Secret-Shared Shuffle with Malicious Security

Xiangfu Song*, Dong Yin‡, Jianli Bai§, Changyu Dong†,✉, Ee-Chien Chang*
*National University of Singapore, †Guangzhou University, ‡Ant Group, §University of Auckland
Email: {songxf, changec}@comp.nus.edu.sg, dybean1994@gmail.com, jbai795@aucklanduni.ac.nz, changyu.dong@gmail.com

***Abstract*—A secret-shared shuffle (SSS) protocol permutes a secret-shared vector using a random secret permutation. It has found numerous applications, however, it is also an expensive operation and often a performance bottleneck. Chase et al. (Asiacrypt'20) recently proposed a highly efficient semi-honest two-party SSS protocol known as the CGP protocol. It utilizes purposely designed pseudorandom correlations that facilitate a communication-efficient online shuffle phase. That said, semi-honest security is insufficient in many real-world application scenarios since shuffle is usually used for highly sensitive applications. Considering this, recent works (CANS'21, NDSS'22) attempted to enhance the CGP protocol with malicious security over authenticated secret sharings. However, we find that these attempts are flawed, and malicious adversaries can still learn private information via malicious deviations. This is demonstrated with concrete attacks proposed in this paper. Then the question is how to fill the gap and design a maliciously secure CGP shuffle protocol. We answer this question by introducing a set of lightweight correlation checks and a leakage reduction mechanism. Then we apply our techniques with authenticated secret sharings to achieve malicious security. Notably, our protocol, while increasing security, is also efficient. In the two-party setting, experiment results show that our maliciously secure protocol introduces an acceptable overhead compared to its semi-honest version and is more efficient than the state-of-the-art maliciously secure SSS protocol from the MP-SPDZ library.**

I. Introduction

Secret-shared shuffle (SSS) protocols [1–3] allow mutually distrustful parties to randomly permute a secret-shared vector. Typically, SSS protocols should ensure *privacy* and *correctnesss*. Roughly, privacy ensures hiding the shared secrets and the permutation being used, and correctness ensures a correct shuffling and integrity of the secrets being shuffled. Secret-shared shuffle is a fundamental primitive in secure multiparty computation (MPC). It has found numerous applications in recent days, and mainstream MPC frameworks [4–7] implement SSS as a core functionality. We discuss the following example applications using SSS and refer to §A for concrete use cases.

- Collaborative data analysis [8–15]. In many such applications, data are contributed by mutually distrustful parties who want to pool their data together for knowledge discovery. To protect privacy, data is secret-shared and MPC protocols are employed to perform the analysis using the shares. While secret-sharing can protect the confidentiality of the data, it cannot prevent leakage from access patterns. To fully protect privacy, it is often required that certain operations (*e.g.*, filtering, selection, matching) be performed without revealing which items in the underlying dataset have been affected. A common approach thus is to employ SSS to shuffle the dataset first before doing the actual analysis.
- Anonymous communication [2, 16]. End-to-end encrypted communication has become widespread. However, communication patterns (*i.e.*, metadata) can still break the anonymity of the communication parties. Many anonymous communication systems adopt the Mixnet-based approach [17] using verifiable shuffle, which is computationally expensive due to the involved zero-knowledge proofs to achieve public verifiability [18, 19]. Recently, there has been another line of anonymous communication systems [2, 16, 20] based on SSS in a *distributed-trust* setting. In these systems, there are multiple servers, each holding a secret share of the data shared by senders, and the servers collaboratively shuffle the data and conduct integrity checks. These systems achieve much better performance in general, with the price of introducing a *non-collusion* assumption: verifiability is not public, but if the number of corrupted servers is below a threshold, then any party can still be convinced of correct shuffling when the protocol ends without abort. The non-collusion assumption allows more efficient design and is assumed by many metadata-hiding systems [2, 16, 21, 22], among which [22] are already deployed by Mozilla [23].
- Shuffle model of differential privacy [24–27]: When collecting data from end users, local differential privacy is often used such that the users locally add noise to their data before submitting. However, the noise added significantly limits the accuracy of computations. Recently, it has been demonstrated that adding a shuffler that randomly permutes the messages can amplify differential privacy guarantees so that the amount of noise added locally can be decreased significantly. The so-called shuffle model is currently one of the most active research topics in differential privacy. While most of the shuffle-DP protocols assume a black-box shuffler, some works (*e.g.*, [26]) are specially designed on top of secret sharing, which makes SSS more suitable.

The state-of-the-art two-party SSS protocol was proposed by Chase *et al.* [1], known as the CGP protocol. The CGP protocol can be divided into two phases: an offline correlation generation phase and an online shuffle phase. In the offline phase, the parties collaboratively generate correlations called *shuffle tuples*. Shuffle tuples facilitate a highly efficient online shuffle phase. Due to its simplicity and good efficiency, the CGP protocol and its variants were recently explored in many privacy-preserving applications [2, 3, 9, 12, 28, 29].

The original CGP protocol is secure in the presence of semi-honest adversaries, which preserves privacy (correctness is trivial to achieve in the semi-honest setting). Ensuring privacy and correctness in the presence of malicious adversaries is necessary for many scenarios because shuffling is usually used in sensitive applications, *e.g.*, private health data analysis and anonymous communication. In view of this, recent works [2, 3]

This is the full version of a paper with the same title appearing at NDSS'24
✉ Corresponding author

attempted to design maliciously secure CGP-like shuffle protocols. In particular, the maliciously secure SSS protocols proposed by Eskandarian and Boneh [2] serve as the core protocols in their anonymous communication system Clarion. However, we find that existing maliciously secure CGP-like SSS protocols failed to achieve their claimed security. To show this, we propose concrete attacks to break privacy claims of existing works, revealing the data being shuffled and/or partial information about the underlying permutations. In particular, our attacks to [2], if successful, reveal "who sends to whom" in Clarion. These attacks highlight that designing maliciously secure CGP-like SSS protocols is not as trivial as expected.

Apart from security analysis, our goal is to design a maliciously secure SSS protocol for authenticated secret sharings, using CGP as a starting point. Roughly, our strategy to obtain malicious security with low overhead is the following: We first propose a set of lightweight correlation checks. These correlation checks ensure the well-formness of generated correlations, which defeats attacks that exploit incorrect correlations. However, these checks are subject to selective failure attacks, which leak information about the underlying permutation(s) in successful attacks. Then our second step is to design a leakage-reduction mechanism to remove possible leakage. Our leakage reduction mechanism explores properties both from permutations and the involved selective failure attacks, and we formally analyze its effectiveness using a new cut-and-choose analysis method, which is of independent interest. Finally, we combine all the techniques with authenticated secret sharing to design a maliciously secure secret-shared shuffle protocol. We also exploit nice properties of the involved correlations and protocol components to optimize overall efficiency concretely.

In the presence of a malicious adversary who can behave arbitrarily, our SSS protocol ensures privacy in any case and correctness if the protocol completes (*i.e.*, *malicious security with abort*). We focus on the two-party setting and show how to extend it to the multi-party setting using a pair-execution paradigm from previous works [2, 3]. In the multiparty case, the adversary can corrupt all but one party, and security guarantees hold as long as there is one honest party.

We have implemented our protocol and evaluated its performance. The correlation checks introduce a low overhead compared to the correlation generation protocols with semi-honest security. In particular, the running time is only doubled, and there is about 20% additional overhead in communication. Compared with existing maliciously secure SSS protocol from MP-SPDZ library [4], our protocol is about $15\times$ faster in the offline phase and $7\times$ faster in the online phase.

Contribution. We summarize our contribution as follows:

- We study the (in)security of existing maliciously secure CGP-like SSS protocols [2, 3]. Our proposed attacks show a better understanding of existing CGP-like SSS protocols on security, demonstrating that designing maliciously secure CGP-like SSS protocols is highly non-trivial.
- We design lightweight correlation checks and a leakage reduction mechanism for designing maliciously secure CGP SSS protocols, along with a new cut-and-choose leakage reduction analysis method. Our techniques exploit nice properties from underlying primitives to improve efficiency and security, some of them may not limited to this paper.
- We implement our protocol and compare the concrete efficiency with existing protocols. The introduced overhead for malicious security is acceptable and our protocol is more efficient than existing maliciously secure SSS protocols.

II. Preliminary & Background

A. Notations

We use κ and λ to denote computational and statistical security parameters, respectively, and $\mathsf{negl}(\cdot)$ to denote a negligible function. We use $\mathbb{G}$ to denote an abelian group and $\mathbb{F}$ to denote a finite field (*e.g.*, $\mathbb{F} = \mathbb{F}_{p^k}$ for a prime p) with elements of ℓ bits. We use $[n]$ to denote the set $\{0, 1, ..., n-1\}$ and $[l, r]$ to denote $\{l, l+1, ..., r-1, r\}$. Given a set $\mathcal{X}$, $x \xleftarrow{\$} \mathcal{X}$ denotes that x is uniformally sampled from $\mathcal{X}$. We use $a||b$ to denote strings concatenation of a and b. For an h-bits string $b \in \{0,1\}^h$, we use b_i to denote its i-th bit and $b = b_1||b_2||\cdots||b_h$. We use $\vec{x}$ to denote a vector, and $\vec{x}_i$ or $\vec{x}[i]$ interchangeably to denote its i-th element. We use bold font $\mathbf{M}$ to represent a matrix and denote its element at the i-th row and the j-th column as $\mathbf{M}_{i,j}$ or $\mathbf{M}[i, j]$.

B. Permutation

A permutation is a bijective function $\pi : [n] \mapsto [n]$. We denote $\mathbf{S}_n$ as the symmetric group containing all $[n] \mapsto [n]$ permutations. We denote π^{-1} as the inverse of a permutation π, and $\pi \circ \rho$ as the composition of two permutations π and ρ such that $\pi \circ \rho(i) = \pi(\rho(i))$ for $i \in [n]$. When applying π over a vector $\vec{x}$ of n elements, we have

$$\vec{y} = \pi(\vec{x}) = (\vec{x}_{\pi(0)}, \cdots, \vec{x}_{\pi(n-1)}), \tag{1}$$

where $\vec{y}_i = \vec{x}_{\pi(i)}$, or equivalently, $\vec{x}_i = \vec{y}_{\pi^{-1}(i)}$, for $i \in [n]$.

C. Puncturable Pseudorandom Functions

The CGP protocol relies on puncturable pseudorandom functions to generate correlations. A puncturable pseudorandom function (PPRF) $F : \mathcal{K} \times \mathcal{X} \mapsto \mathcal{Y}$ is a special pseudorandom function (PRF) such that given a PPRF master key $K \in \mathcal{K}$ and a punctured index α, an evaluator who receives a punctured key can evaluate over $\mathcal{X}$ except α, and $F(K, \alpha)$ is pseudorandom to the evaluator. Def. 1 defines PPRF formally.

Definition 1 (Puncturable pseudorandom functions [30])**.** *A puncturable pseudorandom function (PPRF) with master key space $\mathcal{K}$, domain $\mathcal{X}$, and range $\mathcal{Y}$, is a pseudorandom function F with an additional punctured key space $\mathcal{K}_p$ and three probabilistic polynomial-time (PPT) algorithms* $(\mathsf{KeyGen}, \mathsf{Puncture}, \mathsf{Eval})$ *such that:*

- $F.\mathsf{KeyGen}(1^\kappa)$*: Output a random key* $K \xleftarrow{\$} \mathcal{K}$.
- $F.\mathsf{Puncture}(K, \alpha)$*: On input a key* $K \in \mathcal{K}$*, and a punctured index* $\alpha \in \mathcal{X}$*, output a punctured key* $K\{\alpha\} \in \mathcal{K}_p$*; we use* K^* *and* $K\{\alpha\}$ *interchangeably when the context is clear.*
- $F.\mathsf{Eval}(K\{\alpha\}, x)$*: On input a punctured key* $K\{\alpha\}$*, and an index* x*, output* $F(K, x)$ *if* $x \neq \alpha$*, and* $\perp$ *otherwise.*

Security of PPRF requires that a punctured key holder who has $(\alpha, K\{\alpha\})$ cannot distinguish $F(K, \alpha)$ from a random element $y \xleftarrow{\$} \mathcal{Y}$. Formally, we have Def. 2 for PPRF security.

Definition 2 (PPRF selective security [30]). *A PPRF is selectively secure if for any PPT adversary $\mathcal{A}$, and any $\alpha \in \mathcal{X}$ choosen by $\mathcal{A}$ such that*

$$\left| \mathsf{Pr}\left[\begin{array}{l} K \leftarrow F.\mathsf{KeyGen}(1^\kappa), \\ K\{\alpha\} \leftarrow F.\mathsf{Puncture}(K,\alpha), : \mathcal{A}(1^\kappa, \alpha, K\{\alpha\}, y) = 1 \\ y \xleftarrow{\$} \mathcal{Y}. \end{array}\right] - \mathsf{Pr}\left[\begin{array}{l} K \leftarrow F.\mathsf{KeyGen}(1^\kappa), \\ K\{\alpha\} \leftarrow F.\mathsf{Puncture}(K,\alpha), : \mathcal{A}(1^\kappa, \alpha, K\{\alpha\}, y) = 1 \\ y \leftarrow F.\mathsf{Eval}(K,\alpha). \end{array}\right] \right|$$

is negligible in κ.

PPRF from GGM tree. A PPRF $F : \{0,1\}^\kappa \times \{0,1\}^h \mapsto \{0,1\}^\kappa$ can be constructed from the seminal GGM tree [1, 30–32] using a length-doubling pseudorandom generator (PRG) $G : \{0,1\}^\kappa \mapsto \{0,1\}^{2\kappa}$ as follows:

- $F.\mathsf{KeyGen}(1^\lambda)$: Sample a random key $K \xleftarrow{\$} \{0,1\}^\kappa$, *i.e.*, an initial seed for G.
- $F.\mathsf{Puncture}(K,\alpha)$: On input a key K and a punctured index $\alpha = \alpha_1||\alpha_2||\cdots||\alpha_h \in \{0,1\}^h$, set $K^{(0)} \leftarrow K$. For $i \in [1,h]$, compute $(K_0^{(i)}, K_1^{(i)}) \leftarrow G(K^{(i-1)})$, and set $K^{(i)} \leftarrow K_{\alpha_i}^{(i)}$. Return $K\{\alpha\} \leftarrow \{K_{1-\alpha_1}^{(1)}, \cdots, K_{1-\alpha_h}^{(h)}\}$.
- $F.\mathsf{Eval}(K\{\alpha\}, x)$: On input a punctured key $K\{\alpha\}$ and an index $x = x_1||x_2||\cdots||x_h \in \{0,1\}^h$, output $\perp$ if $\alpha = x$. Otherwise, parse $\{K_{1-\alpha_1}^{(1)}, \cdots, K_{1-\alpha_h}^{(h)}\} \leftarrow K\{\alpha\}$, find the key $K_{1-\alpha_i}^{(i)}$ such that $1-\alpha_i = x_i$, and set $K^{(i)} \leftarrow K_{1-\alpha_i}^{(i)}$. For $j \in [i+1,h]$, compute $(K_0^{(j)}, K_1^{(j)}) \leftarrow G(K^{(j-1)})$, and set $K^{(j)} \leftarrow K_{x_j}^{(j)}$. Output $K_{x_h}^{(h)}$.

PPRF with key verification. When using PPRF in maliciously secure protocols, a master key holder may provide a non-wellformed key. In this case, it's necessary to check whether the key is well-formed. This is formalized as a key verification property in Def. 3 by [30]. Looking ahead, GGM PPRF supports key verification.

Definition 3 (Verification of PPRF keys [30]). *Let $F = (\mathsf{KeyGen}, \mathsf{Puncture}, \mathsf{Eval})$ be a PPRF with master key space $\mathcal{K}$, domain $\mathcal{X}$, and range $\mathcal{Y}$. We say that F allows verification of malicious keys for $\widetilde{\mathcal{K}}$, the malicious keyspace, if there exist efficient algorithms $(\mathsf{Ver}, \mathsf{Puncture}^*, \mathsf{Eval}^*)$, such that:*

- Ver *takes as input a malicious key $\widetilde{K} \in \widetilde{\mathcal{K}}$ and a set $I \subseteq \mathcal{X}$ and outputs $0/1$.*
- $\mathsf{Puncture}^*$ *takes as input a malicious key $\widetilde{K}$ and an index $\alpha \in \mathcal{X}$ and outputs a key $\widetilde{K}\{\alpha\}$ punctured at α.*
- Eval^* *takes as input a malicious key $\widetilde{K}$, a set $I \subseteq \mathcal{X}$ and an index $x \in \mathcal{X}$, and outputs a value in $\mathcal{Y}$ or $\perp$.*

Further, we require for all $I \subseteq \mathcal{X}$ and $\widetilde{K} \in \widetilde{\mathcal{K}}$: if $\mathsf{Ver}(\widetilde{K}, I) = 1$ then $\mathsf{Eval}(\widetilde{K}\{\alpha\}, x) = \mathsf{Eval}^*(\widetilde{K}, I, x)$ for all $\alpha \in I$, $x \in \mathcal{X} \setminus \{\alpha\}$, where $\widetilde{K}\{\alpha\} \leftarrow \mathsf{Puncture}^*(\widetilde{K}, \alpha)$. If this holds then we say $\widetilde{K}$ is consistent with the set I.

