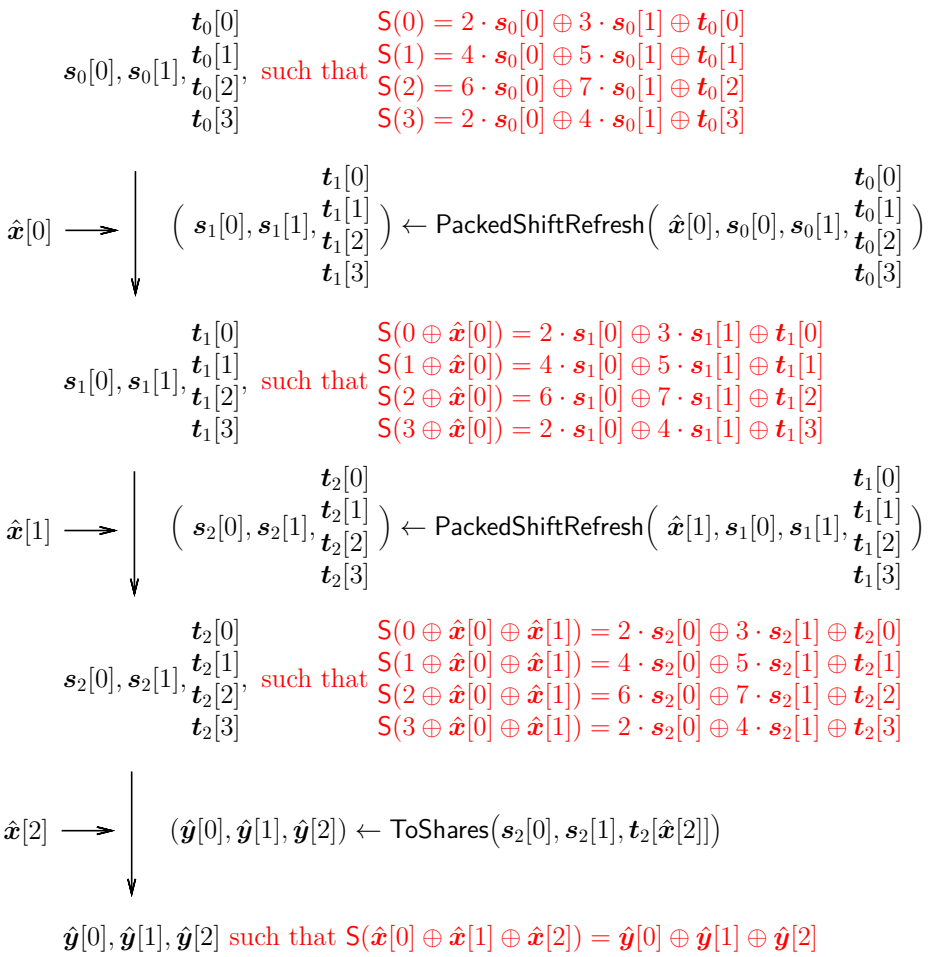


Figure 3: Our scheme with $d = 2$ and $m = 2$

MaskedTable first constructs a tuple $(\boldsymbol{s}_0, \boldsymbol{t}_0) \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$. Note that, the function $\underset{\tilde{q}\to q_o}{\mathsf{FieldMap}} : \mathbb{F}_{\tilde{q}} \to \mathbb{F}_{q_o}$ defined in Section 2.1 is required to map the elements in $\mathbb{F}_{\tilde{q}}$ to elements in $\mathbb{F}_{q_o}$ while keeping the homomorphism over addition. The tuple $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ satisfies that $\mathsf{S}(e) = \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[e,*]\boldsymbol{s}_0^{\mathrm{T}}\right) \oplus \boldsymbol{t}_0[e]$ for any $e \in \mathbb{F}_q$, i.e., $\mathsf{S}(e)$ is determined by $\boldsymbol{s}_0$, $\boldsymbol{t}_0[e]$ and the constant vector $\boldsymbol{A}[e,*]$. The MaskedTable refreshes and shifts (by elements in $\hat{\boldsymbol{x}}$) the tuple by calling PackedShiftRefresh. The input/output of the refreshing is written in single static assignment (SSA) form, where each input/output tuple is defined exactly once. After d refreshings, the tuple $(\boldsymbol{s}_d, \boldsymbol{t}_d[\hat{\boldsymbol{x}}[d]])$ is transformed into the output sharing. The transformation circuit, called ToShares, is separated into two parts: the pre-processing phase and the online phase.

Gadget 3 MaskedTable

Input: sharing $\hat{\boldsymbol{x}} \in \mathbb{F}_q^{d+1}$.

Output: sharing $\hat{\boldsymbol{y}} \in \mathbb{F}_{q_o}^{d+1}$

The gadget is equipped with a lookup table $\mathsf{S} : \mathbb{F}_q \to \mathbb{F}_{q_o}$ and an MDS matrix $\boldsymbol{A} \in \mathbb{F}_{\tilde{q}}^{q\times d}$, where $\tilde{q} \geq q_o$, $\tilde{q} \geq q + d$ and the field $\mathbb{F}_{\tilde{q}}$ shares the same characteristic as $\mathbb{F}_{q_o}$.
The gadget ensures that: $\mathsf{S}(\sum \hat{\boldsymbol{x}}) = \sum \hat{\boldsymbol{y}}$.

——————Pre-processing——————

1: $\boldsymbol{s}_0 = (0, \ldots, 0) \in \mathbb{F}_{\tilde{q}}^d$
2: $\boldsymbol{t}_0 \in \mathbb{F}_{q_o}^q$ is a vector such that $\mathsf{S}(e) = \boldsymbol{t}_0[e]$ for any $e \in \mathbb{F}_q$
3: **for** $k = 0$; $k \leq d-1$; $k{+}{+}$ **do**
4: $\quad(\boldsymbol{s}_{k+1}, \boldsymbol{t}_{k+1}) \leftarrow \mathsf{PackedShiftRefresh}((\boldsymbol{s}_k, \boldsymbol{t}_k), \hat{\boldsymbol{x}}[k])$
5: **end for**
6: $(\hat{\boldsymbol{y}}[0{:}d{-}1], \boldsymbol{w}) \leftarrow \mathsf{ToShares{-}part1}()$
▷ Precomputed variables: $\boldsymbol{w}$, $(\boldsymbol{s}_d, \boldsymbol{t}_d)$ and $\hat{\boldsymbol{y}}[0{:}d{-}1]$

——————Online-computation——————

7: $\hat{\boldsymbol{y}}[d] \leftarrow \mathsf{ToShares{-}part2}\big(\boldsymbol{w}, (\boldsymbol{s}_d, \boldsymbol{t}_d), \hat{\boldsymbol{x}}[d]\big)$

PackedShiftRefresh takes $(\boldsymbol{s}, \boldsymbol{t}) \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$ and $x \in \mathbb{F}_q$, and returns $(\boldsymbol{s}', \boldsymbol{t}') \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$. The process begins by generating a random matrix $\boldsymbol{R}$, and calculating the summation of its columns, resulting in $\boldsymbol{s}'$. Then, it calculates $\boldsymbol{W} \leftarrow \boldsymbol{A}\boldsymbol{R}$, where a linear operation is performed on each column of $\boldsymbol{R}$. We can see that $\sum \boldsymbol{W}[i,*] = \boldsymbol{A}[i,*]\boldsymbol{s}'^{\mathrm{T}}$. After that, we calculate $\boldsymbol{t}'[e]$ by $\boldsymbol{A}[e \oplus x, *], \boldsymbol{s}, \boldsymbol{W}[e,*]$ and $\boldsymbol{t}[e \oplus x]$ for each $e \in \mathbb{F}_q$, such that

$$\boldsymbol{t}'[e] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[e,*]\boldsymbol{s}'^{\mathrm{T}}\right) = \boldsymbol{t}[e \oplus x] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[e \oplus x, *]\boldsymbol{s}^{\mathrm{T}}\right).$$

ToShares takes $(\boldsymbol{s}, \boldsymbol{t}) \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$ and $x \in \mathbb{F}_q$, and returns $(\boldsymbol{s}', \boldsymbol{t}') \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$. The process begins by generating a random matrix $\boldsymbol{R}$, and calculating the summation of the columns, resulting in the first d shares of $\hat{\boldsymbol{y}}$. The summation of the rows of $\boldsymbol{R}$ produces a vector $\boldsymbol{w}$. After that, we can calculate $\hat{\boldsymbol{y}}[d]$ by $\boldsymbol{A}[x,*], \boldsymbol{s}, \boldsymbol{W}[e,*]$ and $\boldsymbol{w}$ such that $\underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\right) \oplus \boldsymbol{t}[x] = \sum \hat{\boldsymbol{y}}$.