D. Authenticated Secret Sharing

Linear secret sharing. We use $[\![x]\!]$ to denote an additive linear secret sharing (LSS) for $x \in \mathbb{F}$ shared between k parties. Each P_i holds a random share $[\![x]\!]_i \in \mathbb{F}$ such that $\sum_{i\in[k]} [\![x]\!]_i = x$. The secret x can be constructed iff all the parties reveal their shares and then sum them up, which means this scheme preserves perfect privacy against $k-1$ corrupted parties. LSS supports the following local linear operations.

- $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$: P_i computes $[\![z]\!]_i \leftarrow [\![x]\!]_i + [\![y]\!]_i$.
- $[\![z]\!] \leftarrow c \cdot [\![x]\!]$: P_i computes $[\![z]\!]_i \leftarrow c \cdot [\![x]\!]_i$.
- $[\![z]\!] \leftarrow c + [\![x]\!]$: P_0 computes $[\![z]\!]_0 \leftarrow c + [\![x]\!]_0$ and P_i computes $[\![z]\!]_i \leftarrow [\![x]\!]_i$ for all $i \in [k] \setminus \{0\}$.

We can verify that $[\![x+y]\!] = [\![x]\!] + [\![y]\!]$, $[\![c \cdot x]\!] = c \cdot [\![x]\!]$, and $[\![c + x]\!] = c + [\![x]\!]$.

Authenticated secret sharing. Authenticated secret sharing (ASS) ensures the integrity of shared secrets. A typical SPDZ-like ASS [33–35] relies on *information-theoretic message authentication codes* (IT-MACs) for integrity. Concretely, the parties additionally share $[\![\xi]\!]$ for a secret MAC key $\xi \xleftarrow{\$} \mathbb{F}$. For a sharing $[\![x]\!]$, the parties additionally share its MAC sharing $[\![\gamma(x)]\!]$ such that $\gamma(x) = \xi \cdot x$. We call $\langle x \rangle = ([\![x]\!], [\![\gamma(x)]\!])$ as an *authenticated secret sharing* for a secret x and $\langle x \rangle_i = ([\![x]\!]_i, [\![\gamma(x)]\!]_i) \in \mathbb{F}^2$ as an *authenticated share* held by P_i. Since the soundness error is proportional to the inverse of the field size, we require $\mathbb{F}$ to be sufficiently large (*i.e.*, $|\mathbb{F}| > 2^\lambda$); this is crucial to detect errors with overwhelming probability. ASS supports the following local computation:

- $\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$: $\langle z \rangle \leftarrow ([\![x]\!] + [\![y]\!], [\![\gamma(x)]\!] + [\![\gamma(y)]\!])$.
- $\langle z \rangle \leftarrow c \cdot \langle x \rangle$: $\langle z \rangle \leftarrow (c \cdot [\![x]\!], c \cdot [\![\gamma(x)]\!])$.
- $\langle z \rangle \leftarrow c + \langle x \rangle$: $\langle z \rangle \leftarrow (c + [\![x]\!], c \cdot [\![\xi]\!] + [\![\gamma(x)]\!])$.

We can verify that $\langle x+y \rangle = \langle x \rangle + \langle y \rangle$, $\langle c \cdot x \rangle = c \cdot \langle x \rangle$, and $\langle c + x \rangle = c + \langle x \rangle$.

The above definitions for LSS and ASS generally extend to vectors. We use $[\![\vec{x}]\!]$ to denote a vector sharing of $\vec{x}$, and $\gamma(\vec{x})$ to denote its MAC vector sharing where $\gamma(\vec{x}_i) = \xi \cdot \vec{x}_i$.

Other operations. The SPDZ-family protocols support a variety of useful operations: 1) generating a sharing for a random secret, 2) sharing a secret known by one party, and 3) generating multiplication triples, *etc.* All the above commands are supported by an ideal SPDZ functionality, which can be securely realized from existing protocols [34, 36, 37]. We will use the corresponding ideal functionality directly whenever these operations are required. For all these functionalities, we refer to existing SPDZ protoocls [34, 36, 37].

E. Malicious SSS: Protocol Setting and Security Goals

A malicious SSS protocol involves k parties who jointly share an authenticated vector sharing $\langle \vec{x} \rangle = ([\![\vec{x}]\!], [\![\gamma(\vec{x})]\!])$ where $\vec{x}, \gamma(\vec{x}) \in \mathbb{F}^n$. The parties want to compute an authenticated shared vector $\langle \vec{y} \rangle$ (with re-randomization) such that $\vec{y} = \pi(\vec{x})$, using a random permutation π that neither party knows. We focus on $k = 2$ in this paper and show how to extend our techniques to $k > 2$. We assume the adversary, who behaves arbitrarily, can corrupt at most $k-1$ parties at the beginning of the protocol (*i.e.*, dishonest majority setting with static corruption). In this setting, our malicious SSS protocol preserves privacy and correctness defined as follows:

- *Privacy*: ensuring secrecy of the secret values being shuffled and the permutation being used.
- *Correctness*: ensuing integrity of secrets being shuffled and a correct shuffling, if the protocol completes.

F. Security Definition

We follow the simulation-based security model [38] with *static corruption* and *malicious security with abort*. Security goals are formally captured by an ideal functionality $\mathcal{F}$, a trusted entity that receives inputs from the parties, performs computation, and sends output to the parties. In the real world, an adversary $\mathcal{A}$ on behalf of corrupted parties runs the protocol with the honest parties. A simulator $\mathcal{S}$ interacts with $\mathcal{F}$ in the ideal world. Let $\mathcal{C}$ be the set of corrupted parties. We use $\mathbf{Real}_{\Pi,\mathcal{A}(z),\mathcal{C}}(1^\kappa, 1^\lambda, \{x_i \mid i \notin \mathcal{C}\})$ to denote the joint output of honest parties and $\mathcal{A}$, where x_i is the input from P_i and z is the auxiliary input of $\mathcal{A}$. $\mathbf{Ideal}_{\Pi,\mathcal{S}(z),\mathcal{C}}(1^\kappa, 1^\lambda, \{x_i \mid i \notin \mathcal{C}\})$ denotes the outputs of honest parties and the simulator in the ideal world execution with $\mathcal{F}$.

Definition 4 (Adapted from [39])**.** *A protocol Π securely computes functionality $\mathcal{F}$ in the presence of a malicious adversary if for every PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that*

$$\mathbf{Real}_{\Pi,\mathcal{A}(z),\mathcal{C}}(1^\kappa, 1^\lambda, \{x_i \mid i \notin \mathcal{C}\}) \stackrel{c}{\equiv} \mathbf{Ideal}_{\Pi,\mathcal{S}(z),\mathcal{C}}(1^\kappa, 1^\lambda, \{x_i \mid i \notin \mathcal{C}\}).$$

We say that Π securely computes $\mathcal{F}$ with statistical error $2^{-\lambda}$ if there exists a negligible function $\mathsf{negl}(\cdot)$ such that the distinguishing probability between outputs of the real and ideal world is less than $2^{-\lambda} + \mathsf{negl}(\kappa)$[1].

III. Maliciously Secure SSS Protocols & Attacks

In this section, we first revisit the semi-honest CGP protocol [1]. Then we present our attacks that break the privacy claim of existing malicious CGP-like shuffle protocols [2, 3].

A. The Semi-honest CGP Protocol

We review correlations used in the CGP protocol [1] and how to use them for two-party SSS.

Oblivious punctured vector (OPV). OPV is the basic correlation in the CGP protocol. In an OPV correlation, a sender owns a vector $\vec{v} \in \mathbb{G}^n$ and a receiver holds $(\alpha, \vec{w}) \in [n] \times \mathbb{G}^n$ such that $\vec{w}_i = \vec{v}_i$ for $i \in [n] \setminus \{\alpha\}$ and $\vec{w}_\alpha = \perp$. In this overview, from now on we will assume P_0 is the receiver and P_1 is the sender.

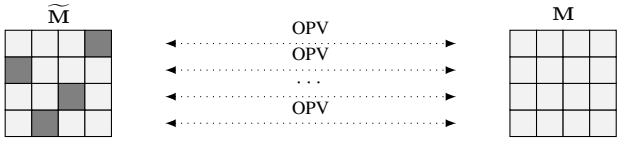

Fig. 1: An example $n \times n$ OPM correlation. Each row forms an OPV with the punctured location colored in black. The punctured location is $\pi(i)$ for i-th row where $\pi \in \mathbf{S}_n$.

Oblivious punctured matrix (OPM). An n-dimention OPM consists of n OPVs with n punctured indexes organized according to a permutation $\pi \in \mathbf{S}_n$. As shown in Fig. 1, a sender holds a matrix $\mathbf{M} \in \mathbb{G}^{n\times n}$, while the receiver holds a permutation $\pi \in \mathbf{S}_n$ and a punctured matrix $\widetilde{\mathbf{M}} \in \mathbb{G}^{n\times n}$. For $i \in [n]$, the i-th row of $\mathbf{M}$ and $\widetilde{\mathbf{M}}$ forms an OPV correlation punctured at $\pi(i)$. We use $\langle\!|\pi|\!\rangle = ((\pi, \widetilde{\mathbf{M}}), (\mathbf{M})) \in (\mathbf{S}_n \times \mathbb{G}^{n\times n}) \times \mathbb{G}^{n\times n}$ to denote an OPM correlation.

Shuffle tuple. In a shuffle tuple $(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times \mathbb{G}^n) \times (\mathbb{G}^n \times \mathbb{G}^n)$, a sender holds two vectors $\vec{a}, \vec{b} \in \mathbb{G}^n$, and a receiver holds a permutation $\pi \in \mathbf{S}_n$ plus an n-dimension vector $\vec{\Delta} \in \mathbb{G}^n$, where $\vec{\Delta} = \pi(\vec{a}) - \vec{b}$. A shuffle tuple can be non-interactively converted from an OPM correlation $\langle\!|\pi|\!\rangle = ((\pi, \widetilde{\mathbf{M}}), (\mathbf{M}))$. Specifically, the parties compute $(\vec{a}, \vec{b})$ and $\vec{\Delta}$, respectively, according to equation (2).

$$\vec{a}_i \leftarrow \sum_j \mathbf{M}_{j,i},\ \vec{b}_i \leftarrow \sum_j \mathbf{M}_{i,j},\ \vec{\Delta}_i \leftarrow \sum_{j\neq i} \widetilde{\mathbf{M}}_{j,\pi(i)} - \sum_{j\neq\pi(i)} \widetilde{\mathbf{M}}_{i,j} \quad (2)$$

One can verify $\vec{\Delta} = \pi(\vec{a}) - \vec{b}$ as required.

Secret-shared shuffle from shuffle tuples. The CGP protocol computes the following ideal SSS functionality $\mathcal{F}_{\mathsf{sss}}$ with semi-honest security: On inputting a secret-shared vector $[\![\vec{x}]\!]$, sample a random $\pi \in \mathbf{S}_n$, and output $[\![\vec{y}]\!]$ where $\vec{y} = \pi(\vec{x})$. $\mathcal{F}_{\mathsf{sss}}$ ensures neither party learns $\vec{x}$ or π.

This can be done with the help of shuffle tuples. Suppose P_0 and P_1 generate a shuffle tuple $(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times \mathbb{G}^n) \times (\mathbb{G}^n \times \mathbb{G}^n)$ with $\mathbb{G} = \mathbb{F}$, where P_0 holds $(\pi, \vec{\Delta})$, and P_1 has $(\vec{a}, \vec{b})$. The parties simply run the protocol as follows: P_1 sends $\vec{\delta} \leftarrow [\![\vec{x}]\!]_1 - \vec{a}$ to P_0. P_0 sets $[\![\vec{y}]\!]_0 \leftarrow \pi([\![\vec{x}]\!]_0 + \vec{\delta}) + \vec{\Delta}$ and P_1 sets $[\![\vec{y}]\!]_1 \leftarrow \vec{b}$. Clearly, $[\![\vec{y}]\!]_0 + [\![\vec{y}]\!]_1 = \pi([\![\vec{x}]\!]_0 + [\![\vec{x}]\!]_1 - \vec{a}) + \pi(\vec{a}) - \vec{b} + \vec{b} = \pi(\vec{x})$. We call the above protocol *one-sided* secret-shared shuffle as P_0 knows the underlying permutation. The parties repeat the above process using another tuple with the role reversed to achieve *fully* secret-shared shuffle, as required by $\mathcal{F}_{\mathsf{sss}}$.

Remark. We call $(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times \mathbb{G}^n) \times (\mathbb{G}^n \times \mathbb{G}^n)$ a *shuffle tuple over* $\mathbb{G}^n$. We stress that $\mathbb{G}$ can be d-ary, *i.e.*, $\mathbb{G} = \mathbb{F}^d$. In this case, the above shuffle tuple can be used to permute a secret-shared vector of tuples defined over $(\mathbb{F}^d)^n$. The d sub-elements in each $\mathbb{F}^d$ element will be moved together when being permuted.

B. Maliciously Secure Protocols & Online-phase Attacks

We review the online phase of existing maliciously secure CGP-like SSS protocols [2, 3]. For the sake of overview, we sketch a simplified version of [3] and show a concrete attack to reveal sensitive information about the underlying permutation.

CGP shuffle for ASS. Existing works [2, 3] design malicious-secure CGP-like SSS protocols over *authenticated secret sharings*. In a nutshell, an authenticated share $\langle x\rangle_i = ([\![x]\!]_i, [\![\gamma(x)]\!]_i) \in \mathbb{F} \times \mathbb{F}$ can be packed as an element from $\mathbb{F}^2$. So, to shuffle an authenticated vector sharing $\langle\vec{x}\rangle = ([\![\vec{x}]\!], [\![\gamma(\vec{x})]\!])$, the parties can perform the CGP shuffle using a shuffle tuple $(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times \mathbb{G}^n) \times (\mathbb{G}^n \times \mathbb{G}^n)$ with $\mathbb{G} = \mathbb{F}^2$. From now on, we will use shuffle tuples with the above form for shuffling authenticated secret sharings.

Let us assume shuffle tuples have been generated by the parties in the offline phase. The parties use the tuple to shuffle an authenticated vector sharing $\langle\vec{x}\rangle = ([\![\vec{x}]\!], [\![\gamma(\vec{x})]\!])$. To ensure data integrity and correct shuffling, both [3] and [2] rely on post-execution checks to detect errors at the end of the protocol. The post-execution check methods in [3] and [2], despite slight differences, both rely on the *integrity of the permuted*

[1]Looking ahead, computational security is from computationally secure primitives (*e.g.*, PRG/PPRF). Statistical security comes from our cut-and-choose game and MAC checks, which are secure in the statistical sense.

Protocol Π_{MACCheck}

Parameter: MAC key $[\![\xi]\!]$ shared between the parties.

Protocol: On inputting an authenticated vector sharing $\langle \vec{m} \rangle$ with $\vec{m} \in \mathbb{F}^n$:

1. The parties call $\{c_i\}_{i\in[n]} \leftarrow \mathcal{F}_{\mathsf{coin}}(\mathbb{F}^n)$ and $\langle r \rangle \leftarrow \mathcal{F}_{\mathsf{rand}}(\mathbb{F})$
2. The parties compute $\langle t \rangle \leftarrow \sum_j c_j \cdot \langle m_i \rangle + \langle r \rangle$.
3. The parties open t (not its MAC) and compute $[\![s]\!] \leftarrow [\![\gamma(t)]\!] - t \cdot [\![\xi]\!]$.
4. After all the parites commit to the shares of $[\![s]\!]$ using $\mathcal{F}_{\mathsf{com}}$, each party invoke $\mathcal{F}_{\mathsf{com}}$ to decommit all the shares and open s. If the protocol aborts or $s \neq 0$, return False. Otherwise, return True.

Fig. 2: Post-execution MAC check protocol Π_{MACCheck}. Here $\mathcal{F}_{\mathsf{coin}}$ are used for generating random coins, $\mathcal{F}_{\mathsf{rand}}$ is used for generating a random ASS sharing, and $\mathcal{F}_{\mathsf{com}}$ is an ideal commitment functionality.

shared MAC values to enforce correctness. A concrete post-execution check is a batch MAC check protocol in Fig. 2, which is used by [2] to detect errors. Combining shuffle tuples over $(\mathbb{F}^2)^n$, post-execution check, and authenticated secret sharings, we sketch the protocol Π_{leaky} as follows:

1. A pre-computed shuffle tuple $([\![\pi]\!]) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n)$ is held by the parites. The sender P_1 holds $(\vec{a}, \vec{b})$ and the receiver P_0 holds $(\pi, \vec{\Delta})$.
2. P_1 sends $\vec{\delta} = \langle \vec{x} \rangle_1 - \vec{a}$ to P_0. P_1 sets $\langle \vec{y} \rangle_1 \leftarrow \vec{b}$.
3. P_0 receives $\vec{\delta}$ and sets $\langle \vec{y} \rangle_0 \leftarrow \pi(\langle \vec{x} \rangle_0 + \vec{\delta}) + \vec{\Delta}$.
4. Run post-execution check over $\langle \vec{y} \rangle$ to detect errors.

When both parties behave honestly, we have:

$$\begin{aligned}\langle \vec{y} \rangle_0 + \langle \vec{y} \rangle_1 &= \pi(\langle \vec{x} \rangle_0 + \vec{\delta}) + \vec{\Delta} + \vec{b} \\ &= \pi(\langle \vec{x} \rangle_0 + \langle \vec{x} \rangle_1 - \vec{a}) + \pi(\vec{a}) - \vec{b} + \vec{b} \\ &= \pi(\langle \vec{x} \rangle_0 + \langle \vec{x} \rangle_1) \\ &= (\pi([\![\vec{x}]\!]_0 + [\![\vec{x}]\!]_1), \pi([\![\gamma(\vec{x})]\!]_0 + [\![\gamma(\vec{x})]\!]_1)) \\ &= (\pi(\vec{x}), \pi(\gamma(\vec{x}))).\end{aligned}$$

A key observation from [2, 3] is that even though the shuffle tuple is not authenticated by MAC, it can be used to shuffle authenticated shares because the shares in a shuffle tuple essentially are blinding masks and will be canceled out at the end of step 3. Hence if the parties are honest, this will not affect the correctness of the authenticated shares being shuffled and the post-execution check in step 4.

Online selective failure attack. Now we show our attacks to Π_{leaky}. First, a malicious receiver P_0 cannot perform any feasible attack either to privacy or correctness. Intuitively, δ reveals no information about $\langle \vec{x} \rangle_1$ to P_0 because the mask $\vec{a}$ is hidden from P_0. As for integrity, P_0 cannot perform any attack without being caught. The reason is that if a malicious receiver tampers with the shares locally, it can always be detected by the integrity check performed over the authenticated secret sharing. On the other hand, this is not the case for a malicious sender. A malicious sender is expected to send $\vec{\delta} = \langle \vec{x} \rangle_1 - \vec{a}$ to the receiver, but it may add errors to $\vec{\delta}$. At first glance, the post-execution check will detect the errors so that nothing can go wrong. This is taken for granted in [2, 3], but actually not true. We show a selective failure attack for a malicious sender to learn secret information about the receiver's permutation with non-negligible probability. A malicious sender can do the following deviations in Π_{leaky}:

- Instead of sending the correct message $\vec{\delta} = \langle \vec{x} \rangle_1 - \vec{a}$ to P_0, P_1 samples a vector $\vec{u} \in (\mathbb{F}^2)^n$, where a non-zero element $e \in \mathbb{F}^2$ appears at position q and all other positions are all set to 0. P_1 sends $\vec{\delta} = \langle \vec{x} \rangle_1 - \vec{a} + \vec{u}$ to P_0.
- P_1 guesses $\pi(p) = q$. *Before* running the post-execution check, P_1 generates a vector $\vec{v} \in (\mathbb{F}^2)^n$ such that $\vec{v}$ is all zero except $\vec{v}_p = e$. P_1 sets $\langle \vec{y} \rangle_1 \leftarrow \vec{b} - \vec{v}$.

In the above attack, there will be two cases depending on whether P_1 guesses $q = \pi(p)$ correctly:

- *Case 1* - $\pi(p) = q$: P_1 makes a correct guess thus $\vec{v} = \pi(\vec{u})$ (refer to equation (1) on how permutation is applied over a vector). Now $\langle \vec{y} \rangle_0 + \langle \vec{y} \rangle_1 = \pi(\langle \vec{x} \rangle_0 + \langle \vec{x} \rangle_1 - \vec{a} + \vec{u}) + \pi(\vec{a}) - \vec{b} + \vec{b} - \vec{v} = (\pi(\vec{x}), \pi(\gamma(\vec{x})))$ still holds. In this case, the integrity of shuffled secrets and the shuffled MACs are still preserved. Hence, the check will output True.
- *Case 2* - $\pi(p) \neq q$: P_1 guesses incorrectly. Now we have $\langle \vec{y} \rangle_0 + \langle \vec{y} \rangle_1 = \pi(\langle \vec{x} \rangle_0 + \langle \vec{x} \rangle_1 - \vec{a} + \vec{u}) + \pi(\vec{a}) - \vec{b} + \vec{b} - \vec{v} = (\pi(\vec{x}), \pi(\gamma(\vec{x}))) + \pi(\vec{u}) - \vec{v}$. In this case, the integrity is broken due to the introduced error $\pi(\vec{u}) - \vec{v}$. Hence, the check can detect the error and output False.

The problem comes from case 1 as the adversary learns $\pi(p) = q$ without being caught. This is a selective failure attack with a success probability of $1/n$, which is non-negligible. One can generalize this attack from adding error to one place to multiple places, though the success probability will drop accordingly. We show more details about our online attack to [3] and [2] in Appendix E and F, respectively.

Summary. This attack starts from the *unauthenticated* protocol message $\vec{\delta}$ since the receiver cannot directly verify the legitimacy of $\vec{\delta}$, and $\vec{\delta}$ is further entangled with subsequent computation and check. By manipulating $\vec{\delta}$ and learning the check output, a malicious sender can extract sensitive information about the permutation with non-negligible probability. This attack breaks the privacy but not the correctness of existing maliciously secure SSS protocols [2, 3].

C. Maliciously Secure Protocols & Offline-phase Attacks

This section demonstrates that CGP correlation generation protocols also suffer from malicious attacks. We propose two attacks: OPV attack and OPM attack. Both attacks exploit incorrect correlations to break privacy.

The semi-honest OPV generation. We first revisit how to generate OPV correlations in the semi-honest setting.

The parties can generate OPVs using a PPRF with domain $\mathcal{X} = [n]$. At a high level, P_1 samples a PPRF key K, and runs a protocol with P_0 who specifies a punctured index $\alpha \in [n]$. The protocol terminates with P_0 receiving a punctured key $K\{\alpha\}$. With $K\{\alpha\}$, P_0 locally recovers $n-1$ PRF outputs except for $F(K, \alpha)$. P_1 can compute all n PRF outputs using K. As required, the parties generate an OPV correlation.

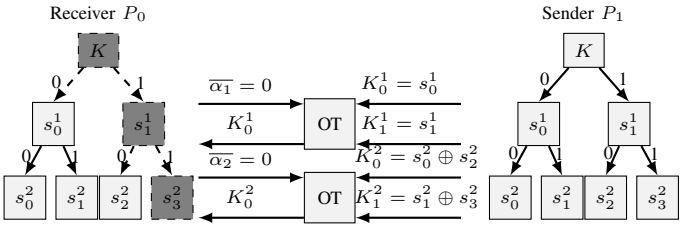

Fig. 3: A 4-dimension OPV generation protocol. P_1 samples a PPRF master key K and P_0 inputs a punctured index $\alpha = 3$ (*i.e.*, $\alpha_1 = 1, \alpha_2 = 1$) and obtains a punctured key $K\{3\} = \{s_0^1, s_2^2\}$ which expends to recover all leaf nodes except s_3^2.

The original CGP protocol [1] proposes a concrete OPV generation protocol using GGM PPRF and oblivious transfer (OT). Fig. 3 sketches the protocol. Specifically, P_1 samples a seed K and expands it to a GGM tree. Then, the parties run a layer-to-layer evaluation. For the i-th layer, P_1 computes two intermediate keys K_0^i and K_1^i, where K_0^i is the XOR of all left-half nodes and K_1^i is the XOR of all right-half nodes in this layer. During the evaluation, the parties maintain an invariant that P_0 holds all but one node in the i-th layer. This means P_0 can recover all but two elements for the next layer using G. To maintain the invariant, P_0 runs OT with P_1 who provides OT messages (K_0^{i+1}, K_1^{i+1}). The receiver chooses one of two values with a choosing bit $\overline{\alpha_{i+1}}$, recovering one of the two missed values, thus maintaining the invariant. At the end of the protocol, the receiver holds a punctured key $K\{\alpha\}$ that can recover all leaves except the one at α.

Remark. The range of the above GGM PPRF is $\{0,1\}^\kappa$. Following existing method [1, 30], we can modify the range to $\mathbb{G} = \mathbb{F}^2$ by applying a conversion function to the leaves; we defer the details to §V-A.

Previous approaches. To generate correlation in the presence of malicious adversaries, [3] simply resorts to an ideal functionality to generate shuffle tuples without giving a concrete instantiation. [2] replaces the semi-honest OTs with maliciously secure OTs, while other parts remain the same as the semi-honest correlation generation protocol. We show that the enhancement from [2] is insufficient for malicious security.

OPV attack. [2] uses maliciously secure OTs to enhance the semi-honest OPV generation protocol. Although using maliciously secure OTs can ensure malicious security for the OT functionality itself, it is insufficient to achieve malicious privacy for the upper-level OPV generation protocol. We show that a malicious sender can mount a selective failure attack via OT message substitution.

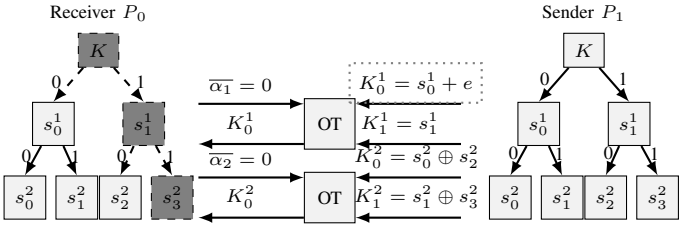

Fig. 4: OT message substitution attack on OPV generation.

Ideally, P_1 is expected to compute all OT messages faithfully using a GGM tree derived from a single seed K, which is not true in malicious cases. We demonstrate this with an example in Fig. 4. The sender P_1 computes a correct GGM tree in the beginning, but it wants to guess α_1 by selectively corrupting one of two OT messages (K_0^1, K_1^1) for the first layer. In particular, the attacker makes a guess $\alpha_1 = 0$ by adding a non-zero error e into K_0^1. There will be two outcomes depending on the true value α_1.

- *Case 1* - $\alpha_1 = 0$: P_1 guessed correctly and the correct K_1^1 is choosen. The parties still share a correct OPV correlation.
- *Case 2* - $\alpha_1 = 1$: P_1 made an incorrect guess and the errored K_1^0 is choosen. The parties share an incorrect OPV.

The malicious sender P_1 does not know whether it guessed correctly or not after the OPV generation, but it will know in the online phase *after* the post-execution integrity check. The OPV generated in the offline phase will be used to construct a shuffle tuple used for the online phase. If the OPV is erroneous (case 2), the tuple containing the OPV will be erroneous too. Consequently, the authenticated sharings, after being shuffled, will be erroneous and will be detected by the post-execution integrity check (refer to §III-B). The problem is that in this example, the adversary can guess correctly with a probability of $1/2$ (case 1). When this happens, the parties still share a valid OPV and the online phase protocol execution will end normally. By learning the protocol does not abort, the adversary learns some information (*i.e.*, α_1) without being caught. Clearly, this is a selective failure attack with a success probability of $1/2$. The malicious sender may perform the above attack over more OT instances in the OPV generation protocol, and the success probability will drop accordingly.

OPM attack. Recall that a correct OPM correlation requires all n punctured indexes to be organized according to a permutation $\pi \in \mathbf{S}_n$. A malicious receiver may organize these indexes following arbitrary strategies. However, previous work [2] does not check the well-formness of OPM correlations, and we show how a malicious receiver P_0 can deviate in the protocol to break privacy.

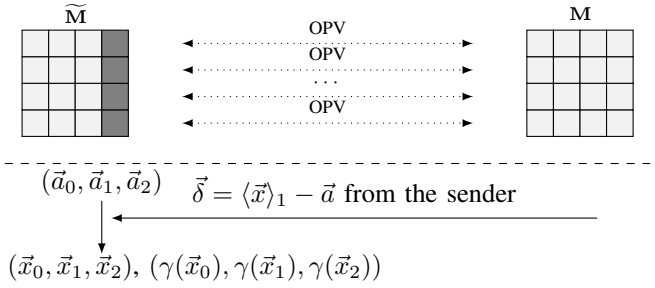

Fig. 5: OPM attack from a malicious receiver.

A malicious P_0 may puncture the same column for more than one row. Fig. 5 shows an example. P_0 always punctures the fourth column and thus can learn $(\vec{a}_0, \vec{a}_1, \vec{a}_2)$ for free. Combined with the message $\vec{\delta} = \langle \vec{x} \rangle_1 - \vec{a}$ in the online phase and the receiver's local share $\langle \vec{x} \rangle_0$, the receiver can fully recover $(\vec{x}_0, \vec{x}_1, \vec{x}_2)$ and their MACs $(\gamma(\vec{x}_0), \gamma(\vec{x}_1), \gamma(\vec{x}_2))$. In this attack, P_0 learns $n-1$ full columns of the OPM matrix,

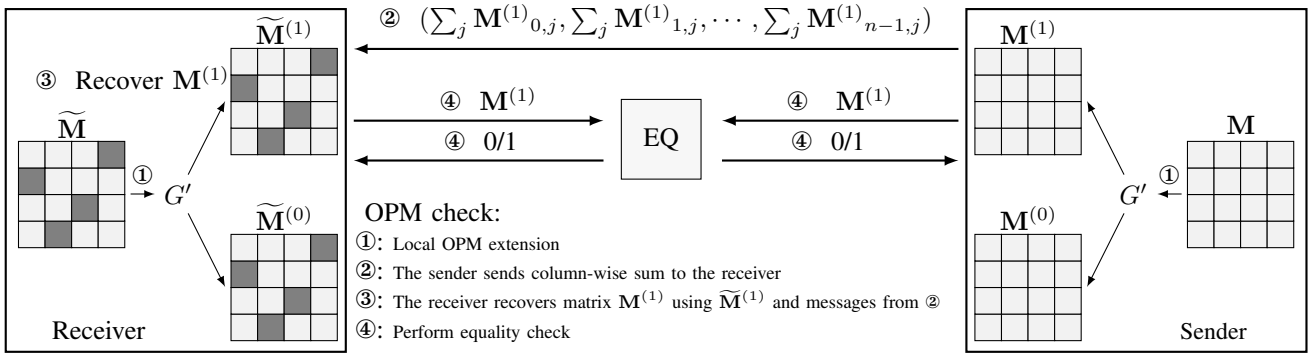

Fig. 6: OPM check from OPM extension and sacrifice

then $n-1$ values of $\vec{a}$. Leaking $\vec{a}$ will reveal corresponding shared secrets and MAC values, breaking privacy.

Summary. The OPV attack is a selective failure attack from a malicious sender, which undermines the secrecy of the permutation used by the receiver. Differently, the OPM attack is from a malicious receiver. It reveals shared secrets and MACs. Above attacks highlight that incorrect correlations can be exploited to break privacy.

IV. Overview of Our Countermeasures

To defend against the aforementioned attacks by the receiver, we use correlation checks to ensure the well-formness of the OPV and OPM correlations. These checks cannot resolve the attacks from the sender. Even worse, the OPM check introduces a new selective failure attack. All selective failure attacks allow a malicious sender to learn sensitive information about the permutation chosen by the receiver with a certain success probability. Hence we further develop a leakage reduction mechanism to tackle the leakages from selective failure attacks so that we can achieve full privacy.

OPV check. To check OPV correctness, we utilize a consistency check trick for GGM trees [30, 40]. The idea is to additionally expand a GGM tree for an additional layer and map the punctured index α to its left children at $\alpha||0$. P_1 sends a hash of all right extended leaf nodes to P_0, from which P_0 can compare with its own value to check consistency. We stress that a malicious P_1 can still perform the OPV (selective failure) attack. The purpose of the OPV check is not to fix the privacy problem but to detect incorrect OPV correlations right after its generation (rather than waiting until the online phase).