Let $\tilde{m} \stackrel{\text{def}}{=} \log_2 \tilde{q}$, the complexities of the multiplication and addition over $\mathbb{F}_{\tilde{q}}$ are $O(\tilde{m}^2)$ and $O(\tilde{m})$, respectively. Thus, the complexity of PackedShiftRefresh is $O(\tilde{m}^2 q d^2)$, and the complexities of ToShares for pre-processing and online phases are $O(\tilde{m} d^2)$ and $O(\tilde{m}^2 d)$, respectively. As MaskedTable consists of d PackedShiftRefresh and one ToShares, its complexities for pre-processing and online phases are $O(\tilde{m}^2 q d^3)$ and $O(\tilde{m}^2 q d)$, respectively.

4.2 Correctness

First of all, we provide the correctness of PackedShiftRefresh and ToShares.

Circuit 1 PackedShiftRefresh

Input: A share $x \in \mathbb{F}_q$ and $(\boldsymbol{s}, \boldsymbol{t}) \in \left(\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q\right)$.

Output: $(\boldsymbol{s}', \boldsymbol{t}') \in \left(\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q\right)$

It ensures: $\boldsymbol{t}'[e] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[e,*]\boldsymbol{s}'^{\mathrm{T}}\right) = \boldsymbol{t}[e \oplus x] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[e \oplus x,*]\boldsymbol{s}^{\mathrm{T}}\right)$, for any $e \in \mathbb{F}_q$.

1: Generate a random matrix $\boldsymbol{R} \in \mathbb{F}_{\tilde{q}}^{d\times d}$
2: $\boldsymbol{s}' \leftarrow \sum(\boldsymbol{R}^{\mathrm{T}})$
3: $\boldsymbol{W} \leftarrow \boldsymbol{A}\boldsymbol{R}$
4: **for** each $e \in \mathbb{F}_q$ **do**
5: $\quad \boldsymbol{V}[e,*] \leftarrow (\boldsymbol{A}[e \oplus x,*] \odot \boldsymbol{s}) \ominus \boldsymbol{W}[e,*]$
6: $\quad \boldsymbol{t}'[e] \leftarrow \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}(\sum \boldsymbol{V}[e,*]) \oplus \boldsymbol{t}[e \oplus x]$
7: **end for**

Circuit 2 ToShares

Input: $(\boldsymbol{s}, \boldsymbol{t}) \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$ and $x \in \mathbb{F}_q$

Output: sharing $\hat{\boldsymbol{y}} \in \mathbb{F}_{q_o}^{d+1}$

The gadget ensures that: $\underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}\left(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\right) \oplus \boldsymbol{t}[x] = \sum \hat{\boldsymbol{y}}$.

ToShares−part1 **(pre-processing)**

Output: $\hat{\boldsymbol{y}}[0\!:\!d-1] \in \mathbb{F}_{q_o}^d, \boldsymbol{w} \in \mathbb{F}_{q_o}^d$

1: Generate a random matrix $\boldsymbol{R} \in \mathbb{F}_{q_o}^{d\times d}$
2: $\hat{\boldsymbol{y}}[0\!:\!d-1] \leftarrow \sum(\boldsymbol{R}^{\mathrm{T}})$
3: $\boldsymbol{w} \leftarrow \sum \boldsymbol{R}$

ToShares−part2 **(online phase)**

Input: $\boldsymbol{w} \in \mathbb{F}_{q_o}^d, (\boldsymbol{s}, \boldsymbol{t}) \in (\mathbb{F}_{\tilde{q}}^d, \mathbb{F}_{q_o}^q)$ and $x \in \mathbb{F}_q$

Output: $\hat{\boldsymbol{y}}[d]$

1: $\boldsymbol{v} \leftarrow \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}(\boldsymbol{A}[x,*] \odot \boldsymbol{s}) \ominus \boldsymbol{w}$
2: $\hat{\boldsymbol{y}}[d] \leftarrow \boldsymbol{t}[x] \oplus \sum \boldsymbol{v}$

Lemma 4. PackedShiftRefresh *is correct. That is, for any* $e \in \mathbb{F}_q$,

$$\boldsymbol{t}'[e] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}(\boldsymbol{A}[e,*]\boldsymbol{s}'^{\mathrm{T}}) = \boldsymbol{t}[e \oplus x] \oplus \underset{\tilde{q}\to q_o}{\mathsf{FieldMap}}(\boldsymbol{A}[e \oplus x,*]\boldsymbol{s}^{\mathrm{T}}) \ .$$

Proof. By the instruction, we have

$$\begin{aligned}
\sum \boldsymbol{W}[e,*] &= \sum \boldsymbol{A}[e,*]\boldsymbol{R} \\
&= \sum_{i=0}^{d-1} \boldsymbol{A}[e,i] \sum_{j=0}^{d-1} \boldsymbol{R}[i,j] \\
&= \boldsymbol{A}[e,*](\sum \boldsymbol{R}^{\mathrm{T}})^{\mathrm{T}} \\
&= \boldsymbol{A}[e,*]\boldsymbol{s}'^{\mathrm{T}} \ .
\end{aligned}$$

Then, we have

$$\begin{aligned}
\boldsymbol{t}'[e] &= \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}(\sum \boldsymbol{V}[e,*]) \oplus \boldsymbol{t}[e\oplus x] \\
&= \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\sum\big((\boldsymbol{A}[e\oplus x,*]\odot \boldsymbol{s})\big) \ominus \sum\big(\boldsymbol{W}[e,*]\big)\Big) \oplus \boldsymbol{t}[e\oplus x] \\
&= \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}(\boldsymbol{A}[e\oplus x,*]\boldsymbol{s}^{\mathrm{T}}) \ominus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}(\boldsymbol{A}[e,*]\boldsymbol{s}'^{\mathrm{T}}) \oplus \boldsymbol{t}[e\oplus x] \ .
\end{aligned}$$

Thus, we have $\boldsymbol{t}'[e] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[e,*]\boldsymbol{s}^{\mathrm{T}}\big) = \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[e\oplus x,*]\boldsymbol{s}^{\mathrm{T}}\big) \oplus \boldsymbol{t}[e\oplus x]$. □

Lemma 5. ToShares *is correct. That is,* $\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\big) \oplus \boldsymbol{t}[x] = \sum \hat{\boldsymbol{y}}$.

Proof. By the instruction, we have $\sum \hat{\boldsymbol{y}}[0\!:\!d-1] = \sum \boldsymbol{w}$, and thus:

$$\begin{aligned}
\hat{\boldsymbol{y}}[d] &= \boldsymbol{t}[x] \oplus \sum \boldsymbol{v} \\
&= \boldsymbol{t}[x] \oplus \sum \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[x,*]\odot \boldsymbol{s}\big) \ominus \sum \boldsymbol{w} \\
&= \boldsymbol{t}[x] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\sum(\boldsymbol{A}[x,0]\boldsymbol{s}[0],\ldots,\boldsymbol{A}[x,d-1]\boldsymbol{s}[d-1])\big) \ominus \sum \boldsymbol{w} \\
&= \boldsymbol{t}[x] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\big) \ominus \sum \boldsymbol{w} \\
&= \boldsymbol{t}[x] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\big) \ominus \sum \hat{\boldsymbol{y}}[0\!:\!d-1] \ .
\end{aligned}$$

Thus, we have $\hat{\boldsymbol{y}}[d] \oplus \sum \hat{\boldsymbol{y}}[0\!:\!d-1] = \sum \hat{\boldsymbol{y}} = \boldsymbol{t}[x] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[x,*]\boldsymbol{s}^{\mathrm{T}}\big)$. □

Theorem 1. MaskedTable *is correct. That is,* $\mathsf{S}(\sum \hat{\boldsymbol{x}}) = \sum \hat{\boldsymbol{y}}$.