OPM check. We propose a lightweight OPM check to check whether the OPM receiver punctures OPM honestly. Since the parties perform OPV checks right after OPV generation, this means that row-wise OPVs are correct if all previous OPV checks pass. Condition on that, it remains to check whether each column also maintains an OPV correlation. If this is true, P_1 is convinced that all n punctured indexes are organized properly according to a permutation.

Check OPM via sacrifice. The challenge here is checking OPM without leaking information about the permutation being used. We propose a sacrifice-based OPM check strategy. At a high level, our technique allows P_0 to fully recover the whole OPM matrix held by P_1 *if and only if* P_0 punctures the OPM honestly. To sacrifice an OPM, P_1 simply sends n column-wise sum (*i.e.*, the vector $\vec{a}$ of the produced tuple) to P_0. Now, if P_0 punctures the OPM honestly, it can recover a full OPM matrix for sure; otherwise, P_0 would have missed two more values for certain columns and cannot recover the whole matrix. Therefore, P_1 can instead check whether P_0 can fully recover the complete matrix. This can be done by each party committing its own matrix first and then revealing to check equality, a common trick dating back to [41].

Check without destroying correlation. However, sacrifice destroys an OPM correlation (as the whole matrix is revealed to the receiver). To enable check without losing correlation, we further propose an *OPM extension* trick. In principle, the parties can generate two OPM correlations embedded with identical punctual information. Thus, the parties can sacrifice one OPM and keep the other for subsequent shuffle operations. As we will show, this OPM extension can be directly achieved by modifying the range of GGM PPRF, using a PRG G'. Combining the extension trick and OPM sacrifice-based check, we design a check for OPM correlations, as sketched in Fig. 6.

New selective failure attack. Unfortunately, our OPM check introduces a new selective failure attack from a malicious sender P_1. In particular, suppose P_1 guesses $q = \pi(p)$. It can selectively add a non-zero error e to the q-th column-wise sum. By definition, the error will be moved to $\widetilde{\mathbf{M}}^{(1)}_{\pi^{-1}(q),q}$ by P_0. P_1 adds e to $\mathbf{M}^{(1)}_{p,q}$ and use the updated $\mathbf{M}^{(1)}_{p,q}$ for equality check. Now, if P_1 guessed $q = \pi(p)$ correctly, the error will be added to the same place thus the equality check completes normally; in this case, P_1 learns $q = \pi(p)$ without being caught. Otherwise, P_1 would be caught. Clearly, this introduces a coordinate-wise selective failure attack, which is of the same type as the online selective failure attack.

The leakage reduction mechanism. We have captured three kinds of selective failure leakage: 1) leakage from offline OPV check; 2) leakage from offline OPM check; and 3) leakage from online post-execution check. To reduce leakage, we first observe an inherent property of selective failure attacks: they can be detected with a probability.

Reduce leakage via repeated execution. We use sufficient tuples to perform repeated execution to reduce the leakage. Specifically, whenever a random shuffle over $\langle \vec{x} \rangle$ is required, the parties will perform B cascaded shuffle execution using B tuples $(\![\pi_0]\!]$, $(\![\pi_1]\!]$, $\cdots$, $(\![\pi_{B-1}]\!]$, and the parties perform post-execution MAC check for each shuffle. In this manner,

the parties share $\langle \vec{y} \rangle = \langle \pi(\vec{x}) \rangle$ where $\pi = \pi_{B-1} \circ \cdots \circ \pi_1 \circ \pi_0$. To learn information about π, the adversary must guess correctly B times and pass all checks. With a large enough B, the probability of the attacks going undetected becomes negligible. As long as one non-attacked permutation exists, the resulting permutation π will still be random in the view of the malicious sender, hence preventing leakage.

Cut-and-choose leakage reduction and analysis. We further design a cut-and-choose bucketing strategy to reduce B for generating a batch of correlations. In particular, given a bucket number N, the parties will generate NB tuples and they randomly assign NB tuples into N buckets, each containing B tuples. Since the number of leaky tuples is limited, it's possible to generate a large number of tuples such that each bucket contains sufficient non-leaky tuples. Each bucket of B tuples will be used in the online phase for leakage reduction.

To analyze B in this game, we propose a combinatorial analysis using *generating functions* [42, 43]. Our method can compute the tightest B given batch size N, tuple dimension n, and statistical security parameter λ. However, combinatorial analysis using generating functions will face a combination explosion for large Ns. We propose a faster polynomial multiplication algorithm and exploit inherent constraints between involved parameters, making this approach practical. Our method may be helpful to other protocols using a similar cut-and-choose game, which is of independent interest.

Put all pieces together. With correlation checks, leakage reduction, and carefully combining them with authenticated secret sharing, we design a maliciously secure secret-shared shuffle protocol. Our protocol provides privacy in any case and correctness if the protocol does not abort. Furthermore, our protocol inherits good properties from the original CGP protocol; thus, existing optimizations can be directly applied to our protocol to improve efficiency. We will provide more details in the following sections.

V. Our Maliciously Secure SSS Protocol

Following the intuition stated in the previous section, we present the concrete protocols in this section.

A. Concrete GGM PPRF Instantiation

As required for shuffling authenticated sharings and for facilitating correlation checks, we use a GGM PPRF:

$$F' : \{0,1\}^\kappa \times \{0,1\}^{h+1} \mapsto (\mathbb{F}^2 \times \{0,1\}^\kappa) \times \{0,1\}^\kappa.$$

F' is obtained from a standard GGM PPRF F with $\mathcal{K} = \{0,1\}^h$, $\mathcal{X} = \{0,1\}^h$ and $\mathcal{Y} = \{0,1\}^\kappa$ using $G : \{0,1\}^\kappa \mapsto \{0,1\}^{2\kappa}$, by simply applying a PRG $G' : \{0,1\}^\kappa \mapsto (\mathbb{F}^2 \times \{0,1\}^\kappa) \times \{0,1\}^\kappa$ to extend one additional layer over the h-th layer of the GGM tree. We will parse each extended leaf node (at level $h+1$) as $((m_0, m_1), \beta) \in (\mathbb{F}^2 \times \{0,1\}^\kappa) \times \{0,1\}^\kappa$, where (m_0, m_1) is the left children and β is the right children. This arrangement facilitates correlation checks and shuffling authenticated sharings: $\beta \in \{0,1\}^\kappa$ will serve the OPV check, $m_1 \in \{0,1\}^\kappa$ will serve the OPM check, and $m_0 \in \mathbb{F}^2$ is kept for generating shuffle tuples if the checks passed.

B. Offline Phase Protocol: OPV Setup with Check

We present OPV setup protocol with up to selective failure attacks in the presence of a malicious sender S.

Functionality $\mathcal{F}_{\mathsf{GenV}}$

Parameters: $n \in \mathbb{N}$; PPRF F' with malicious key space $\widetilde{\mathcal{K}}$.

Gen: Upon receiving α from S:

- Honest S : Sample $K \leftarrow \{0,1\}^\kappa$ and compute $K^* \leftarrow F'.\mathsf{Puncture}(K, \alpha||0)$.
- Corrupted S:
  - Wait for the adversary $\mathcal{A}$ to input a guess set $I \subseteq [n]$ and a key $\widetilde{K} \in \widetilde{\mathcal{K}}$.
  - Check $\alpha \in I$ and $\mathsf{Ver}(\widetilde{K}, I) = 1$. If the check fails, send abort to R and wait for a response. When R responds with abort, forward to S and halt.
  - Compute $K^* \leftarrow F'.\mathsf{Puncture}^*(\widetilde{K}, \alpha||0)$.
- Send (K^*, α) to R. S receives K ($\widetilde{K}$ for a malicious S)

Fig. 7: OPV setup functionality.

Protocol Π_{GenV}

Parameters: $n, h \in \mathbb{N}$ where $n = 2^h$; a hash function $h_1 : \{0,1\}^{n\kappa} \mapsto \{0,1\}^{2\kappa}$; two PRGs $G : \{0,1\}^\kappa \mapsto \{0,1\}^{2\kappa}$ and $G' : \{0,1\}^\kappa \mapsto (\mathbb{F}^2 \times \{0,1\}^\kappa) \times \{0,1\}^\kappa$.

Protocol: R inputs an index $\alpha \in [n]$:

1. S samples a random $K \in \{0,1\}^\kappa$ and defines $s_0^0 \leftarrow K$.
2. For each $i \in [1,h]$, S computes $(s_{2j}^i, s_{2j+1}^i) \leftarrow G(s_j^{i-1})$ for all $j \in [1, 2^{i-1}]$. Then, for each $i \in [1,h]$, S computes $K_0^i \leftarrow \bigoplus_{j \in [2^{i-1}]} s_{2j}^i$ and $K_1^i \leftarrow \bigoplus_{j \in [2^{i-1}]} s_{2j+1}^i$.
3. For $i \in [1,h]$, the parties invoke $\mathcal{F}_{\mathsf{OT}}$ such that in the i-th OT, R inputs a choice bit $\overline{\alpha_i}$, and S inputs (K_0^i, K_1^i). R receives $K_{\overline{\alpha_i}}^i$.
4. R defines $s_{\overline{\alpha_1}}^1 \leftarrow K_{\overline{\alpha_1}}^1$. For $i \in [2,h]$: R computes $(s_{2j}^i, s_{2j+1}^i) \leftarrow G(s_j^{i-1})$ for $j \in [2^{i-1}] \setminus \{\alpha_1 \cdots \alpha_{i-1}\}$; R defines $\alpha_i^* \leftarrow \alpha_1 || \cdots || \alpha_{i-1} || \overline{\alpha_i}$ and computes $s_{\alpha_i^*}^i \leftarrow K_{\overline{\alpha_i}}^i \oplus (\bigoplus_{j \in [2^{i-1}]} s_{2j+\overline{\alpha_i}}^i)$.
5. For $j \in [1,n]$, S computes $(s_{2j}^{h+1}, s_{2j+1}^{h+1}) \leftarrow G'(s_j^h)$. S computes $K_1^{h+1} = \bigoplus_{j \in [n]} s_{2j+1}^{h+1}$, $\tau = h_1(s_1^{h+1}, s_3^{h+1}, \cdots, s_{2n-1}^{h+1})$, and sends (K_1^{h+1}, τ) to R.
6. Let $K\{\alpha||0\} \leftarrow \{K_{\overline{\alpha_i}}^i\}_{i \in [h]} \cup \{K_1^{h+1}\}$. With $K\{\alpha||0\}$, R can compute all right leaf nodes $\{s_{2j+1}^{h+1}\}_{j \in [n]}$ at level $h+1$.
7. R sets $\gamma_j = s_{2j+1}^{h+1}$ for all $j \in [n]$, and computes $\tau' = h_1(\gamma_0, \gamma_1, \cdots, \gamma_{n-1})$. If $\tau = \tau'$, R outputs $(K\{\alpha||0\}, \alpha)$. Otherwise, R aborts.

Fig. 8: OPV setup protocol Π_{GenV}.

Ideal OPV setup functionality. We formalize functionality $\mathcal{F}_{\mathsf{GenV}}$ by adapting the one for PPRF setup from [30], with simplification to suit the OPV setting. $\mathcal{F}_{\mathsf{GenV}}$ receives a PPRF master key K from a sender S and a punctured index α from a sender R, output a punctured key $K\{\alpha||0\}$ to R. $\mathcal{F}_{\mathsf{GenV}}$ ensures malicious security against corrupted R, but $\mathcal{F}_{\mathsf{GenV}}$ is up to selective failure attacks from a corrupted S, which allows the adversary to guess an index range I; this captures the OPV selective failure attack from S, as mentioned in §III-C.

OPV setup with check. Protocol Π_{GenV} in Fig. 8 is for securely computing $\mathcal{F}_{\mathsf{GenV}}$. There are two enhancements compared with its semi-honest version. First, Π_{GenV} uses maliciously secure OTs in the punctured PPRF key generation phase, for which we directly use ideal OT functionality $\mathcal{F}_{\mathsf{OT}}$. Since we use a GGM PPRF $F' : \{0,1\}^\kappa \times \{0,1\}^{h+1} \mapsto (\mathbb{F}^2 \times \{0,1\}^\kappa) \times \{0,1\}^\kappa$ in this paper, R will map the punctured coordinate α to 2α (equivalently, $\alpha||0$). Second, the parties run the OPV check at the end of the protocol. S sends τ, a tag computed by hashing all the right leaves at layer $h+1$. Since R can compute a tag τ' of the right leaves, R can check whether $\tau = \tau'$, and aborts if it is not the case.

Complexity. Communication comes from h OTs of κ-bit messages plus a 2κ-bit tag for OPV check, thus the communication complexity is $O(\kappa \log n)$. Computation-wise, each party invokes $O(n)$ PRG evaluation for GGM tree expansion.

Security. Theorem 1 shows that Π_{GenV} securely computes $\mathcal{F}_{\mathsf{GenV}}$. The proof is in §G.

Theorem 1. *Protocol Π_{GenV} securely computes functionality $\mathcal{F}_{\mathsf{GenV}}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model in the presence of a malicious adversary, assuming G and G' are secure PRGs, and h_1 is a collision-resistance hash function.*

C. Offline Phase Protocol: OPM Setup with Check

Ideal OPM setup functionality. We design a functionality $\mathcal{F}_{\mathsf{GenM}}$ in Fig. 9. $\mathcal{F}_{\mathsf{GenM}}$ ensures malicious security against a corrupted R, but it allows a corrupted S to selectively guess a subset index set per row; this is to capture the selective failure attacks from a malicious sender, as we discussed in §III-C.

Functionality $\mathcal{F}_{\mathsf{GenM}}$

Parameters: $n \in \mathbb{N}$; PPRF F' with malicious key space $\widetilde{\mathcal{K}}$.

Gen: Upon receiving π from R:

- If $\pi \notin \mathbf{S}_n$, abort.
- Sample $K_i \leftarrow \{0,1\}^\kappa$ for $i \in [n]$ and compute $K_i^* \leftarrow F'.\mathsf{Puncture}(K_i, \pi(i)||0)$.
- If S is corrupted:
  - Wait for the adversary to input n sets $I_0, \cdots, I_{n-1} \subseteq [n]$ and a set of keys $\widetilde{K}_0, \cdots, \widetilde{K}_{n-1} \in \widetilde{\mathcal{K}}$.
  - For $i \in [n]$, check $\pi(i) \in I_i$ and $\mathsf{Ver}(\widetilde{K}_i, I_i) = 1$. If any check fails, send abort to R and wait for a response. When R responds with abort, forward to S and halt.
  - $K_i^* \leftarrow F'.\mathsf{Puncture}^*(\widetilde{K}_i, \pi(i)||0)$ for $i \in [n]$.
- Send $(\pi, \{K_i^*\}_{i\in[n]})$ to R. S receives $\{K_i\}_{i\in[n]}$ ($\{\widetilde{K}_i\}_{i\in[n]}$ for malicious S).

Fig. 9: OPM setup functionality.

OPM setup with check. Fig. 10 shows the OPM generation protocol Π_{GenM} with OPM check. The challenge is checking R punctures the OPM according to a valid permutation; this is crucial to ensure privacy as discussed in §III-C.

We propose a sacrifice-based check strategy. In particular, S sends n column-wise sum to R and our protocol checks whether R can recover the exact matrix held by S. We use a common equality check functionality $\mathcal{F}_{\mathsf{eq}}$ dating back to [41]. $\mathcal{F}_{\mathsf{eq}}$ allows the parties to check whether they hold the same value followed by revealing inputs. To reduce the communication, each party hashes its own matrix to generate a short tag. The parties instead use $\mathcal{F}_{\mathsf{eq}}$ over tags to perform equality check. Using a concrete protocol from [41] for $\mathcal{F}_{\mathsf{eq}}$, this check only requires $O(\kappa)$ bits of communication.

PPRF F' supports OPM check without losing correlation. In particular, S can non-interactively parse the matrix OPM $\mathbf{M}$ as two matrices: $\mathbf{M}^{(0)} \in (\mathbb{F}^2)^{n\times n}$ and $\mathbf{M}^{(1)} \in (\{0,1\}^\kappa)^{n\times n}$. Similarly, R obtains the punctured matrices $\widetilde{\mathbf{M}}^{(0)}$ and $\widetilde{\mathbf{M}}^{(1)}$. Note that $\widetilde{\mathbf{M}}^{(0)}$ and $\widetilde{\mathbf{M}}^{(1)}$ are punctured in the same way. The parties can sacrifice $\langle\!|\pi|\!\rangle^{(1)}$ for check and keep $\langle\!|\pi|\!\rangle^{(0)}$ to generate a shuffle tuple over $(\mathbb{F}^2)^n$ for online use.

Protocol Π_{GenM}

Parameters: $n \in \mathbb{N}$; PPRF F'; hash function $H : \{0,1\}^{n^2\kappa} \mapsto \{0,1\}^{2\kappa}$.

Protocol: R inputs $\pi \in \mathbf{S}_n$. S inputs n PPRF keys $\{K_i\}_{i\in[n]}$.

1. For $i \in [n]$, R and S call $\mathcal{F}_{\mathsf{GenV}}$ for n times with respective inputs $\pi(i)$ and $\perp$. R receives $\{\pi(i), K_i^*\}_{i\in[n]}$ and S receives $\{K_i\}_{i\in[n]}$. If any call fails, R receives abort from $\mathcal{F}_{\mathsf{GenV}}$.
2. The parties generate an OPM correlation $\langle\!|\pi|\!\rangle = ((\pi, \widetilde{\mathbf{M}}), (\mathbf{M}))$, where $\mathbf{M} \in (\mathbb{F}^2 \times \{0,1\}^\kappa)^{n\times n}$, as follows:
   a) For $i \in [n]$ and $j \in [n]$ S computes $\mathbf{M}_{i,j} \leftarrow F'.\mathsf{Eval}(K_i, j||0)$.
   b) For $i \in [n]$ and $j \in [n] \setminus \{\pi(i)\}$, R computes $\widetilde{\mathbf{M}}_{i,j} \leftarrow F'.\mathsf{Eval}(K_i^*, j||0)$.
3. The parties locally parse $\langle\!|\pi|\!\rangle = ((\pi, \widetilde{\mathbf{M}}), (\mathbf{M}))$ as two OPMs:
   $$\langle\!|\pi|\!\rangle^{(0)} = ((\pi, \widetilde{\mathbf{M}}^{(0)}), (\mathbf{M}^{(0)})), \langle\!|\pi|\!\rangle^{(1)} = ((\pi, \widetilde{\mathbf{M}}^{(1)}), (\mathbf{M}^{(1)})),$$
   where $\mathbf{M}^{(0)} \in (\mathbb{F}^2)^{n\times n}$ and $\mathbf{M}^{(1)} \in (\{0,1\}^\kappa)^{n\times n}$.
4. The parties sacrifice $\langle\!|\pi|\!\rangle^{(1)}$ for OPM check:
   a) For $j \in [n]$, S computes $\omega_j \leftarrow \bigoplus_{i\in[n]} \mathbf{M}_{i,j}^{(1)}$. S sends $\{\omega_j\}_{j\in[n]}$ to R.
   b) R recovers $\mathbf{M}_{i,\pi(i)}^{(1)} \leftarrow \omega_{\pi(i)} \oplus (\bigoplus_{j\neq\pi(i)} \widetilde{\mathbf{M}}_{i,j}^{(1)})$ for $i \in [n]$. R updates $\widetilde{\mathbf{M}}_{i,\pi(i)}^{(1)} \leftarrow \mathbf{M}_{i,\pi(i)}^{(1)}$ for $i \in [n]$.
   c) S computes $\tau \leftarrow H_1(\mathbf{M}^{(1)}))$. Similarly, R computes $\tilde{\tau} \leftarrow H_1(\widetilde{\mathbf{M}}^{(1)})$. S and R invoke $\mathcal{F}_{\mathsf{eq}}$ with inputs τ and $\tilde{\tau}$, respectively. $\mathcal{F}_{\mathsf{eq}}$ will abort if $\tau \neq \tilde{\tau}$.
5. R outputs $(\pi, \{K_i^*\}_{i\in[n]})$ and S outputs $\{K_i\}_{i\in[n]}$.

Fig. 10: OPM setup protocol Π_{GenM}.

From keys to correlations. Protocol Π_{GenM} outputs a pair of correlated OPM keys after the OPM well-formness check. With the correlated keys, the parties obtain an OPM correlation $\langle\!|\pi|\!\rangle^{(0)} = ((\pi, \widetilde{\mathbf{M}}^{(0)}), (\mathbf{M}^{(0)}))$, and then a shuffle tuple:

$$[\![\pi]\!] = ((\pi, \vec{\Delta}), (\vec{a}, \vec{B})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n).$$

Note that $[\![\pi]\!]$ is defined over $(\mathbb{F}^2)^n$ in order to shuffle authenticated secret sharings. When invoking $\mathcal{F}_{\mathsf{GenM}}$ in the hybrid model, the parties can locally expand their keys to share $\langle\!|\pi|\!\rangle^{(0)}$, from which they define a shuffle tuple $[\![\pi]\!]$ over $(\mathbb{F}^2)^n$. We will assume this local computation when invoking $\mathcal{F}_{\mathsf{GenM}}$ in our maliciously secure SSS protocol.

Complexity. Communication complexity is $O(\kappa n \log n)$ from OPV key generation. Besides, R sends n column-wise sum values, which incurs additional $O(\kappa n)$ communication. The communication complexity is $O(\kappa n \log n + \kappa n)$. In terms of computation, each OPV requires $O(n)$ PRG invocations resulting in $O(n^2)$ computation complexity in total.

Security. Like $\mathcal{F}_{\mathsf{GenV}}$, we capture the selective failure attacks in $\mathcal{F}_{\mathsf{GenM}}$. Π_{GenM} securely computes $\mathcal{F}_{\mathsf{GenM}}$ in Theorem 2. The proof can be found in §H.

Theorem 2. *Protocol Π_{GenM} securely computes functionality $\mathcal{F}_{\mathsf{GenM}}$ in the $(\mathcal{F}_{\mathsf{GenV}}, \mathcal{F}_{\mathsf{eq}})$-hybrid model in the presence of a malicious adversary, assuming H_1 is a random oracle.*

D. Online Phase Protocol

Our online phase protocol is essentially the CGP protocol plus online authenticated sharing checks and leakage reduction. Recall that the offline checks ensure correct correlations, the online check ensures correct shuffling and integrity, and the leakage reduction mechanism deals with selective failure attacks. Together they achieve correctness and full privacy. We focus on the leakage reduction mechanism in this section, for the CGP online protocol with authenticated sharing checks, please refer to the previous section.

Ideal functionality. Our goal is to compute a two-party one-sided secret-shared shuffle functionality $\mathcal{F}_{\mathsf{oss}}$: the parties S and R jointly hold an authenticated secret-shared vector $\langle \vec{x} \rangle$ and R additionally choose a permutation $\pi \in \mathbf{S}_n$. $\mathcal{F}_{\mathsf{oss}}$ computes $\vec{y} = \pi(\vec{x})$ and shares $\langle \vec{y} \rangle$ (with randomization) between the parties. Fig. 11 shows a formal description of $\mathcal{F}_{\mathsf{oss}}$.

Functionality $\mathcal{F}_{\mathsf{oss}}$

Parameters: $n \in \mathbb{N}$; a dictionary Val storing all authenticated secrets.

Inputs: R provides a permutation $\pi \in \mathbf{S}_n$; the identifiers $\{\mathsf{id}_i\}_{i\in[n]}$ for the values being shuffled, and the identifiers $\{\mathsf{id}'_i\}_{i\in[n]}$ for the shuffled values to be stored with.

Functionality: receiving $(\mathsf{oss}, \pi, \{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]})$ from R and $(\mathsf{oss}, \{\mathsf{id}_i\}_{i\in[n]}, \{\mathsf{id}'_i\}_{i\in[n]})$ from S:

- If $\pi \notin \mathbf{S}_n$, abort.
- Define a vector $\vec{x}$ such that $\vec{x}_i = \mathsf{Val}[\mathsf{id}_i]$ for $i \in [n]$.
- Compute $\vec{y} = \pi(\vec{x})$ and store $\mathsf{Val}[\mathsf{id}'_i] \leftarrow \vec{y}_i$ for $i \in [n]$.

Fig. 11: Functionality for one-sided secret-shared shuffle.

Leakage reduction via repeated execution. As mentioned in §IV, we use repeated shuffle to amplify the hardness of a selective failure attack. Specifically, $\langle \vec{x} \rangle$ is shuffled with B cascaded sub-shuffles, i.e. $\vec{y} = \pi_{B-1} \circ \pi_{B-2} \circ \cdots \circ \pi_0(\vec{x})$. The parties run the MAC checks at the end of each sub-shuffle.

Let us assume shuffle tuples are distributed by a trusted party. In this case, a malicious S is restricted to performing the online-phase selective failure attack with success probability at most $1/n$ for each guess. Now S will face a dilemma: on the one hand, if S wants to learn non-trivial information about the composed permutation π, S must guess all immediate coordinates of mapping correctly. The success probability will drop to n^{-B} in this case. By setting $B > \lceil \frac{\lambda}{\log_2 N} \rceil$, we can ensure $n^{-B} < 2^{-\lambda}$, which means S will be caught with probability at least $1 - 2^{-\lambda}$. On the other hand, if S only attacks a small (strict) subset of B permutations, S may luckily not be caught. Nevertheless, the remaining non-attacked permutations can still ensure full privacy. Namely, $\pi = \pi_{B-1} \circ \pi_{B-2} \circ \cdots \circ \pi_0$ is still a random permutation in the view of S. Therefore, the attacker will either be detected with overwhelming probability (in the first case), or S will learn no information (in the second case).

Reducing B via cut-and-choose batch generation. The above analysis only considers online-phase leakages. We extend it to handle leakages uniformly from the *offline* and the *online* phases. Now, the attacker has more advantage in the online-phase attack since the (possible) accumulated offline-phase leakage can facilitate the online-phase attack.

Given statistical security parameter λ, the adversary is allowed to learn at most λ bits of information in the offline phase, corresponding to detecting probability $1-2^{\lambda}$. Condition on that, we want to compute a minimal B such that after running B cascaded shuffles, the composed permutation is still a random permutation in the view of the adversary, except with statistical error $2^{-\lambda}$. It is possible to set $B = \lambda + \lceil \frac{\lambda}{\log_2 n} \rceil$ for our purpose: Since the number of the leaky tuples will be at most λ, we can still ensure at least $\lceil \frac{\lambda}{\log_2 n} \rceil$ non-leaky tuples, as required to counter online selective failure attack.

To reduce B further, we propose a cut-and-choose leakage reduction mechanism in a batch tuple generation setting. Concretely, given a bucket number N, the parties generate NB tuples and randomly assign them into N buckets, each containing B tuples. Since the number of leaky tuples from the offline phase is limited, we want to ensure each bucket contains at least $\lceil \frac{\lambda}{\log_2 n} \rceil$ non-leaky tuples. It is possible to generate sufficient tuples and thereby obtain a smaller bucket size $B < \lambda + \lceil \frac{\lambda}{\log_2 n} \rceil$, such that all the buckets contain at least $\lceil \frac{\lambda}{\log_2 n} \rceil$ non-leaky tuples.

We propose a new combinatorial analysis method based on *generating functions* [42, 43]. Using generating function for combinatorial analysis is a generic method but is usually impractical for complex combinations. We propose a fast computing method and exploit inherent constraints in our analysis to reduce complexity, making this approach practical. Our analysis method may be helpful to other protocols using a similar cut-and-choose game, which is of independent interest. We move details to §B. As a concrete example, we have $B = 7$ when $N = 2^{20}$, $n = 2^{10}$ and $\lambda = 40$.

E. Combine All Together

Two-party OSS protocol. Combining correlation generation with check, authenticated sharings, MAC check, and leakage reduction, we finally propose one-sided protocol $\Pi_{\mathsf{oss}}^{[\mathbb{R}]}$ in Fig. 12. $\Pi_{\mathsf{oss}}^{[\mathbb{R}]}$ consists an online phase and an offline phase.

Offline phase. The offline phase performs cut-and-choose batch tuple generation. Specifically, the parties generate NB tuples by calling $\mathcal{F}_{\mathsf{GenM}}$, where the receiver R specifies NB permutations. Note that $\mathcal{F}_{\mathsf{GenM}}$ generates a pair of correlated OPM keys, from which the parties locally generate a tuple

Protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$

Parameters: N: number of buckets; B: bucket size.

[Offline]: On inputting sharings N and B, do the following:

1. The parties invoke $\mathcal{F}_{\mathsf{GenM}}$ to generated NB OPM instances, where $\mathtt{R}$ plays the role of OPM receiver to specify all NB permutations. The parties locally evaluate NB pairs of OPM keys to share NB shuffle tuples, where each tuple $(\!|\pi|\!)$ is defined as $(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{b})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n)$.
2. $\mathtt{R}$ samples a random seed $seed \xleftarrow{\$} \{0,1\}^\kappa$ and sends $seed$ to $\mathtt{S}$. $\mathtt{R}$ and $\mathtt{S}$ use $seed$ to determine a random permutation $\rho : [NB] \to [NB]$.
3. The parties use ρ to permute NB shuffle tuples, then divide them into N bucket, each containing B tuples.

[Online]: On inputting sharings $\langle \vec{x} \rangle$, do the following:

1. Let $\langle \vec{x}^{(0)} \rangle \leftarrow \langle \vec{x} \rangle$.
2. Fetch B shuffle tuples $(\!|\pi_0|\!), (\!|\pi_1|\!), \cdots, (\!|\pi_{B-1}|\!)$ from the next unused bucket.
3. For $i \in [B]$, do the following:
   a) $\mathtt{S}$ parses $(\!|\pi_i|\!)_{\mathtt{S}} = (\vec{a}^{(i)}, \vec{b}^{(i)})$, sends $\vec{\delta}^{(i)} = \langle \vec{x}^{(i)} \rangle_{\mathtt{S}} - \vec{a}^{(i)}$ to $\mathtt{R}$. $\mathtt{S}$ sets $\langle \vec{x}^{(i+1)} \rangle_{\mathtt{S}} \leftarrow \vec{b}^{(i)}$.
   b) $\mathtt{R}$ receives $\vec{\delta}^{(i)}$ and parse $(\!|\pi_i|\!)_{\mathtt{R}} = (\pi_i, \vec{\Delta}^{(i)})$. $\mathtt{R}$ sets $\langle \vec{x}^{(i+1)} \rangle_{\mathtt{R}} \leftarrow \pi_i(\langle \vec{x}^{(i)} \rangle_{\mathtt{R}} + \vec{\delta}^{(i)}) + \vec{\Delta}^{(i)}$.
4. $\mathtt{S}$ and $\mathtt{R}$ run Π_{MACCheck} over $(\langle \vec{x}^{(1)} \rangle, \langle \vec{x}^{(2)} \rangle, \cdots, \langle \vec{x}^{(B)} \rangle)$. If the check fails, abort.
5. Output $\langle \vec{x}^{(B)} \rangle$ if the protocol does not abort.

Fig. 12: Maliciously secure one-sided shuffle protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$.

$(\!|\pi|\!) = ((\pi, \vec{\Delta}), (\vec{a}, \vec{B})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n)$. The order of generated tuples is shuffled using a public permutation $\rho \in \mathbf{S}_{NB}$ chosen by the receiver, which is determined by a κ-bits random seed. The parties divide NB permutated tuples into N buckets, each containing B tuples. Each bucket of B tuples will be used to perform OSS shuffle in the online phase.

Online phase. The parties use tuples generated from the offline phase to perform OSS shuffle for an authenticated vector sharing $\langle \vec{x} \rangle$ and outputs a vector sharing $\langle \pi(\vec{x}) \rangle$. Specifically, the parties fetch B tuples from the next unused bucket, perform B cascaded CGP shuffles, and run MAC check for all intermediate shared vectors $(\langle \vec{x}^{(1)} \rangle, \langle \vec{x}^{(2)} \rangle, \cdots, \langle \vec{x}^{(B)} \rangle)$. If the check passes, the parties output $\langle \vec{x}^{(B)} \rangle$ as the final output.

Correctness. When both parties behave honestly, we can verify that $\langle \vec{x}^{(i+1)} \rangle_{\mathtt{R}} + \langle \vec{x}^{(i+1)} \rangle_{\mathtt{S}} = \pi_i(\langle \vec{x}^{(i)} \rangle_{\mathtt{R}} + \vec{\delta}^{(i)}) + \vec{\Delta}^{(i)} + \vec{b}^{(i)} = \pi_i(\langle \vec{x}^{(i)} \rangle_{\mathtt{R}} + \langle \vec{x}^{(i)} \rangle_{\mathtt{S}} - \vec{a}^{(i)}) + \pi_i(\vec{a}^{(i)}) - \vec{b}^{(i)} + \vec{b}^{(i)} = \pi_i(\langle \vec{x}^{(i)} \rangle_{\mathtt{R}} + \langle \vec{x}^{(i)} \rangle_{\mathtt{S}}) = (\pi_i([\![\vec{x}^{(i)}]\!]_{\mathtt{R}} + [\![\vec{x}^{(i)}]\!]_{\mathtt{S}}), \pi_i([\![\gamma(\vec{x}^{(i)})]\!]_{\mathtt{R}} + [\![\gamma(\vec{x}^{(i)})]\!]_{\mathtt{S}})) = (\pi_i(\vec{x}^{(i)}), \pi_i(\gamma(\vec{x}^{(i)})))$, which is a correct shuffle of secrets $\vec{x}^{(i)}$ and the MACs $\gamma(\vec{x}^{(i)})$ using π_i. By a simple induction argument over $i \in [B]$, we have $\vec{x}^{(B)} = \pi(\vec{x})$ where $\pi = \pi_{B-1} \circ \cdots \circ \pi_1 \circ \pi_0$.