Proof. By the instruction, we can see that $\boldsymbol{t}_0[e] \oplus \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[e,*]\boldsymbol{s}_0^{\mathrm{T}}\big) = \mathsf{S}(e)$ for any $e \in \mathbb{F}_q$. Then, by Lemma 4, after k^{th} iterations, we haves for any $e \in \mathbb{F}_q$,

$$\begin{aligned}
&\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\boldsymbol{A}[e,*]\boldsymbol{s}_{k+1}^{\mathrm{T}}\Big) \oplus \boldsymbol{t}_{k+1}[e] \\
=&\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\boldsymbol{A}\big[e\oplus\hat{\boldsymbol{x}}[k],*\big]\boldsymbol{s}_k^{\mathrm{T}}\Big) \oplus \boldsymbol{t}_k\big[e\oplus\hat{\boldsymbol{x}}[k]\big] \\
=&\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\boldsymbol{A}\big[e\oplus\hat{\boldsymbol{x}}[k-1],*\big]\boldsymbol{s}_{k-1}^{\mathrm{T}}\Big) \oplus \boldsymbol{t}_{k-1}\big[e\oplus\hat{\boldsymbol{x}}[k-1]\big] \\
=&\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\boldsymbol{A}\big[e\oplus\hat{\boldsymbol{x}}[k-1]\oplus\hat{\boldsymbol{x}}[k-2],*\big]\boldsymbol{s}_{k-2}^{\mathrm{T}}\Big) \oplus \boldsymbol{t}_{k-2}\big[e\oplus\hat{\boldsymbol{x}}[k-1]\oplus\hat{\boldsymbol{x}}[k-2]\big] \\
&\ \vdots \ \text{by deduction} \\
=&\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\Big(\boldsymbol{A}\big[e\oplus\sum_{i=0}^{k-1}\hat{\boldsymbol{x}}[i],*\big]\boldsymbol{s}_0^{\mathrm{T}}\Big) \oplus \boldsymbol{t}_0\big[e\oplus\sum_{i=0}^{k-1}\hat{\boldsymbol{x}}[i]\big] \\
=&\mathsf{S}\big(e\oplus\sum_{i=0}^{k-1}\hat{\boldsymbol{x}}[i]\big) \ .
\end{aligned}$$

By setting $k+1=d$, we have $\underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[e,*]\boldsymbol{s}_d^{\mathrm{T}}\big) \oplus \boldsymbol{t}_d[e] = \mathsf{S}\big(e\oplus\sum_{i=0}^{d-1}\hat{\boldsymbol{x}}[i]\big)$. At last, By setting $e=\hat{\boldsymbol{x}}[d]$ and by Lemma 5, we have

$$\mathsf{S}\big(\sum\hat{\boldsymbol{x}}\big) = \mathsf{S}\big(\hat{\boldsymbol{x}}[d]\oplus\sum_{i=0}^{d-1}\hat{\boldsymbol{x}}[i]\big) = \underset{\tilde{\mathsf{q}}\to\mathsf{q}_\mathsf{o}}{\mathsf{FieldMap}}\big(\boldsymbol{A}[\hat{\boldsymbol{x}}[d],*]\boldsymbol{s}_d^{\mathrm{T}}\big) \oplus \boldsymbol{t}_d[\hat{\boldsymbol{x}}[d]] = \sum\hat{\boldsymbol{y}}.$$

□

4.3 Security

Prior to presenting the security analysis of MaskedTable, we initially establish the security properties of PackedShiftRefresh and ToShares in Lemmas 6 and 7, respectively. It's worth noting that both PackedShiftRefresh and ToShares share similarities with SNI. However, it's essential to highlight that NI/SNI are typically defined as properties of gadgets with input and output sharings, whereas neither PackedShiftRefresh nor ToShares precisely fit this definition, as their inputs do not exclusively consist of sharings. Furthermore, despite the resemblance between Lemmas 6 and 7, they are presented separately due to the distinct proofs required for each.

Lemma 6. *Given* PackedShiftRefresh *with input variables* $\{x\}$ *and* $(\boldsymbol{s}, \boldsymbol{t})$*, along with internal and output probes denoted as* $\mathcal{P}_{int}$ *and* $\mathcal{P}_{out}$ *respectively, satisfying the condition* $|\mathcal{P}_{int}| + |\mathcal{P}_{out}| \leq d$*, we have* $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ *can be simulated with* $|\mathcal{P}_{int}|$ *variables in* $(\boldsymbol{s}, \boldsymbol{t})$ *and* $\min(1, |\mathcal{P}_{int}|)$ *variable in* $\{x\}$*.*

Proof of Lemma 6. We divide the probes $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ as follows:

- Internal probes $\mathcal{P}_{int}$:
  - Probes in the input variables: $\mathcal{P}_{input}$
  - Probes in the calculation of $\boldsymbol{s}' \leftarrow \sum(\boldsymbol{R}^{\mathrm{T}})$: $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$. For each probe p in $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$, there exists a function $\mathsf{g} : \mathbb{F}_q^d \to \mathbb{F}_q$ and an index $i \in [0 : d-1]$, such that $p = \mathsf{g}(\boldsymbol{R}[i, 0], \ldots, \boldsymbol{R}[i, d-1])$. Note that, the probes to the vector $\boldsymbol{s}'$ are excluded from $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$ since they are output probes.
  - Probes in the calculation of $\boldsymbol{W} \leftarrow \boldsymbol{AR}$: $\mathcal{P}_{\boldsymbol{AR}}$. For each probe p in $\mathcal{P}_{\boldsymbol{AR}}$, there exists a function g and an index $j \in \{0, \ldots, d-1\}$ such that $p = \mathsf{g}(\boldsymbol{R}[0, j], \ldots, \boldsymbol{R}[d-1, j])$.
  - Probes in the calculation of $\boldsymbol{t}'[e] \leftarrow \underset{\tilde{\mathsf{q}} \to \mathsf{q}_{\mathsf{o}}}{\mathsf{FieldMap}}(\sum \boldsymbol{V}[e, *]) \oplus \boldsymbol{t}[e \oplus x]$ for any $e \in \mathbb{F}_q$: $\mathcal{P}_{sum}$. Note that the probes to the vectors $\boldsymbol{t}$ and $\boldsymbol{t}'$ are excluded from $\mathcal{P}_{sum}$.
- Output probes $\mathcal{P}_{out}$:
  - The probes in $\boldsymbol{s}'$: $\mathcal{P}_{\boldsymbol{s}'}$.
  - The probes in $\boldsymbol{t}'$: $\mathcal{P}_{\boldsymbol{t}'}$.

We build sets $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$, $\mathcal{S}_x$, $\mathcal{I}_{sum}$ and $\mathcal{J}$ and run a simulator as following steps.

1. Initiate sets $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$, $\mathcal{I}_{sum}$ and $\mathcal{J}$ to be empty. If $\mathcal{P}_{int} \neq \emptyset$, let $\mathcal{S}_x = x$, otherwise let $\mathcal{S}_x = \emptyset$. After this step, we have $\mathcal{S}_x = \begin{cases} x & \text{if } |\mathcal{P}_{int}| \geq 1 \\ \emptyset & \text{if } |\mathcal{P}_{int}| = 0 \end{cases}$.
2. For the probes in $\mathcal{P}_{input}$, put them (excluding x, since it should be in $\mathcal{S}_x$) into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$, and thus they can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}|$.
3. For each probe in $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$ or $\mathcal{P}_{\boldsymbol{s}'}$, it can be simulated by sampling the corresponding row $\boldsymbol{R}[i, *]$ from a uniform distribution. In this step, $|\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| + |\mathcal{P}_{\boldsymbol{s}'}|$ rows of $\boldsymbol{R}$ are simulated, and the rest $d - |\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| + |\mathcal{P}_{\boldsymbol{s}'}|$ rows are still uniformly distributed. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}|$.

4. For each probe in $\mathcal{P}_{\boldsymbol{AR}}$, it can be simulated by sampling the corresponding column $\boldsymbol{R}[*, j]$ from a uniform distribution, and we put j into $\mathcal{J}$ and $\boldsymbol{s}[j]$ into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$. Let $\bar{\mathcal{J}} \stackrel{\text{def}}{=} \{0, \ldots, d-1\} \setminus \mathcal{J}$. As $\boldsymbol{A}$ is MDS, by Lemma 1, any $d - |\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| - |\mathcal{P}_{\boldsymbol{s}'}|$ rows of $\boldsymbol{W}[*, \bar{\mathcal{J}}]$ are uniformly distributed after this step. Besides, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{AR}}|$.

5. For each probe in $\mathcal{P}_{sum}$, if the corresponding index i is not in $\mathcal{I}_{sum}$, put i into $\mathcal{I}_{sum}$, and simulate the elements in $\boldsymbol{V}[i, \bar{\mathcal{J}}]$ from a uniform distribution. This can be done since $|\mathcal{I}_{sum}| \leq d - |\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| - |\mathcal{P}_{\boldsymbol{s}'}|$. we then put $\boldsymbol{t}[i \oplus x]$ into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$. This probe is determined by $\boldsymbol{V}[i, \bar{\mathcal{J}}]$, $\boldsymbol{t}[i \oplus x]$, $\boldsymbol{s}[\mathcal{J}]$, $\boldsymbol{W}[i, \mathcal{J}]$ and x, where $\boldsymbol{V}[i, \bar{\mathcal{J}}]$ and $\boldsymbol{W}[i, \mathcal{J}]$ have been simulated, and we also have $\{\boldsymbol{t}[i \oplus x]\} \cup \boldsymbol{s}[\mathcal{J}] \subseteq \mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$. Thus, this probe can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{AR}}| + |\mathcal{P}_{sum}|$.

6. Finally, we consider the probes in $\mathcal{P}_{\boldsymbol{t}'}$. For each probe in $\mathcal{P}_{\boldsymbol{t}'}$ (say, $\boldsymbol{t}[i]$), we separate the analysis as follows.

   - If $i \notin \mathcal{I}_{sum}$, since $|\mathcal{I}_{sum}| + |\mathcal{P}_{\boldsymbol{t}'}| + |\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| + |\mathcal{P}_{\boldsymbol{s}'}| \leq d$, we can simulate the probe from a uniform distributions.
   - If $i \in \mathcal{I}_{sum}$, the probe is determined by $\boldsymbol{V}[i, \bar{\mathcal{J}}]$, $\boldsymbol{t}[i \oplus x]$, $\boldsymbol{s}[\mathcal{J}]$, $\boldsymbol{W}[i, \mathcal{J}]$ and x, where $\boldsymbol{V}[i, \bar{\mathcal{J}}]$ and $\boldsymbol{W}[i, \mathcal{J}]$ have been simulated, and $\boldsymbol{t}[i'] \cup \boldsymbol{s}[\mathcal{J}] \subseteq \mathcal{S}$. Thus, it can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$.

   After all steps, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{AR}}| + |\mathcal{P}_{sum}| \leq |\mathcal{P}_{int}|$ and $\mathcal{S}_x = \begin{cases} x & \text{if } |\mathcal{P}_{int}| \geq 1 \\ \emptyset & \text{if } |\mathcal{P}_{int}| = 0 \end{cases}$.

Now, all the probes are simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. Therefore, for any internal probes $\mathcal{P}_{int}$ and output probes $\mathcal{P}_{out}$ such that $|\mathcal{P}_{int}| + |\mathcal{P}_{out}| \leq d$, $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ can be simulated with $|\mathcal{P}_{int}|$ variables in $(\boldsymbol{s}, \boldsymbol{t})$ and $\min(1, |\mathcal{P}_{int}|)$ variable in $\{x\}$. □

Lemma 7. *Given* ToShares *with input variables* $\{x\}$ *and* $(\boldsymbol{s}, \boldsymbol{t})$*, along with internal and output probes denoted as* $\mathcal{P}_{int}$ *and* $\mathcal{P}_{out}$ *respectively, satisfying the condition* $|\mathcal{P}_{int}| + |\mathcal{P}_{out}| \leq d$*, we have* $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ *can be simulated with* $|\mathcal{P}_{int}|$ *variables in* $(\boldsymbol{s}, \boldsymbol{t})$ *and* $\min(1, |\mathcal{P}_{int}|)$ *variable in* $\{x\}$*.*

Proof of Lemma 7. We divide the probes $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ as follows:

- Internal probes $\mathcal{P}_{int}$:
  - Probes in the input variables: $\mathcal{P}_{input}$.
  - Probes in the calculation of $\hat{\boldsymbol{y}}[0\!:\!d-1] \leftarrow \sum(\boldsymbol{R}^{\mathrm{T}})$: $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$. For each probe p in $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$, there exists a function $\mathsf{g} : \mathbb{F}_q^d \to \mathbb{F}_q$ and an index $i \in [1\!:\!d]$, such that $p = \mathsf{g}(\boldsymbol{R}[i, 0], \ldots, \boldsymbol{R}[i, d-1])$. Note that, the probes to the vector $\hat{\boldsymbol{y}}[0\!:\!d-1]$ are excluded from $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$ since they are output probes.
  - Probes in the calculation of $\boldsymbol{w} \leftarrow \sum \boldsymbol{R}$: $\mathcal{P}_{\boldsymbol{R}}$. For each probe p in $\mathcal{P}_{\boldsymbol{R}}$, there exists a function $\mathsf{g} : \mathbb{F}_q^d \to \mathbb{F}_q$ and an index $j \in \{0, \ldots, d-1\}$, such that $p = \mathsf{g}(\boldsymbol{R}[0, j], \ldots, \boldsymbol{R}[d-1, j])$.
  - Probes in the calculation of $\hat{\boldsymbol{y}}[d] \leftarrow \boldsymbol{t}[x] \oplus \sum \boldsymbol{v}$: $\mathcal{P}_{sum}$. Note that the probe to the variable $\hat{\boldsymbol{y}}[d]$ is excluded from $\mathcal{P}_{sum}$ since it is an output share.
- Output probes $\mathcal{P}_{out}$:
  - Probes to $\hat{\boldsymbol{y}}[0\!:\!d-1]$: $\mathcal{P}_{\hat{\boldsymbol{y}}[0:d-1]}$.

– Probes to $\hat{\boldsymbol{y}}[d]$: $\mathcal{P}_{\hat{\boldsymbol{y}}[d]}$.

We build sets $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$, $\mathcal{S}_x$ and a temporary set $\mathcal{J}$ and run a simulator as following steps.

1. Initiate sets $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{J}$ to be empty. If $|\mathcal{P}_{int}| > 0$, let $\mathcal{S}_x = x$, otherwise let $\mathcal{S}_x = \emptyset$. After this step, we have $\mathcal{S}_x = \begin{cases} x & \text{if } |\mathcal{P}_{int}| \geq 1 \\ \emptyset & \text{if } |\mathcal{P}_{int}| = 0 \end{cases}$.

2. For the probes of $\mathcal{P}_{input}$, put them (excluding x, since it should be in $\mathcal{S}_x$) into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$, and thus they can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}|$.

3. For each probe in $\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}$ or $\mathcal{P}_{\hat{\boldsymbol{y}}[0..d-1]}$, it can be simulated by sampling the corresponding row $\boldsymbol{R}[i,*]$ from a uniform distribution. In this step, at most $|\mathcal{P}_{\boldsymbol{R}^{\mathrm{T}}}| + |\mathcal{P}_{\hat{\boldsymbol{y}}[0..d-1]}|$ rows of $\boldsymbol{R}$ are simulated.

4. For each probe in $\mathcal{P}_{\boldsymbol{R}}$, it can be simulated by sampling the corresponding column $\boldsymbol{R}[*,j]$ from a uniform distribution, and we put j into $\mathcal{J}$ and $\boldsymbol{s}[j]$ into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$. Now, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{int}|$. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{w}}|$.

5. The probes in $\mathcal{P}_{sum}$ are determined by $\boldsymbol{v}$ and $\boldsymbol{t}[x]$. We put $\boldsymbol{t}[x]$ into $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and simulate the elements in $\boldsymbol{v}[\bar{\mathcal{J}}]$ from a uniform distribution. Probes in $\mathcal{P}_{sum}$ are determined by $\boldsymbol{v}[\bar{\mathcal{J}}]$, $\boldsymbol{t}[i]$, $\boldsymbol{s}[\mathcal{J}]$, $\boldsymbol{w}[\mathcal{J}]$ and x, where $\boldsymbol{v}[\bar{\mathcal{J}}]$ and $\boldsymbol{w}[\mathcal{J}]$ have been simulated, and $\boldsymbol{t}[\mathcal{K}_1'] \cup \boldsymbol{s}[\mathcal{J}] \subseteq \mathcal{S}$. Thus, probes in $\mathcal{P}_{sum}$ can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. After this step, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{w}}| + |\mathcal{P}_{sum}|$.

6. Finally, we consider $\mathcal{P}_{\hat{\boldsymbol{y}}[d]}$. We have $\hat{\boldsymbol{y}}[d] = \boldsymbol{v}[\bar{\mathcal{J}}] \oplus \boldsymbol{s}[\mathcal{J}] \oplus \boldsymbol{w}[\mathcal{J}] \oplus \boldsymbol{t}[k]$. We separate our analysis into two cases:

   - If $\mathcal{P}_{sum} = \emptyset$, we can simulate $\hat{\boldsymbol{y}}[d]$ from a uniform distribution.
   - If $\mathcal{P}_{sum} \neq \emptyset$, then $x \in \mathcal{S}_x$. We also have $\hat{\boldsymbol{y}}[d]$ is determined by $\boldsymbol{v}[\bar{\mathcal{J}}]$, $\boldsymbol{t}[x]$, $\boldsymbol{s}[\mathcal{J}]$, $\boldsymbol{w}[\mathcal{J}]$ and x, where $\boldsymbol{v}[\bar{\mathcal{J}}]$ and $\boldsymbol{w}[\mathcal{J}]$ have been simulated, and $\boldsymbol{t}[x] \cup \boldsymbol{s}[\mathcal{J}] \subseteq \mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$. Thus, it can be simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$.