Security. Secuirty of protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ is formalized as Theorem 3, with a proof in §I.

Theorem 3. *$\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ securely computes $\mathcal{F}_{\mathsf{oss}}$ in the $\mathcal{F}_{\mathsf{GenM}}$-hybrid model with statistical security error $2^{-\lambda}$, in the presence of malicious adversaries, under the condition that $|\mathbb{F}| > 2^\lambda$ and proper setting of B as specified in Table V.*

SSS protocol. A two-party SSS protocol can be obtained by invoking $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ twice using the reversed execution strategy (see §III-A). This is straightforward and we provide such a protocol Π_{sss} in Fig. 13. The offline phase simply invokes the offline phase of $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ twice, and each party takes turns playing the role of the receiver $\mathtt{R}$. In the online phase, the parties run the OSS online protocol where each party plays the role of the receiver. Security of Π_{sss} directly follows from $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ (since Π_{sss} does nothing more than simply invoking $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$ twice).

Protocol Π_{sss}

Parameters: N: number of buckets; B: bucket size.

[Offline]: On inputting sharings N and B, do the following:

1. For $i \in [2]$, the parties run $\Pi_{\mathsf{oss}}^{[P_i]}.\mathsf{Offline}(N, B)$, where P_i specifices NB permutations.

[Online]: On inputting sharings $\langle \vec{x} \rangle$, do the following:

1. The parties run $\langle \vec{y} \rangle \leftarrow \Pi_{\mathsf{oss}}^{[P_0]}.\mathsf{Online}(\langle \vec{x} \rangle)$.
2. The parties run $\langle \vec{z} \rangle \leftarrow \Pi_{\mathsf{oss}}^{[P_1]}.\mathsf{Online}(\langle \vec{y} \rangle)$.
3. Ourput $\langle \vec{z} \rangle$.

Fig. 13: Two-party maliciously secure SSS protocol.

F. Extension and Optimizations

Multi-party shuffle protocols. While this paper focuses on the two-party setting, we can extend our two-party protocols to the multi-party setting. Using the pair-wise execution paradigm from existing works [2, 3], we provide k-party OSS protocol $\Pi_{k\text{-oss}}$ and k-party SSS protocol $\Pi_{k\text{-sss}}$ in §C. We note that $\Pi_{k\text{-sss}}$ requires k^2 invocations of pair-wise CGP shuffle, incurring $O(\ell k^2 n)$ online communication and $O(\kappa k^2 n \log n)$ offline communication. It remains to design maliciously secure multi-party CGP shuffle with a better complexity for large k.

Trade-off using generalized Benes networks. Π_{oss} inherits the low-communication property of the CGP protocol, but its computation complexity is proportional to the dimension of shuffled vector squared, incurring huge computational overhead when shuffling large-dimension vectors. We borrow a permutation decomposition technique from [1] to balance computation and communication. This decomposition trick utilizes a generalized Benes network [44], which we call GBN decomposition. GBN decomposition decomposes a permutation $\pi \in \mathbf{S}_n$ into $d = 2\lceil \log n / \log T \rceil - 1$ permutations $\pi_1 \circ \pi_2 \circ \cdots \circ \pi_d$, and each $\pi_i \in \mathbf{S}_n$ is composed by n/T disjoint smaller permutations from $\mathbf{S}_T$. Since each $T \times T$ OPM requires only $O(T^2)$ computational cost, the offline computational cost will be $O(dnT)$. Overall, the smaller T is, the less computational overhead is required. However, running dn/T small shufflings increases online communication costs. By configuring T, we can achieve a trade-off between computation and communication. We use GBN decomposition in our implementation to achieve trade-offs between communication and computation. We discuss more details on GBN decomposition and how to combine GBN decomposition with our SSS protocols in §D.

VI. Implementation and Performance

A. Experiment Settings

We implement our protocol in C++. We use PRG and hash functions designed from fix-key AES and IKNP-type maliciously secure OT extension from emp-tookit (https://github.com/emp-toolkit). We run the experiment on a desktop PC equipped with Intel(R) Xeon(R) Platinum 8163 CPU at 2.50GHz × 16 running Ubuntu 20.04 LTS and 32 GB of memory. We use Linux tc command to simulate local-area network (LAN, RTT: 0.2 ms, 1 Gbps) and wide-area network (WAN, RTT: 80 ms, 40 Mbps). We set the computational security parameter $\kappa = 128$, statistical security parameter $\lambda = 40$, and element size $\ell = 128$ bits.

B. Performance of Correlation Generation

We first report the performance of our correlation generation protocol in terms of running time and communication. Performance is measured for generating one shuffle tuple, where tuple dimension n is taken from $\{2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\}$.

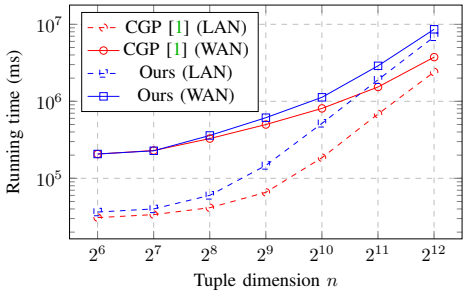

Fig. 14: Running time (ms) for single tuple generation.

Running time. Fig. 14 compares the running time of our protocol with [1] on correlation generation under LAN and WAN settings, respectively. Our protocol is only about 1.1-2.9× (resp. 1.01-2.3×) slower than the semi-honest CGP protocol [1] in LAN (resp. WAN) setting. The extra overhead comes from the use of malicious OTs and the correlation check in correlation generation. We note all OTs execution can be done in a batch in one round. We use similar low-round tricks in other parts of implementation to reduce the impact of a lower network. Notably, our protocol is only 1.3-5.7× slower in WAN compared with the LAN setting. Also, as n increases, computation will dominate the running time, and communication contributes less to the overall running time.

Protocol	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
[1]	0.031	0.056	0.111	0.234	0.504	1.094	2.372
Ours	0.037	0.066	0.119	0.246	0.525	1.131	2.442

TABLE I: Communication (MB) for generating a shuffle tuple

Communication. Our protocol requires maliciously secure OTs and correlation checks. The maliciously secure IKNP OT extension introduces little communication overhead. For correlation checks, we design in a communication-efficient fashion. As a result, our correlation checks introduce low communication overhead. As shown in Table I, our protocol only incurs about 20% more communication than [1].

C. Performance of Shuffle

We report the performance of our protocol and compare it with the state-of-the-art maliciously secure SSS protocol in MP-SPDZ library [4].

Settings. We take n from $\{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$ and use GBN decomposition in our implementation. We use $T = 2^d$ to perform shuffle where $d \in \{4, 6, 8, 10\}$. Performance is reported in the two-party setting. Both our protocol and the protocol from [4] can be divided into offline correlation generation phase and online shuffle phase. In the offline phase, the parties generate sufficient correlations to facilitate online shuffle. The difference is that our protocol utilizes shuffle tuples while [4] uses multiplications triples to evaluate a two-swap permutation network.

Protocol	LAN					
	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Our (2^4)	0.08	0.15	0.72	2.54	13.79	49.90
Our (2^6)	0.03	0.14	0.52	1.68	6.39	38.85
Our (2^8)	0.02	0.07	0.18	0.72	6.20	22.05
Our (2^{10})	**0.01**	**0.07**	**0.15**	**0.69**	**2.43**	**9.81**
[4]	0.05	0.18	0.73	3.24	17.90	81.89
Protocol	WAN					
	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Our (2^4)	1.34	4.77	9.65	27.26	119.34	456.34
Our (2^6)	0.92	2.83	6.35	16.30	53.31	282.39
Our (2^8)	0.73	2.29	3.79	9.14	43.46	165.67
Our (2^{10})	**0.57**	**2.02**	**3.29**	**7.91**	**23.26**	**84.86**
[4]	2.49	3.70	7.98	26.30	119.04	522.38

TABLE II: Online running time (s)

Running time. We report the running time for the online phase in Table II. All candidate protocols outperform the one from MP-SPDZ both in LAN and WAN settings. We can see that T is a tunable parameter. The larger T is, the less running time is required for shuffling the same dimension of vectors. In particular, the online running time of our protocol is around the same level as [4] when $T = 2^4$, and the efficiency gap is enlarged for larger Ts. For example, in the WAN setting, our protocol, taking $T = 2^{10}$, is 4× faster than the SSS protocol from [4] for $n = 2^{10}$ but is 6× faster for $n = 2^{20}$, showing that our protocol scales well for shuffling large-dimension vectors.

Table III reports the running time for offline correlation generation. Since we use cut-and-choose leakage reduction for our protocol, the running time is computed in an amortized sense. In the LAN setting where computation is the bottleneck, it is better to use low-dimension tuples to reduce offline running time. However, when changing to the WAN setting, communication matters more to overall offline running time. On average, setting $T = 2^6$ achieves a most balanced efficiency over different ns in the WAN setting, compared to $T = 2^4$ in the LAN setting. When $T = 2^6$, the offline running time of our protocol is about 15× faster than the protocol from MP-SPDZ (using HE-based preprocessing).

Communication. Table IV reports communication overhead for offline and online phases. We run both our protocol and MP-SPDZ shuffle protocol to perform shuffles over the different dimensions of vectors. MP-SPDZ supports either OT or HE for preprocessing, thus we report each, respectively.

Protocol	LAN						WAN					
	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Ours (2^4)	**0.36**	**1.22**	**6.29**	**24.78**	**132.43**	**507.21**	6.56	12.35	44.51	155.06	761.93	**3,030.09**
Ours (2^6)	0.48	1.78	11.11	44.70	178.15	986.47	4.89	**9.92**	**42.88**	147.25	**571.71**	3,213.94
Ours (2^8)	1.30	5.40	20.41	79.67	553.30	2,126.90	5.39	13.37	83.60	**145.72**	938.52	4,005.16
Ours (2^{10})	1.68	18.44	76.84	273.84	1,125.21	4,578.25	**4.33**	26.49	91.04	345.30	1,365.92	5,853.09
[4] (OT)	9.53	45.27	211.41	1,077.76	4,209.90	-	125.21	593.10	2,769.98	12,706.00	-	-
[4] (HE)	4.73	17.81	79.36	357.64	1,610.24	-	59.26	90.71	427.87	1,978.11	9,056.06	-

TABLE III: Amortized offline running time (s)

Protocol	Offline						Online					
	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Ours (2^4)	7.03	27.8	155.17	620.34	3,189.88	12,759.18	1.06	4.26	23.86	95.42	490.73	1,962.93
Ours (2^6)	4.76	**18.76**	124.65	498.16	1,992.38	11,156.93	0.49	1.97	13.11	52.43	209.72	1,174.41
Ours (2^8)	4.98	19.73	**78.71**	**314.64**	2,097.22	8,388.68	0.39	1.57	6.29	25.17	167.77	671.09
Ours (2^{10})	**1.84**	21.39	85.39	341.38	**1,365.31**	**5,461.04**	**0.11**	**1.38**	**5.51**	**22.02**	**88.08**	**352.32**
[4] (OT)	1,757.43	8,561.07	40,193.90	184,763.00	835,816.00	-	1.37	6.65	31.33	144.18	652.22	2,910.86
[4] (HE)	123.89	497.72	1,115.25	5,050.48	22,832.60	-	1.37	6.65	31.33	144.18	652.22	2,910.86

TABLE IV: Amortized communication (MB) for offline and online phases

Our protocol requires much less communication than [4]. The gap between communication consumption is particularly large for the offline phase: even compared with our protocol using $T = 2^4$, HE-based MP-SPDZ preprocessing requires at least $7\times$ more communication, and OT-based preprocessing requires two orders of magnitude more communication.

Our protocol utilizes GBN decomposition to achieve flexible trade-offs when configured with different Ts. For online communication, our protocol requires the least communication when $T = 2^{10}$. This is clear since using large-dimension shuffle tuples can reduce online communication.

Towards offline communication, setting $T = 2^{10}$ only sometimes achieves the best communication among all candidates of our protocol. As we can see, setting $T = 2^6$, $T = 2^8$ and $T = 2^8$ result in the least communication over $n = 2^{12}$, 2^{14}, and 2^{16}, respectively. We stress this is because $d = 10$ does not divide n for these cases: the parties use tuples of dimension 2^{10} to shuffle the middle layers of the Benes network with less than 10 layer, which essentially makes a waste. As such, we recommend choosing d that divides $\log_2 n$ to optimize offline communication.

VII. Related Work

Secret-shared shuffle protocols are essential in many secure computation tasks, including oblivious sorting [45–48], private set intersection/union [9, 49–51] and private function evaluation [52, 53], anonymous communication [2, 16], oblivious RAM/database [8, 54], and private data analysis [10, 12, 55].

There are two classical approaches for designing secret-shared shuffle protocols: homomorphic encryption based [18, 49] and MPC-based on permutation network [49, 52, 53]. Protocols constructed on homomorphic encryption can achieve asymptotically optimal linear communication complexity, but the computation overhead is relatively high. In order to achieve malicious security, each party has to use zero-knowledge proof to prove it follows the shuffle protocol honestly, which can be expensive for vectors with large dimensions. The approach based on permutation networks relies on MPC techniques to compute atomic swaps of a permutation network, in which re-randomization is done to ensure the obliviousness of the swap. Using OT extension techniques, protocols from the second approach only require a constant number of public-key operations plus cheap symmetric operations. Their communication overhead is proportional to $\ell n \log n$, where ℓ is the bit length of elements and n is the dimension of the vector to be shuffled. The overhead will be increased for malicious security.

Other SSS protocols [8, 10, 51, 56] designed in the three-party honest-majority setting. We note that techniques under the honest-majority assumption cannot trivially apply to the dishonest-majority settings.

VIII. Conclusion & Discussion

This paper proposes a maliciously secure secret-shared shuffle protocol. We first show existing constructions are insecure by proposing concrete attacks. We then design lightweight correlation check and leakage reduction mechanisms and carefully combine them with authenticated secret sharing to achieve malicious security. The experiments show that our protocol introduces acceptable overhead and is more efficient than the state-of-the-art.

Limitations & future work. Our work has limitations and leaves future work that requires further investigation:

- *More efficient leakage reduction mechanisms*. While our protocol can generate correct correlations with low overhead, the cost of leakage reduction for the online phase is still relatively high. It is worth designing an efficient mechanism to directly check the legitimacy of $\vec{\delta}$ instead of running a post-execution check.
- *More efficient multi-party SSS protocols*. Although we can extend our protocol to multi-party settings, each party must run a one-sided shuffle with all the other parties, requiring k^2 invocations of pair-wise shuffle. It remains to design maliciously secure CGP protocols with better complexity.

Acknowledgement

We thank anonymous reviewers for their insightful comments and suggestions.

This research is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative and the National Natural Science Foundation of China under Grant 62072132, Grant 62261160651, and Grant 62302118. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] M. Chase, E. Ghosh, and O. Poburinnaya, "Secret-shared shuffle," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2020, pp. 342–372.

[2] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," in *NDSS*, 2022.

[3] P. Laud, "Linear-time oblivious permutations for spdz," in *International Conference on Cryptology and Network Security*. Springer, 2021, pp. 245–252.

[4] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1575–1590.

[5] "Secretflow," https://github.com/secretflow/secretflow, accessed: 2023-11-17.

[6] "Private computation framework library from meta," https://github.com/facebookresearch/fbpcf/tree/main/fbpcf/mpc_std_lib/shuffler, accessed: 2023-11-17.

[7] "Petace (release 0.1.0)," https://github.com/tiktok-privacy-innovation/PETAce, 2023, tikTok Pte. Ltd.

[8] P. Mohassel, P. Rindal, and M. Rosulek, "Fast database joins and psi for secret shared data," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1271–1287.

[9] Y. Jia, S. Sun, H. Zhou, J. Du, and D. Gu, "Shuffle-based private set union: Faster and more secure," in *USENIX Security 2022*, 2022, pp. 2947–2964.

[10] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 610–629.

[11] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, "Efficient secure three-party sorting with applications to data analysis and heavy hitters," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 125–138.