   After all steps, we have $|\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}| \leq |\mathcal{P}_{input}| + |\mathcal{P}_{\boldsymbol{AR}}| + |\mathcal{P}_{sum}| \leq |\mathcal{P}_{int}|$ and $\mathcal{S}_x = \begin{cases} x & \text{if } |\mathcal{P}_{int}| \geq 1 \\ \emptyset & \text{if } |\mathcal{P}_{int}| = 0 \end{cases}$.

Now, all the probes are simulated with $\mathcal{S}_{(\boldsymbol{s},\boldsymbol{t})}$ and $\mathcal{S}_x$. Therefore, for any internal probes $\mathcal{P}_{int}$ and output probes $\mathcal{P}_{out}$ such that $|\mathcal{P}_{int}| + |\mathcal{P}_{out}| \leq d$, $\mathcal{P}_{int} \cup \mathcal{P}_{out}$ can be simulated with $|\mathcal{P}_{int}|$ variables in $(\boldsymbol{s},\boldsymbol{t})$ and $\min(1, |\mathcal{P}_{int}|)$ variable in $\{x\}$. □

Theorem 2. MaskedTable *is SNI.*

Proof. Let

$$\begin{aligned}
&\mathsf{Refresh}_0 \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}((\boldsymbol{s}_0, \boldsymbol{t}_0), \hat{\boldsymbol{x}}[0]) \\
&\mathsf{Refresh}_1 \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}((\boldsymbol{s}_1, \boldsymbol{t}_1), \hat{\boldsymbol{x}}[1]) \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \\
&\mathsf{Refresh}_{d-1} \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}((\boldsymbol{s}_{d-1}, \boldsymbol{t}_{d-1}), \hat{\boldsymbol{x}}[d-1]) \\
&\mathsf{Refresh}_d \stackrel{\text{def}}{=} \mathsf{ToShares}((\boldsymbol{s}_d, \boldsymbol{t}_d), \hat{\boldsymbol{x}}[d])
\end{aligned}$$

and let

$$\begin{aligned}
&\mathsf{C}_0 \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}\big((\boldsymbol{s}_0, \boldsymbol{t}_0), \hat{x_0}\big)\\
&\mathsf{C}_1 \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}\Big(\mathsf{C}_0\big((\boldsymbol{s}_0, \boldsymbol{t}_0), \hat{x_0}\big), \hat{x}_1\Big)\\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad .\\
&\mathsf{C}_{d-1} \stackrel{\text{def}}{=} \mathsf{PackedShiftRefresh}\Big(\mathsf{C}_{d-2}\big((\boldsymbol{s}_0, \boldsymbol{t}_0), \hat{x}_{d-2}\big), \hat{x}_{d-2}\Big)\\
&\mathsf{C}_d \stackrel{\text{def}}{=} \mathsf{ToShares}\Big(\mathsf{C}_{d-1}\big((\boldsymbol{s}_0, \boldsymbol{t}_0), \hat{x}_{d-1}\big), \hat{x}_{d-1}\Big)
\end{aligned}$$

Let $\mathcal{P}_{int_k}$ and $\mathcal{P}_{out_k}$ be the internal and output probes of C_k, and let $\mathcal{P}'_{int_k}$ and $\mathcal{P}'_{out_k}$ be the internal and output probes of $\mathsf{Refresh}_k$. Without loss of generality, for any $k \in \{1, \ldots, d\}$, we count the probes in $(\boldsymbol{s}_k, \boldsymbol{t}_k)$ as the internal probes of $\mathsf{Refresh}_k$ or C_k rather than the output probes of $\mathsf{Refresh}_{k-1}$ or C_{k-1}. It means that, for any $k = \{0, \ldots, d-1\}$, $\mathcal{P}_{out_k} = \emptyset$ and $\mathcal{P}'_{out_k} = \emptyset$. Besides, we have $\mathsf{C}_d = \mathsf{MaskedTable}$, and thus internal and output probes of $\mathsf{MaskedTable}$ are $\mathcal{P}_{int_d}$ and $\mathcal{P}_{out_d}$, respectively. We illustrate the above partition in Figure 4.

Assuming $\mathcal{P}_{int_k} \cup \mathcal{P}_{out_k}$ can be simulated by $|\mathcal{P}_{int_k}|$ shares in $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ and $\min\big(|\mathcal{P}_{int_k}|, k+1\big)$ shares in $\hat{\boldsymbol{x}}$, we aims at proving that $\mathcal{P}_{int_{k+1}} \cup \mathcal{P}_{out_{k+1}}$ can be simulated by $|\mathcal{P}_{int_{k+1}}|$ shares in $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ and $\min\big(|\mathcal{P}_{int_{k+1}}|, k+2\big)$ shares in $\hat{\boldsymbol{x}}$. First of all, we have $\mathcal{P}_{int_{k+1}} = \mathcal{P}_{int_k} \cup \mathcal{P}_{out_k} \cup \mathcal{P}'_{int_{k+1}}$ and $\mathcal{P}_{out_{k+1}} = \mathcal{P}'_{out_{k+1}}$. By Lemma 6, $\mathcal{P}'_{int_{k+1}}$ and $\mathcal{P}'_{out_{k+1}}$ can be simulated with $|\mathcal{P}'_{int_{k+1}}|$ variables in $(\boldsymbol{s}_{k+1}, \boldsymbol{t}_{k+1})$ and $\min(1, |\mathcal{P}'_{int_{k+1}}|)$ variable in $\{\hat{x}[k+1]\}$. Further, as $|\mathcal{P}'_{int_{i+1}}| + |\mathcal{P}_{out_k}| + |\mathcal{P}_{int_k}| \leq d$, $\mathcal{P}_{int_k} \cup \mathcal{P}_{out_k}$ and $|\mathcal{P}'_{int_{k+1}}|$ variables in $(\boldsymbol{s}_{k+1}, \boldsymbol{t}_{k+1})$ can be simulated with $|\mathcal{P}_{int_k}|$ shares in $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ and $\min\big(|\mathcal{P}_{int_k}|, k+1\big)$ shares in $\hat{\boldsymbol{x}}$. Thus, $\mathcal{P}_{int_{k+1}} \cup \mathcal{P}_{out_{k+1}}$ can be simulated by $|\mathcal{P}_{int_{k+1}}|$ shares in $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ and $\min\big(|\mathcal{P}_{int_{k+1}}|, k+2\big)$ shares in $\hat{\boldsymbol{x}}$.

As last, by deduction, we have $\mathcal{P}_{int_d} \cup \mathcal{P}_{out_d}$ can be simulated with $|\mathcal{P}_{int_d}|$ shares in $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ and $\min\big(|\mathcal{P}_{int_d}|, d+1\big) = |\mathcal{P}_{int_d}|$ shares in $\hat{\boldsymbol{x}}$, indicating that $\mathsf{MaskedTable}$ is SNI. □

5 A Discussion on the Overhead of Random Bits Generation

As most masking schemes (including ours) require true random bits, the deployment should naturally target the microprocessors that contain a True Random Number Generator (TRNG). In this respect, we investigate how the commonly used TRNG meets the requirement of our masking scheme. Usually, the TRNG is composed of a live entropy source (analog) and an internal conditioning component, and it has the following features:

1. It can provide full entropy outputs periodically.
2. The random bits generation process is independent of (and in parallel with) the running of the program.

Figure 5 illustrates the platform for the software implementation of the masking. The TRNG independently generates a batch of random bits, and then requests an interrupt (indicating that the random bits are ready) to the processor. Subsequently, an interrupt handler stores the random bits in the RAM buffer.

An ideal strategy to execute a masked program is to make TRNG run in parallel. Concretely, one can perform the program with the TRNG enabled. Once a batch of random bits (e.g., a 32-bit true random number) is ready, the program is interrupted, and an interrupt handler stores the random bits in a buffer. The program directly accesses

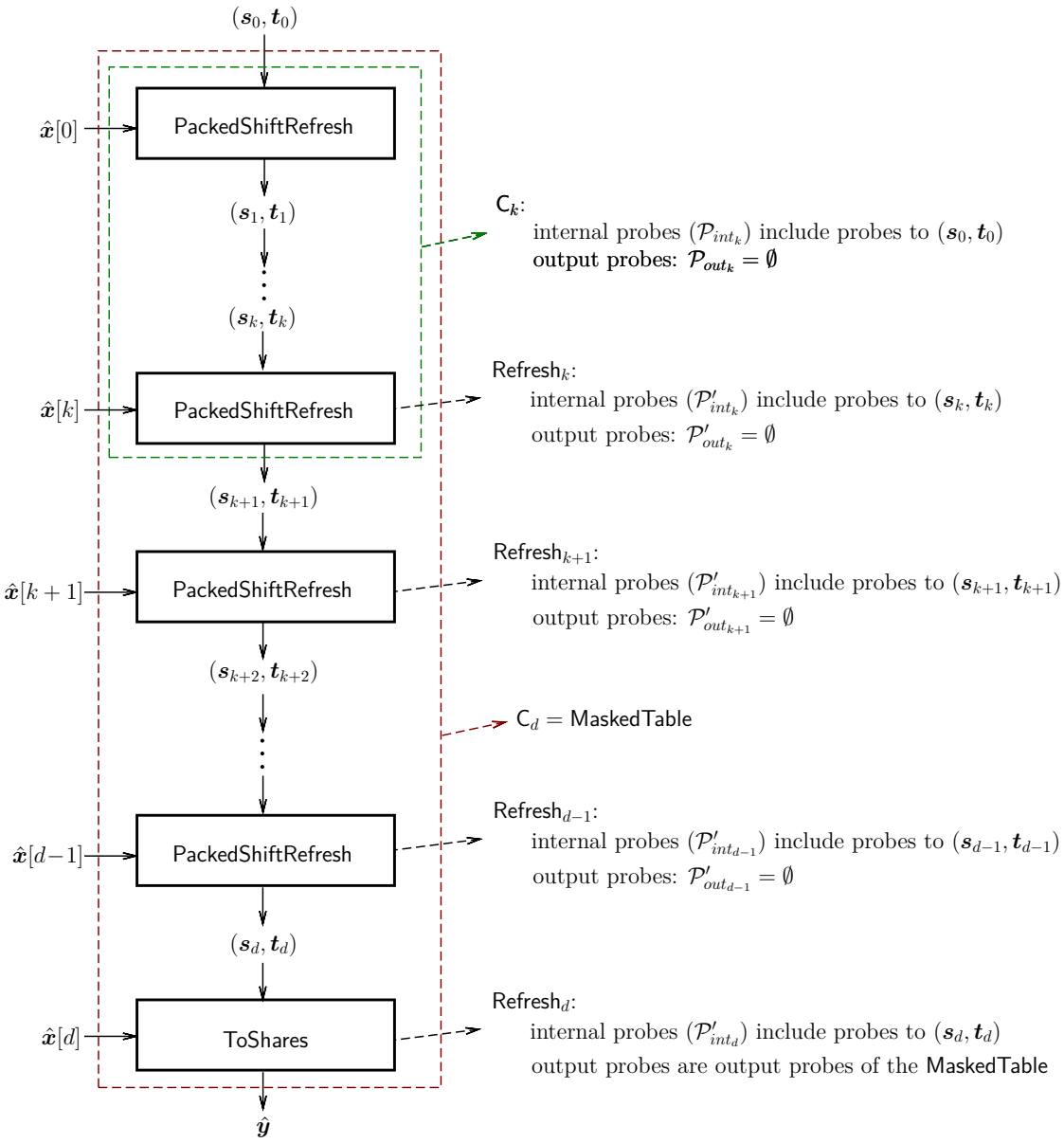


Figure 4: The partition of probes in MaskedTable.

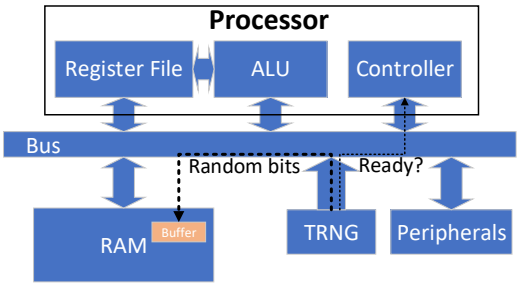


Figure 5: Illustration of the platform for the software implementation of the masking.

the buffer when it requires random bits. Therefore, the storing of the random bits (to the buffer) dominates the overhead running time of the random number generation is only the storing of the random bits (to the buffer), as long as the buffer always provides sufficient random bits (i.e., the program does not need to wait for the TRNG).

We consider the case that the masked implementation requires c-bit of randomness every h cycles, and the TRNG can provide c'-bit of randomness every h' cycles. We can see that, the buffer always provides sufficient random bits, if $\frac{c'}{h'} \geq \frac{c}{h}$.

We then investigate the performance of known TRNGs. From the academic side, we refer to the recent work [KHL21] that provided comparisons of 8 TRNGs in FPGA and 17 ones in ASIC. Based on the comparison, there exist multiple candidates (with quite a small footprint) whose performances are faster than 4 Mbps. Assuming the running frequency of the microprocessor is up to 100 MHz, we have $\frac{c'}{h'} \geq 0.4$. From the industry side, the TRNG peripheral embedded in STM32 can provide 128-bit random samples every $200 \sim 400$ cycles, and thus $c' = 128$ [2]. For a very conservative evaluation, we consider $h' = 400$, and thus $\frac{c'}{h'} = 0.32$.

The pre-processing of MaskedTable requires $c = \lceil \log_2 \tilde{q} \rceil \, d^2$ random bits for each call of PackedShiftRefresh. There exist qd^2 field multiplications in PackedShiftRefresh. Supposing each field multiplication takes τ cycles, we have $h \geq qd^2\tau$, and $\frac{c}{h} \leq \frac{\lceil \log_2 \tilde{q} \rceil d^2}{qd^2\tau} = \frac{\lceil \log_2 \tilde{q} \rceil}{q\tau}$. In the following, we consider two different sizes of lookup tables.

- We consider the case of $q = q_o = 2^8$ and $\tilde{q} = 2^9$ that can be instantiated as the 8-bit to 8-bit AES S-box shown later in Section 6.1. Then, we have $\frac{c}{h} \leq \frac{0.0352}{\tau} \leq 0.0352 < \frac{c'}{h'}$.
- We also consider the case of a smaller lookup table, where $q = q_o = 2^4$ and $\tilde{q} = 2^5$. Then, we have $\frac{c}{h} \leq \frac{0.3125}{\tau} \leq 0.3125 < \frac{c'}{h'}$.

The above conveys that the overhead running time corresponding to the random number generation can be quite small (only the storing of the random bits to the buffer).

It should be noted that the above discussion only considers the speed of the pre-processing. At the same time, the generation of random bits also impacts other aspects such as the power consumption.

6 Applications to AES and Security Analysis in Practice

6.1 Applications to AES-128

AES is a block cipher that is performed on 16 variables in $\mathbb{F}_{2^8}$ called state. The block size of AES is 128 bits, but it has three different key lengths: 128, 192 and 256 bits. In this paper, we consider AES-128 that is the variant with 128 bits of key length. The round function of AES comprises four transformations over the state: AddRoundKey, SubBytes, ShiftRows and MixColumns, where AddRoundKey, ShiftRows and MixColumns are linear and can be implemented using Gadget 1. We then focus on the masked implementation of SubBytes, which is made up of 16 S-boxes (that can be regarded as 16 lookup tables) in parallel. More details of AES block cipher can be found in e.g., [DR02].

AES S-box is a function $\mathsf{S} : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$. To adapt our new gadget, we set $q_1 = q_2 = 2^8$, and set $\tilde{q} = 2^9$. We store each 9-bit variable using a 16-bit RAM unit. Thus, it requires $d \times 2 + d + 256 = 256 + 3d$ Bytes of RAM for pre-computed variables (e.g., $(\boldsymbol{s}, \boldsymbol{t}) \in (\mathbb{F}_{2^9}^d, \mathbb{F}_{2^8}^{256})$ and $\boldsymbol{w} \in \mathbb{F}_{2^8}^d$) of one S-box. Considering that AES-128 consists of 160 calls of S-box, it requires $(256 + 3d) \times 160$ Bytes $= 40 + 0.47d$ KBytes of RAM for all pre-computed variables.

[2] See, e.g., the programming manual provided in https://www.st.com/resource/en/reference_manual/rm0432-stm32l4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

MaskedTable is made up of $d-1$ calls of PackedShiftRefresh and 1 call of ToShares. Each call of PackedShiftRefresh or ToShares takes $\lceil \log_2 \tilde{q} \rceil \, d^2$ random bits. Thus, each call of MaskedTable takes $\lceil \log_2 \tilde{q} \rceil \, d^3$ random bits. For AES Sbox, it takes $9d^3$ random bits. Considering that AES-128 consists of 160 calls of S-box, it requires $160 \cdot 9d^3$ random bits.