[12] E. Anderson, M. Chase, F. B. Durak, E. Ghosh, K. Laine, and C. Weng, "Aggregate measurement via oblivious shuffling," *Cryptology ePrint Archive*, 2021.

[13] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root oram: efficient random access in multi-party computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 218–234.

[14] Q. Luo, Y. Wang, Z. Ren, K. Yi, K. Chen, and X. Wang, "Secure machine learning over relational data," *arXiv preprint arXiv:2109.14806*, 2021.

[15] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 19–38.

[16] D. Lu and A. Kate, "Rpm: Robust anonymity at scale," *Cryptology ePrint Archive*, 2022.

[17] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–88, 1981.

[18] C. A. Neff, "A verifiable secret shuffle and its application to e-voting," in *Proceedings of the 8th ACM conference on Computer and Communications Security*, 2001, pp. 116–125.

[19] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer, 2012, pp. 263–280.

[20] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, "Mcmix: Anonymous messaging via secure multiparty computation." in *USENIX security symposium*, 2017, pp. 1217–1234.

[21] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS Symposium 2020*, 2020.

[22] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics." in *NSDI*, 2017, pp. 259–282.

[23] J. Aas and T. Geoghegan, "Introducing isrg prio services for privacy respecting metrics," https://www.abetterinternet.org/post/introducing-prio-services/, accessed: 2023-06-01.

[24] Ú. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta, "Amplification by shuffling: From local to central differential privacy via anonymity," in *SODA*, T. M. Chan, Ed., 2019, pp. 2468–2479.

[25] A. Cheu, A. D. Smith, J. R. Ullman, D. Zeber, and M. Zhilyaev, "Distributed differential privacy via shuffling," in *EUROCRYPT*, 2019, pp. 375–403.

[26] B. Balle, J. Bell, A. Gascón, and K. Nissim, "Private summation in the multi-message shuffle model," in *ACM CCS '20*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds., 2020, pp. 657–676.

[27] A. Cheu and M. Zhilyaev, "Differentially private histograms in the shuffle model from fake users," in *IEEE SP*, 2022, pp. 440–457.

[28] N. Attrapadung, G. Hanaoaka, T. Matsuda, H. Morita, K. Ohara, J. C. Schuldt, T. Teruya, and K. Tozawa, "Oblivious linear group actions and applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 630–650.

[29] P. Laud, "Efficient permutation protocol for mpc in the head," in *International Workshop on Security and Trust Management*. Springer, 2021, pp. 62–80.

[30] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round ot extension and silent non-interactive secure computation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 291–308.

[31] O. Goldreich, S. Goldwasser, and S. Micali, "How to con-

struct random functions," *Journal of the ACM (JACM)*, vol. 33, no. 4, pp. 792–807, 1986.

[32] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias, "Delegatable pseudorandom functions and applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 669–684.

[33] I. Damgård and C. Orlandi, "Multiparty computation for dishonest majority: From passive to active security at low cost," in *Annual cryptology conference*. Springer, 2010, pp. 558–576.

[34] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 1–18.

[35] M. Keller, E. Orsini, and P. Scholl, "Mascot: faster malicious arithmetic secure computation with oblivious transfer," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 830–842.

[36] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[37] M. Keller, E. Orsini, and P. Scholl, "Actively secure ot extension with optimal overhead," in *Annual Cryptology Conference*. Springer, 2015, pp. 724–741.

[38] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of CRYPTOLOGY*, vol. 13, pp. 143–202, 2000.

[39] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2017, pp. 225–255.

[40] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated ot with small communication," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1607–1626.

[41] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *Annual Cryptology Conference*. Springer, 2012, pp. 681–700.

[42] P. Doubilet, G.-C. Rota, and R. Stanley, "On the foundations of combinatorial theory. vi. the idea of generating function," in *Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 2: Probability Theory*. University of california press, 1972, pp. 267–318.

[43] R. L. Graham, D. E. Knuth, O. Patashnik, and S. Liu, "Concrete mathematics: a foundation for computer science," *Computers in Physics*, vol. 3, no. 5, pp. 106–107, 1989.

[44] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *Bell system technical journal*, vol. 43, no. 4, pp. 1641–1656, 1964.

[45] D. Bogdanov, S. Laur, and R. Talviste, "A practical analysis of oblivious sorting algorithms for secure multiparty computation," in *Nordic Conference on Secure IT Systems*. Springer, 2014, pp. 59–74.

[46] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, "Practically efficient multi-party sorting protocols from comparison sort algorithms," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 202–216.

[47] S. Laur, J. Willemson, and B. Zhang, "Round-efficient oblivious database manipulation," in *International Conference on Information Security*. Springer, 2011, pp. 262–277.

[48] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket oblivious sort: An extremely simple oblivious sort," in *Symposium on Simplicity in Algorithms*. SIAM, 2020, pp. 8–14.

[49] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.

[50] G. Garimella, P. Mohassel, M. Rosulek, S. Sadeghian, and J. Singh, "Private set operations from oblivious switching," in *IACR International Conference on Public-Key Cryptography*. Springer, 2021, pp. 591–617.

[51] P. H. Le, S. Ranellucci, and S. D. Gordon, "Two-party private set intersection with an untrusted third party," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2403–2420.

[52] P. Mohassel and S. Sadeghian, "How to hide circuits in mpc an efficient framework for private function evaluation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 557–574.

[53] P. Mohassel, S. Sadeghian, and N. P. Smart, "Actively secure private function evaluation," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2014, pp. 486–505.

[54] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The melbourne shuffle: Improving oblivious storage in the cloud," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2014, pp. 556–567.

[55] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon, "Secure parallel computation on national scale volumes of data," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2487–2504.

[56] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, "An efficient secure three-party sorting protocol with an honest majority," *Cryptology ePrint Archive*, 2019.

[57] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung, "Two-sided malicious security for private intersection-sum with cardinality," in *Annual International Cryptology Conference*. Springer, 2020, pp. 3–33.

[58] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2020.

[59] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority mpc for malicious adversaries—breaking the 1 billion-gate per second barrier," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 843–862.

APPENDIX

A. Concrete Use Cases Using SSS

Collaborative data analysis [8–15]. SSS protocols are useful for reducing/eliminating leakage from access patterns. We provide an example in Fig. 15. Suppose two computing parties jointly share a table `T` with three attributes `C1, C2, C3`, where `C1` is a binary attribute denoting whether a person has diabetes. In the data collection phase, though the sensitive attribute values are hidden from the parties using secret sharing, the computing parties know who contributes to each record. When the parties want to execute an SQL query `SELECT (C2, C3) FROM T WHERE C1 = 1` to select out the shared `(C2, C3)` values of records satisfying `C1=1` for certain private data analysis. The parties cannot reveal column `C1` in plaintext because this approach reveals whether a person has diabetes. The parties can jointly perform a row-wise secret-shared shuffle, then reveal column `C1` in clear to make the selection. The second approach hides which two out of four have diabetes.

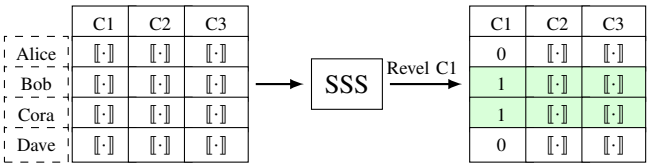

Fig. 15: Record filtering using SSS. $[\![\cdot]\!]$ denotes a shared secret.

SSS for secure sorting [10, 11, 45]. Assume two service providers (SPs) want to run a secure sorting protocol over their joint dataset (*e.g.*, computing top-k elements), where the initial inputs are shared between two SPs from many data contributors. If using a non-oblivious sorting algorithm (*e.g.*, quick sort), even if the comparison is securely computed, the adversary can still learn sensitive information by simply observing access patterns. A common approach is evaluating an oblivious sorting algorithm over the shared data with an access pattern that is independent of the secrets to be sorted. Existing oblivious sorting protocol with practical efficiency (*e.g.*, bitonic sort) requires $O(n \log^2 n)$ secure comparisons. Several works [10, 11, 45] exploit shuffling to secure sorting protocols to improve efficiency, known as the *shuffle-then-compute* paradigm. The idea is that the parties perform SSS over the input secrets to break the relation between the shuffled secrets and the original inputs, then run a non-oblivious sort algorithm with $O(n \log n)$ comparison (*e.g.*, quick sort) over the shuffled inputs. In these works, an efficient shuffle protocol is vital: it's meaningless to use the shuffle-then-compute paradigm if the overhead introduced from shuffling is larger than the overhead reduction from using non-oblivious sorting.

Private attribution reporting. Google released a private intersection-sum (PSI-SUM) protocol with applications to attributing aggregate ad conversions (https://developer.chrome.com/en/docs/privacy-sandbox/attribution-reporting). In this application, one party holds a set of identifiers corresponding to users who have seen the advertising campaign, and the identifiers and integer values held by the other party correspond to the users who bought the related item and how much they paid, respectively. The goal is to compute the sum of integer values associated with all matched identifiers and the number of matched identities, but nothing more. A maliciously secure PSI-SUM protocol [57] was proposed to compute the above task using shuffling via homomorphic encryption and zero-knowledge proofs, which is computationally expensive but communicationally efficient. We can compute the same functionality in the secret-shared fashion using our SSS protocol plus two-party secure computation.

B. Combinatorial Analysis

We define our cut-and-choose leakage reduction game and propose an efficient method to compute the tightest B in the batch correlation generation setting.

We define a "ball-and-bin" game to capture selective failure attacks from a corrupted adversary in the offline phases. The game $G(N, B, n, k)$ is defined as follows:

- *Setup*: The game samples NB random permutations, where $B \geq \lceil \frac{\lambda}{\log_2 n} \rceil$.
- *Offline-phase attack*: For the i-th permutation π_i, the adversary can guess c_i bits of information about π_i. The adversary can guess k bits of information combined from all NB permutations. If any guess in this phase fails, the game aborts and terminates. If all guesses are correct, we call these attacked permutations as leaky permutations and non-attacked ones non-leaky permutations.
- *Cut-and-choose*: the game randomly shuffles all NB permutations and divides NB permutations into N buckets sequentically, each contains B permutations.
- *Output*: the game outputs 1 if 1) the game does not abort and 2) every bucket contains $\geq \lceil \frac{\lambda}{\log_2 n} \rceil$ non-leaky permutations; the first condition corresponds to the adversary succeeds in the offline attack and the second condition is required to defeat against online-phase attacks.

We need to ensure that all buckets contain sufficient non-leaky permutations, under the condition that there exists $k \in [\lambda]$ leaky permutations in the cut-and-choose game. Specifically, we want

$$\Pr[G(N, B, n, k) = 1] \geq 1 - 2^{-\lambda},$$

For all $k \in [\lambda]$.

We use generating functions [42, 43] to compute the exact combinations that each bucket contains less than $M = B - \lceil \frac{\lambda}{\log_2 n} \rceil$ leaky permutations. Let us first define the generating function for one single bucket. For a bucket with a capacity of B, the bucket may contain $i \in [0, B]$ leaky permutations, thus there are $\binom{B}{i}$ ways to have i ($i \in [0, B]$) leaky permutations in the bucket. The generating function for a single bucket containing at most M leaky permutations can be represented as

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_M x^M, \tag{3}$$

where $a_i = \binom{B}{i} = \frac{B!}{i!(B-i)!}$. Since there are N buckets in the game, the generating function to count all layouts following the condition is

$$C(x) = A(x)^N = c_0 + c_1 x + c_2 x^2 + \cdots + c_{NM} x^{NM}.$$

Following the definition of generating functions, the coefficient c_k corresponds to all possible placements of k leaky permutations over N buckets under the condition that each bucket

Dimension n	Cut-and-choose batch size N																
	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
2^4	27	24	21	20	19	18	17	16	15	15	15	14	14	14	14	13	13
2^5	25	22	19	18	16	15	15	13	13	13	12	12	12	12	12	11	11
2^6	23	20	18	17	15	14	14	13	12	12	12	11	11	11	11	10	10
2^7	22	19	17	16	14	13	13	12	11	11	11	10	10	10	10	9	9
2^8	21	18	16	14	13	12	12	11	10	10	10	9	9	9	9	8	8
2^9	21	18	16	14	13	12	12	11	10	10	10	9	9	9	9	8	8
2^{10}	20	17	15	13	12	11	10	10	9	9	9	8	8	8	8	7	7

TABLE V: Bucket size for statistical security $\lambda = 40$. Given batch size $N \in \{2^4, 2^5, \cdots, 2^{19}, 2^{20}\}$ and tuple dimension $T \in \{2^4, \cdots, 2^9, 2^{10}\}$, the bucket size we give in the table can ensure that each of N buckets contains at least $\lceil \lambda / \log_2(n) \rceil$ non-leaky tuples with probability $1 - 2^{\lambda}$.

contains less or equal to M leaky permutations after the cut-and-choose game.

If we can efficiently compute c_k, then we have $\Pr[\text{Game}(N, B, n, k) = 1] = c_k / \binom{NB}{k}$. To ensure $\Pr[\text{Game}(N, B, n, k) = 1] \geq 1 - 2^{-\lambda}$, we only need

$$c_k / \binom{NB}{k} \geq 1 - 2^{-\lambda}. \tag{4}$$

Our goal is: given λ, N, and n, we need to find the minimal B such that the inequation (4) always holds for all $k \in [\lambda]$.

Fast parameter computation. Using generating functions for combinatorial analysis is a generic method, the challenge is handling combination explosion when computing $C(x)$. We propose a fast polynomial exponentiation algorithm PolyExp in Fig. 16. We first leverage "square-and-multiply" trick [58] (Line. 3-6, Algorithm 16) to reduce polynomial multiplication complexity from $O(N)$ to $O(\log N)$. However, the computation overhead is still high as the polynomial coefficients are large during computation and the polynomial degree doubles after each polynomial multiplication.

Algorithm PolyExp$(A(x), N)$

1: $C(x) \leftarrow 1$, $T(x) \leftarrow A(x)$.
2: **while** $N > 0$ **do**
3: **if** N is odd **then**
4: $C(x) \leftarrow C(x) \cdot T(x)$, $N \leftarrow N - 1$.
5: **end if**
6: $T(x) \leftarrow T(x)^2$, $N \leftarrow N/2$.
7: Get $\{t_i\}_{i \in [0,\lambda]}$ of $T(x)$. Set $T(x) \leftarrow \sum_{i=0}^{\lambda} t_i x^i$.
8: Get $\{c_i\}_{i \in [0,\lambda]}$ of $C(x)$. Set $C(x) \leftarrow \sum_{i=0}^{\lambda} c_i x^i$.
9: **end while**
10: **return** $C(x)$.

Fig. 16: Fast polynomial exponentiation algorithm

Since we care about the coefficient c_k for $k \in [B, \lambda]$, only the coefficients of lower degrees $k \in [B, \lambda]$ contribute to the computation. Hence, we can drop all high-degree coefficients during computation (Line. 7-8, Algorithm 16). This allows two low-degree polynomials throughout the computation, saving huge computation overhead. Since $B \in [\lceil \frac{\lambda}{\log_2 n} \rceil, \lambda + \lceil \frac{\lambda}{\log_2 n} \rceil]$, we use binary search to find the minimal B that just satisfies inequation (4). Table V shows the computation result for different N's under statistical security parameter $\lambda = 40$.

Comparison with existing analysis methods. We stress that our cut-and-choose analysis method is different from the existing ones used in [39, 59]; though we get inspirations from this method. The method from [39, 59] is for checking the correctness of multiplication triples. Their main idea is to first randomly pick a few tuples and open them completely and the remaining ones are bucketed, and then the parties check the correctness of triples in each bucket using a triple-based re-execution method. Differently, our goal is for leakage reduction rather than for correctness check (all correlations are checked before bucketing). This results in different winning conditions for the adversaries in the two cut-and-choose games. In addition, our method uses generating functions (not concentration inequalities) to compute B, which can compute the tightest B.

C. k-party SSS Protocol

Functionality $\mathcal{F}_{k\text{-oss}}$

Parameters: $n \in \mathbb{N}$; a dictionary Val storing all authenticated values; k parties set $\mathcal{P} = \{P_i\}_{i \in [k]}$.

Inputs: A party $\mathtt{R} \in \mathcal{P}$ provides a permutation $\pi \in \mathbf{S}_n$; the identifiers $\{\mathsf{id}_i\}_{i \in [n]}$ for the values being shuffled, and the identifiers $\{\mathsf{id}'_i\}_{i \in [n]}$ for the shuffled values to be stored with.