We implement the masked AES-128 on the ARM Cortex M architecture. The field multiplication is performed using the log-exp method [GR17]. That is, for multipliers x and y and a primitive element a, we first map the multipliers to x' and y' such that $a^{x'} = x$ and $a^{y'} = y$, and then we calculate $z' = x' + y'$ and finally map the z' to z such that $z = a^{z'} = a^{x'+y'} = xy$. The above two mappings are performed by using lookup tables, and we also need to additionally handle the case when $x = 0$ or $y = 0$.

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 matrix $\boldsymbol{A}$ remains constant across all implementations, allowing us to embed a predetermined MDS matrix derived from the Vandermonde matrix. The selection of $\boldsymbol{A}$ does not impact the performance outcomes, due to our utilization of the log-exp table for field multiplication.

Our results and state-of-the-art implementations are shown in Table 2. We consider the schemes proposed in [VV21, WGY+22, WJZY23, AVV23, GR17] as the benchmarks for comparison. We provide the implementation results regarding running cycles, random bits, RAM size for precomputed variables, and code size. For the line work of TBM schemes, the work [VV21] provides the first TBM suitable to the precomputation paradigm. [AVV23] is confined to the third masking order, thus we specifically provided the performance of our scheme with $d = 3$. We omitted the results of [Cor14, CRZ18] because the focus of our paper is primarily on a TBM scheme designed for deployment on devices with limited RAM capacity (RAM capacity $<$ 100KB). As we recall, [Cor14, CRZ18] requires approximately 40(d + 1) KBytes of RAM. For example, when $d = 2$, the required RAM capacity would be 120KB (which is higher than we expected), and when $d = 10$, it would be 440KB. Consequently, those works do not align with the purpose of our paper. As the other line work of circuit-based schemes, Wang et al. proposed a scheme [WGY+22] for the cost amortization, with a byproduct that most intermediates can be precomputed before the input shares are accessed. This work shows that the masking schemes without lookup tables can also be compatible with the precomputation paradigm. Recently, a more efficient scheme has been provided in [WJZY23]. Besides, we add the performance of masking without pre-processing [GR17].

Compared with the state-of-the-art schemes, we believe ours has quite valuable significance in practice for resource-constrained scenarios such as the IoT. Software implementations tailored for IoT devices operate on embedded processors characterized by features of low power consumption, constrained RAM/ROM capacities, and low clock frequencies, where the power consumption is closely related to the number of cycles. Therefore, we place particular emphasis on evaluating the number of cycles, as well as the RAM requirements and code size for an implementation in ARM Cortex M architecture.

For security order $d = 8$, the online phase of our implementation runs in 47.81 Kcycles, and the pre-processing takes 622.77 Mcycles to produce 43.76 Kbytes of precomputed variables. In spite of the fact that our scheme required much more RAM space (for precomputed tables) than the related works, we believe it has quite valuable significance in practice, especially for resource-constrained scenarios such as the IoT. For example, more than 30% STM32 Mainstream microcontrollers provide at least 64 KBytes of RAM[3], which is sufficient for deploying our scheme with security orders $d = 1$ to 16. Besides, the RAM will be free after the running of AES. At last, the (speed) impact of the random bits generation of our masking can be balanced out, as we have discussed in Section 5 that all random bits can be generated in parallel with the running of the masked AES for any

[3]See, e.g., the descriptions of STM32 mainstream MCUs in https://www.st.com/en/microcontrollers-microprocessors/stm32-mainstream-mcus.html#products

security orders.

Regarding the randomness generation in our scheme, we consider the case that it is executed with a TRNG running in parallel, ensuring a constant supply of fresh random bits for the masked computation at regular intervals. As elucidated in Section 5, the additional time overhead of random number generation in our scheme only encompasses the process of moving the generated random bits to a buffer. Considering the ARM Cortex M4 architecture, the loading of a 32-bit number requires 2 cycles, followed by an additional 2 cycles for storage (into the buffer). The overhead running time can thus be calculated as $4n/32$, where n is the number of required random bits. For instance, in the case of AES-128 with $d = 2$, the calculated overhead running time is 0.18 Mcycles, which is far lower than 1% of the overall precomputation time.

Last but not least, we make the source codes of our AES-128 available on github.com/wjwangcrypto/TBMwithPreprocessing.

6.2 Practical Evaluations

Our masked implementation has been proven to achieve d-probing security, ensuring that any d intermediate variables remain independent of the secret. However, in practical threat scenarios, the measured side-channel information may expose all the intermediate variables, albeit with a certain level of noise. Existing research (e.g., [DDF14, DFS15]) has demonstrated that if an implementation is d-probing secure, its security in a practical threat model increases exponentially with d, under the assumption of two critical conditions that should hold in practice.

The first assumption is noisy leakage, indicating that the side-channel information must contain a certain level of noise. The second assumption involves independent leakage, assuming independence between the leakages corresponding to different intermediates. As the noise level is usually directly related to the characteristics of the target platform, our focus is on validating the assumption of independent leakage, along with assessing the security order. A commonly employed approach for this purpose is the Test Vector Leakage Assessment (TVLA) [Sta18].

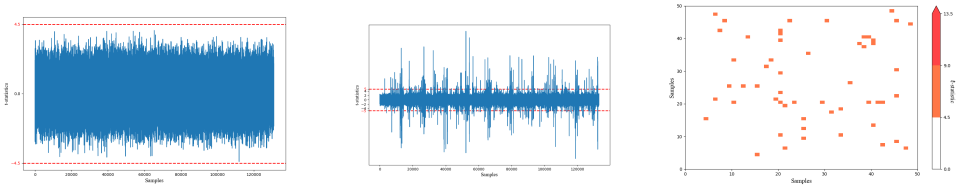
To validate the security order of the masked AES in practice, we conducted experiments using a ChipWhisperer STM32F303 UFO target board (with the frequency of 8Mhz) connecting to a PC with 12th Gen Intel(R) Core(TM) i7-127002.10 Ghz. We target the AES-128 with security order of $d = 1$. The MDS matrix $\boldsymbol{A}$ can be built from the Vandermonde matrix (see, Section 2.2). The round keys are pre-extended and stored in a masked form. The power traces of the full round AES are collected using Picoscope 5244D at a sampling rate of 31.25 MS/s. We utilize traces for each fixed vs. random Welch's T-test.

In terms of the pre-processing phase, it is important to note that all variables involved are independent of the secure input. Consequently, the pre-processing does not disclose any information about the secret, and thus has no leakage. Figure 6(a) depicts the first-order T-test results of the online phase with 0.5 million traces. In contrast, Figure 6(b) shows the result for the implementation with 10 000 traces when the randomness source (TRNG) is disabled (meaning that the random bits are all zeros). The comparison between these two figures reveals that our implementation with a security order of $d = 1$ exhibits no detectable first-order leakage.

We also conduct a second-order t-test on the implementation as shown in Figure 6(c), with a specific focus on the first round of AES with 0.5 million traces. This strategy stems from the substantial time required for the t-test evaluation, with the first round alone demanding 12 hours. Extrapolating from this, the evaluation of the entire round would take 100 times that duration.

Table 2: Summary of masked AES implementations.

d	Scheme	Mcycles for precomp.	Random bits	RAM for precomp.	Kcycles for online-comp.	Code Size
1	TBM, our work	6.37	0.176 KB	40.47 KB	28.77	23.98 KB
2	No pre-proc., [GR17]	–	3.75 KB	–	83.9	7.5 KB
	Mult. gadget, [WGY+22]	0.705	0.094 KB	5.63 KB	60	unreported
	Mult. gadget, [WJZY23]	**0.068**	2.22 KB	**2.91 KB**	50.3	unreported
	TBM, [VV21]	72.59	**0.011 KB**	40.1 KB	423	unreported
	TBM, our work	22.35	1.406 KB	40.94 KB	**31.49**	24.23 KB
3	TBM, [AVV23]	**3.80**	40.62 KB	87.22 KB	100	26.5 KB
	TBM, our work	54.65	**4.86 KB**	**41.41 KB**	**37.58**	24.51 KB
8	No pre-proc., [GR17]	–	45 KB	–	404.5	unreported
	Mult. gadget, [WGY+22]	3.66	1.5 KB	**11 KB**	137	unreported
	Mult. gadget, [WJZY23]	**0.45**	23.88 KB	11.66 KB	92.27	unreported
	TBM, [VV21]	3 265.3	**0.56 KB**	40.8 KB	2 873	unreported
	TBM, our work	622.77	90 KB	43.76 KB	**47.81**	26.46 KB
10	No pre-proc., [GR17]	–	68.75KB	–	563.9	unreported
	TBM, our work	1 140	175.78 KB	44.7 KB	53.25	27.48 KB
12	No pre-proc., [GR17]	–	97.5KB	–	749.5	unreported
	TBM, our work	1 888.5	303.75 KB	45.64 KB	58.69	28.68 KB
14	No pre-proc., [GR17]	–	131.25KB	–	961.3	unreported
	TBM, our work	2 899.7	482.34 KB	46.58 KB	64.13	29.96 KB
16	No pre-proc., [GR17]	–	170KB	–	1199.4	unreported
	Mult. gadget, [WGY+22]	12.23	6KB	18.2KB	239	unreported
	TBM, our work	4 225.5	720 KB	47.52 KB	69.57	31.41 KB



(a) TRNG is on, first order (b) TRNG is off, first order (c) TRNG is on, second order

Figure 6: First-order T-test results of AES-128 with $d = 1$.

7 Conclusion and Future Works

In this paper, we introduce a new TBM that is compatible with the pre-processing paradigm. Our scheme is designed to address the challenges of available RAM size for precomputed variables and the speed of the online phase. Both the theoretical asymptotic analysis and application to AES in practice show that the scheme is specifically tailored to optimize the utilization of RAM resources for precomputed variables while maintaining efficient execution during the online phase. The security of our scheme is proved rigorously and validated by performing T-test evaluation in practice. We believe a promising future work is to investigate the masking schemes in the precomputation-based paradigm for the protection of other crypto-systems such as post-quantum cryptography. Moreover, all TBM schemes remain vulnerable to some dedicated side-channel attacks exploiting the repeated use of shares, for instance, in TBM scheme, including our own, every input share is utilized $O(2^m)$ times. In scenarios where leakage noise is low, adversaries can exploit the repetition of input shares to effectively nullify the masking with high probability. A notable instance of such attacks is the horizontal attack [BCPZ16]. Other techniques like soft analytical side-channel attacks [BCS21], side-channel countermeasure dissection [BS20], and deep learning-based SCA [PWP22, PPM+23] demonstrated considerable effectiveness in executing such dedicated attacks. Additionally, various studies have highlighted concerns regarding TBM schemes [TWO13, BGNT18]. On the other hand, it should be noted that those attacks only succeed in low-noise scenarios and do not contradict the concept of the probing model, where the security increases exponentially with the security order d in high-noise conditions. The random probing model [ISW03] effectively captures repeated leakage of shares. Given the recent advancements [BCP+20, BRT21, BRTV21] of the masking schemes in the random probing model, we believe proposing secure TBM schemes under the random probing model in the future is of great importance.

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 (Nos. 2021YFA1000600), the National Natural Science Foundation of China (Grant Nos. 62372273, 62125204 and 61872236), 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). Yu Yu also acknowledges the support from the XPLORER PRIZE.

References

[AVV23] Anju Alexander, Annapurna Valiveti, and Srinivas Vivek. A faster third-order masking of lookup tables. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):538–556, 2023.

[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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 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.

[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.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *CHES 2016*, pages 23–39, 2016.

[BCS21] Olivier Bronchain, Gaëtan Cassiers, and François-Xavier Standaert. Give me 5 minutes: Attacking ASCAD with a single side-channel trace. *IACR Cryptol. ePrint Arch.*, page 817, 2021.

[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.

[BGNT18] Nicolas Bruneau, Sylvain Guilley, Zakaria Najm, and Yannick Teglia. Multivariate high-order attacks of shuffled tables recomputation. *J. Cryptol.*, 31(2):351–393, 2018.

[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.

[BS20] Olivier Bronchain and François-Xavier Standaert. Side-channel countermeasures' dissection and the limits of closed source security evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):1–25, 2020.

[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.

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

[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.

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

[IKL+13] Yuval Ishai, Eyal Kushilevitz, Xin Li, Rafail Ostrovsky, Manoj Prabhakaran, Amit Sahai, and David Zuckerman. Robust pseudorandom generators. In *ICALP 2013(1)*, pages 576–588, 2013.

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

[KHL21] Netanel Klein, Eyal Harel, and Itamar Levi. The cost of a true random bit - on the electronic cost gain of ASIC time-domain-based trngs. *Cryptogr.*, 5(3):25, 2021.

[PPM+23] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. *ACM Comput. Surv.*, 55(11):227:1–227:35, 2023.

[PWP22] Guilherme Perin, Lichao Wu, and Stjepan Picek. Exploring feature selection scenarios for deep learning-based side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):828–861, 2022.

[Seg55] Beniamino Segre. Curve razionali normali ek-archi negli spazi finiti. *Annali di Matematica Pura ed Applicata*, 39:357–379, 1955.

[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.

[Sta18] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, volume 11389 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2018.

[TWO13] Michael Tunstall, Carolyn Whitnall, and Elisabeth Oswald. Masking tables - an underestimated security risk. 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 425–444. Springer, 2013.

[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 Schramm and Paar's scheme

Gadget 6 presents Schramm and Paar's TBM scheme considering m-bit to m_o-bit lookup table with any values of m, m_o and d. It takes shares $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d] \in \mathbb{F}_{2^m}$ as inputs and computes the output shares $\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d] \in \mathbb{F}_{2^{m_o}}$, such that $S(\sum \hat{\boldsymbol{x}}) = \sum \hat{\boldsymbol{y}}$.

Gadget 6 Schramm and Paar's scheme [SP06] (only secure with $d = 1$)

Input: Shares $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d] \in \mathbb{F}_{2^m}, \ldots, \mathbb{F}_{2^m}$.
Output: Shares $\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d] \in \mathbb{F}_{2^{m_o}}, \ldots, \mathbb{F}_{2^{m_o}}$.
It ensures: $\sum \hat{\boldsymbol{y}} = \mathsf{S}(\sum \hat{\boldsymbol{x}})$ with S an m-bit to m_o-bit lookup table.

——————Pre-processing——————

Generate a vector of random variables $\boldsymbol{s} \in \mathbb{F}_{2^{m_o}}^{d-1}$
$\boldsymbol{t}[i] \leftarrow \mathsf{S}(i) \oplus \sum \boldsymbol{s}$ for $i \in \{0, \ldots, 2^m - 1\}$
for $k = 0$; $k < d$; $i{+}{+}$ **do**
    $\boldsymbol{t}[i] \leftarrow \boldsymbol{t}[i \oplus \hat{\boldsymbol{x}}[k]]$ for any $i \in \{0, \ldots, 2^m - 1\}$
        ▷ It ensures $\mathsf{S}(i \oplus \sum \hat{\boldsymbol{x}}[0\!:\!k]) = \boldsymbol{t}[k] \oplus \sum \boldsymbol{s}$
end for

——————Online-computation——————

$\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d-1] \leftarrow \boldsymbol{s}[0], \ldots, \boldsymbol{s}[d-1]$
$\hat{\boldsymbol{y}}[d] \leftarrow \boldsymbol{t}[\hat{\boldsymbol{x}}[d]]$

B Coron's scheme

Gadget 6 presents Coron's TBM scheme considering m-bit to m_o-bit lookup table with any values of m, m_o and d.

Gadget 7 Coron's scheme (not quite compatible with pre-processing)

Input: Shares $\hat{\boldsymbol{x}}[0], \ldots, \hat{\boldsymbol{x}}[d] \in \mathbb{F}_{2^m}, \ldots, \mathbb{F}_{2^m}$.
Output: Shares $\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d] \in \mathbb{F}_{2^{m_o}}, \ldots, \mathbb{F}_{2^{m_o}}$.
It ensures: $\sum \hat{\boldsymbol{y}} = \mathsf{S}(\sum \hat{\boldsymbol{x}})$ with S an m-bit to m_o-bit lookup table.

——————Pre-processing——————

$\boldsymbol{T}[i, *] \leftarrow (\mathsf{S}(i), \underbrace{0, \ldots, 0}_{d})$ for $i \in \{0, \ldots, 2^m - 1\}$
        ▷ It ensures $\mathsf{S}(i) = \sum \boldsymbol{T}[i, :]$ for $k \in \{0, \ldots, 2^m - 1\}$
for $k = 0$; $k < d$; $k{+}{+}$ **do**
    $\boldsymbol{T}[i, *] \leftarrow \mathsf{Refresh}\big(\boldsymbol{T}[i \oplus \hat{\boldsymbol{x}}[k], *]\big)$ for any $i \in \{0, \ldots, 2^m - 1\}$
        ▷ It ensures $\mathsf{S}(i \oplus \sum \hat{\boldsymbol{x}}[0\!:\!k]) = \sum \boldsymbol{T}[i, *]$
end for

——————Online-computation——————

$\hat{\boldsymbol{y}}[0], \ldots, \hat{\boldsymbol{y}}[d] \leftarrow \mathsf{Refresh}(\boldsymbol{T}[\hat{\boldsymbol{x}}[d], *])$