Functionality: receiving $(\mathsf{oss}, \pi, \{\mathsf{id}_i\}_{i \in [n]}, \{\mathsf{id}'_i\}_{i \in [n]})$ from $\mathtt{R}$ and $(\mathsf{Shuffle}, \{\mathsf{id}_i\}_{i \in [n]}, \{\mathsf{id}'_i\}_{i \in [n]})$ from $\mathcal{P} \setminus \{\mathtt{R}\}$:

1. If $\pi \notin \mathbf{S}_n$, abort.
2. Define a vector $\vec{x}$ such that $\vec{x}_i = \mathsf{Val}[\mathsf{id}_i]$ for $i \in [n]$.
3. Compute $\vec{y} = \pi(\vec{x})$ and store $\mathsf{Val}[\mathsf{id}'_i] \leftarrow \vec{y}_i$ for $i \in [n]$.

Fig. 17: Functionality for k-party OSS

Ideal k-party OSS functionality. We define a k-party one-side secret-shared shuffle functionality $\mathcal{F}_{k\text{-oss}}$ in Fig. 17. One single party $\mathtt{R}$ provides a permutation π which is used to shuffle the authenticated secret-shared vector $\vec{x}$ stored by the functionality. $\mathcal{F}_{k\text{-oss}}$ checks if π is well-formed, and performs shuffle over $\vec{x}$ to obtain a new shuffled vector $\vec{y} = \pi(\vec{x})$.

k-party OSS protocol. We provide an k-party OSS protocol $\Pi_{k\text{-oss}}^{[P_i]}$ in Fig. 18 to compute $\mathcal{F}_{k\text{-oss}}$. $\Pi_{k\text{-oss}}^{[P_i]}$ is an extension of the two-party OSS protocol $\Pi_{\mathsf{oss}}^{[\mathtt{R}]}$. In particular, it runs a pair-wise CGP shuffle between any two parties. Now we show the offline phase and online phase, respectively.

Offline phase. In the offline phase, P_i run the two-party

cut-and-choose tuple generation protocols with each P_j for $j \in [k] \setminus \{i\}$. To ensure correctness, P_i has to use the same NB permutations and the same permutation ρ with each of P_j for $j \in [k] \setminus \{i\}$. The formal description can be found from the offline phase of $\Pi_{k\text{-oss}}^{[P_i]}$ in Fig. 18.

Online phase. The online phase of $\Pi_{k\text{-oss}}$ is a straightforward extension of the two-party OSS protocol $\Pi_{\mathsf{oss}}^{[\mathbb{R}]}$. The goal of the online phase is to (secret-shared) shuffle $\langle \vec{x} \rangle$ using B permutations $\{\pi_t^{(i)}\}_{t\in[B]}$ known by the party P_i.

Protocol $\Pi_{\mathsf{oss}}^{[P_i]}$

Parameters: $n \in \mathbb{N}$; N denotes cut-and-choose bucket number and B denotes cut-and-choose bucket size. k parties $\mathcal{P} = \{P_j\}_{j\in[k]}$; a party $P_i \in \mathcal{P}$ with id $i \in [k]$.

[Offline]: On inputting sharings N and B, do the following:

1. For any $j \in [k] \setminus \{i\}$, the parties P_i and P_j invoke $\mathcal{F}_{\mathsf{GenM}}$ to generated NB OPM instances, where P_i plays the role of OPM receiver and provides NB permutations $\{\pi_t^{(i)}\}_{t\in[NB]}$. We denote the t-th shuffle tuple between P_i and P_j as $(\!|\pi_t^{(i)}|\!)^{(j)} = ((\pi_t^{(i)}, \Delta_t^{(i,j)}), (\vec{a}_t^{(i,j)}, \vec{b}_t^{(i,j)})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n)$ where $t \in [NB]$.
2. P_i samples a random seed $s \xleftarrow{\$} \{0,1\}^\kappa$ and sends s to $\mathcal{P} \setminus \{P_i\}$. All the parties use s to determine a random permutation $\rho \in \mathbf{S}_{NB}$.
3. For $j \in [k] \setminus \{i\}$, P_i and P_j use ρ to shuffle the order of NB OPMs and divide the permuted OPM instances into N bucket, each containing B OPMs. The parties can locally compress OPMs to get NB shuffle tuples.

[Online]: On inputting sharings $\langle \vec{x} \rangle$, do the following:

1. For $j \in [k] \setminus \{i\}$, fetch the next unused bucket of B tuples between P_i and P_j, in which the t-th shuffle is denoted as $(\!|\pi_t^{(i)}|\!)^{(j)} = ((\pi_t^{(i)}, \Delta_t^{(i,j)}), (\vec{a}_t^{(i,j)}, \vec{b}_t^{(i,j)})) \in (\mathbf{S}_n \times (\mathbb{F}^2)^n) \times ((\mathbb{F}^2)^n \times (\mathbb{F}^2)^n)$.
2. Let $\langle \vec{x}^{(0)} \rangle \leftarrow \langle \vec{x} \rangle$.
3. For $t \in [B]$:
   a) For $j \in [k] \setminus \{i\}$, P_j sends $\vec{\delta}_t^{(i,j)} = \langle \vec{x}^{(t)} \rangle_j - \vec{a}_t^{(i,j)}$ to P_i, and P_j sets $\langle \vec{x}^{(t+1)} \rangle_j \leftarrow \vec{b}_t^{(i,j)}$.
   b) P_i receives $\vec{\delta}_t^{(i,j)}$ from P_j for $j \in [k] \setminus \{i\}$. P_i computes $\langle \vec{x}^{(t+1)} \rangle_i \leftarrow \pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i + \sum_{j\in[k]\setminus\{i\}} \vec{\delta}_t^{(i,j)}) + \sum_{j\in[k]\setminus\{i\}} \vec{\Delta}_t^{(i,j)}$.
4. Run $b \leftarrow \Pi_{\mathsf{MACCheck}}$ over $(\langle \vec{x}^{(1)} \rangle, \langle \vec{x}^{(2)} \rangle, \cdots, \langle \vec{x}^{(B)} \rangle)$. If $b = \mathsf{False}$, abort. Otherwise, output $\langle \vec{x}^{(B)} \rangle$.

Fig. 18: Maliciously secure k-party OSS protocol.

The parties run B iteration as required for leakage reduction. In the t-th iteration, P_i run $k-1$ pair-wise CGP shuffle with each P_j for $j \in [k] \setminus \{i\}$ (step 3, $\Pi_{k\text{-oss}}.\mathsf{online}$). This is done by using the shuffle tuples that P_i has generated with each P_j for $j \in [k] \setminus \{i\}$. In particular, for $j \in [k] \setminus \{i\}$, each P_j sends the CGP shuffle message $\vec{\delta}_t^{(i,j)} = \langle \vec{x}^{(t)} \rangle_j + \vec{a}_t^{(i,j)}$ to P_i (step 3a, $\Pi_{k\text{-oss}}.\mathsf{online}$). After receiving all $\vec{\delta}_t^{(i,j)}$ for $j \in [k] \setminus \{i\}$, P_i can compute $\langle \vec{x}^{(t+1)} \rangle_i \leftarrow \pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i + \sum_{j\in[k]\setminus\{i\}} \vec{\delta}_t^{(i,j)}) + \sum_{j\in[k]\setminus\{i\}} \vec{\Delta}_t^{(i,j)}$ (step 3b, $\Pi_{k\text{-oss}}.\mathsf{Online}$). If all parties behave honestly, one can verify correctness:

$$
\begin{aligned}
&\langle \vec{x}^{(t+1)} \rangle_i + \sum_{j\in[k]\setminus\{i\}} \langle \vec{x}^{(t+1)} \rangle_j \\
=&\pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i + \sum_{j\in[k]\setminus\{i\}} \vec{\delta}_t^{(i,j)}) + \sum_{j\in[k]\setminus\{i\}} \vec{\Delta}_t^{(i,j)} + \sum_{j\in[k]\setminus\{i\}} \vec{b}_t^{(i,j)} \\
=&\pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i) + \pi_t^{(i)}(\sum_{j\in[k]\setminus\{i\}} \vec{\delta}_t^{(i,j)}) + \sum_{j\in[k]\setminus\{i\}} \vec{\Delta}_t^{(i,j)} + \sum_{j\in[k]\setminus\{i\}} \vec{b}_t^{(i,j)} \\
=&\pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i) + \pi_t^{(i)}(\sum_{j\in[k]\setminus\{i\}} \langle \vec{x}^{(t)} \rangle_j) - \pi_t^{(i)}(\sum_{j\in[k]\setminus\{i\}} \vec{a}_t^{(i,j)}) \\
&+ \sum_{j\in[k]\setminus\{i\}} (\pi_t^{(i)}(\vec{a}_t^{(i,j)}) - \vec{b}_t^{(i,j)}) + \sum_{j\in[k]\setminus\{i\}} \vec{b}_t^{(i,j)} \\
=&\pi_t^{(i)}(\langle \vec{x}^{(t)} \rangle_i) + \pi_t^{(i)}(\sum_{j\in[k]\setminus\{i\}} \langle \vec{x}^{(t)} \rangle_j) \\
=&\pi_t^{(i)}(\vec{x}^{(t)}).
\end{aligned}
$$

This means $\langle \vec{x}^{(t+1)} \rangle$ is a correct shuffle of $\langle \vec{x}^{(t)} \rangle$ using permutation $\pi_t^{(i)}$. By a simple induction arguement, we have $\vec{x}^{(B)} = \pi(\vec{x})$ where $\pi = \pi_{B-1} \circ \cdots \circ \pi_1 \circ \pi_0$.

At the end of the online protocol, the parties run the MAC check protocol Π_{MACCheck} for all intermediate sharings $\{\langle \vec{x}^{(t)} \rangle\}_{t\in[B]}$ and abort if the check fails.

k-party SSS protocol. With the k-party OSS protocol $\Pi_{k\text{-oss}}$, Fig. 19 shows how to construct a k-party SSS protocol $\Pi_{k\text{-sss}}$.

Protocol $\Pi_{k\text{-sss}}$

Parameters: N denotes cut-and-choose bucket number and B denotes cut-and-choose bucket size.

[Offline]: On inputting N and B, do the following:

1. For $i \in [k]$, the parties run $\Pi_{\mathsf{oss}}^{[P_i]}.\mathsf{Offline}(N, B)$, where P_i provides NB permutations.

[Online]: On inputting sharings $\langle \vec{v} \rangle$, do the following:

1. Let $\langle \vec{v}^{(0)} \rangle \leftarrow \langle \vec{v} \rangle$.
2. For $i \in [k]$, run $\langle \vec{v}^{(i+1)} \rangle \leftarrow \Pi_{\mathsf{oss}}^{[P_i]}.\mathsf{Online}(\langle \vec{v}^{(i)} \rangle)$.
3. Ourput $\langle \vec{v}^{(k)} \rangle$.

Fig. 19: k-party maliciously secure SSS protocol.

In the offline phase, each party P_i for $i \in [k]$ run the OSS offline tuple generation protocol with the other $k-1$ parties. In doing this, each pair of two parties $(P_i, P_j)_{i\in[k], j\in[k]\setminus\{i\}}$ will share NB shuffle tuples such that P_i knows the permutations.

In the online phase, each party P_i takes turns running $\langle \vec{v}^{(i+1)} \rangle \leftarrow \Pi_{\mathsf{oss}}^{[P_i]}.\mathsf{Online}(\langle \vec{v}^{(i)} \rangle)$ with the other $k-1$ parties, which shuffles $\langle \vec{v}^{(i)} \rangle$ using B permutations choosen by P_i. The protocol outputs $\langle \vec{v}^{(k)} \rangle = \pi(\vec{v})$ where π is a composition of kB permutations in which each party contributes B of them. In this way, no subset of k parties learn information about π.

D. More Details about GBN Decomposition

In this section, we detail how the GBN decomposition works and how the decomposition trick is combined with our cut-and-choose batch tuple generation protocol.

Benes Network [44]. The Benes network (BN) for permuting n elements has $2\log n - 1$ layers, each layer containing $n/2$ 2-element permutations. Suppose the inputs are indexed with $0, 1, ..., n-1$, and each index σ is parsed as $\sigma_1||\sigma_2||\cdots||\sigma_h$ where $n = 2^h$. The j-th and $2\log n - j$-th layer of the Benes network contains 2-element permutations, each acting on two indexes $\sigma_1||\cdots||\sigma_{j-1}||0||\sigma_{j+1}||\cdots||\sigma_h$ and $\sigma_1||\cdots||\sigma_{j-1}||1||\sigma_{j+1}||\cdots||\sigma_h$, for all σ_1, $\cdots$, σ_{j-1}, σ_{j+1}, $\cdots$, $\sigma_h \in \{0,1\}^{h-1}$.

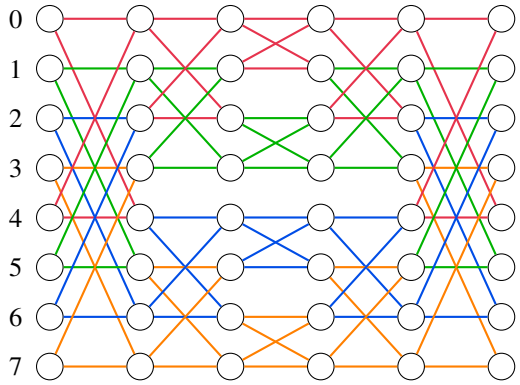

Fig. 20: An example Benes network structure for $\mathbf{S}_8$.

Fig. 20 shows a toy example of the Benes network, containing 5 layers from the left to the right. We use edges between nodes to denote the possible swap within each layer. In particular, there are 4 pairs of 2-element permutations for each layer, and we mark each of them with different colors. As we can see, each 2-element permutation forms a butterfly structure, which can be either an identity permutation or a swap when configuring the network concretely.

Generilized Benes Network [1]. BN decomposes a permutation using 2-element permutations. Chase *et al.* [1] proposed a generalized decomposition trick to decompose permutations to T-element permutations where $T = 2^t$ for $t \geq 1$. GBN decomposes a permutation $\pi \in \mathbf{S}_n$ as $\pi_1 \circ \cdots \circ \pi_d$ where $d = \lceil \frac{\log n}{t} \rceil$. To achieve this, we set π_1 to consist of the first t layers $(1, \cdots, t)$ of the Benes network, π_2 to consist of the next t layers $(t+1, \cdots, 2t)$, and so on, and the middle permutation $\pi_{\lfloor \frac{d}{2} \rfloor + 1}$ consists of $2t-1$ layers. Each layer of GBN consists of n/T T-element permutations. GBN sets each π_i for $i = 1, \cdots, \lceil \frac{d}{2} \rceil$, to consist of t consecutive layers number $i \cdot t - (t-1), \cdots, i \cdot t - 1, i \cdot t$, and π for $i = \lceil \frac{d}{2} \rceil + 2, \cdots, d$ are defined symmetricly. For the i-th layer, π_i can be further decomposed as n/T T-element permutations, and each permutation is defined over $\sigma_1||\cdots||\sigma_{(i-1)\cdot t}||x||\sigma_{i\cdot t}, \cdots, \sigma_h$ for $x \in \{0,1\}^t$ and fixed $\sigma_1, \cdots, \sigma_{(i-1)\cdot t}, \sigma_{(i+1)\cdot t}, \cdots, \sigma_h$. For example, when $T = 4$, one can parse the 5-layer BN in Fig. 20 as a 3-layer GBN, where the 1-th and 3-th layers of the GBN correspond to the 1-th and 5-th layers of the BN, respectively, and the 2-th layer of GBN corresponds to the combined (2,3,4)-th layers of the BN. Each layer of the GBN consists of two 4-element permutations. In particular, the 4-element permutations of the first layer of the GBN are defined over indexes $(0, 2, 4, 6)$ and $(1, 3, 5, 7)$, respectively.

GBN with Cut-and-Choose Batch Generation. One should be careful when using GBN-based optimization in our maliciously secure SSS protocol. Recall that we use cut-and-choose-based shuffle tuple generation. To enable GBN-based optimization, one may simply generate NB shuffle tuples for NB random T-element permutations using our cut-and-choose-based tuple generation, and then compose sufficient shuffle tuples using the GBN trick in the online phase to perform shuffle over n secret-shared values. We call the above method the *bottom-up* approach. However, it's known that this bottom-up approach cannot generate a uniformly random n-element permutation [49].

Instead, one should always use a *top-down* manner following the GBN decomposition. In particular, the sender should sample random n-element permutations first, and then obtain many smaller T-element permutations by decomposing the bigger permutations following the GBN structure. Now, for each obtained T-element permutation π, the sender further samples B T-element *random* permutations $\pi_0, \pi_1, \cdots, \pi_{B-1}$ under the constraint $\pi = \pi_{B-1} \circ \cdots \circ \pi_2 \circ \pi_1$. After that, all T-element permutations will be fed into the cut-and-choose game. However, recall that these permutations will be shuffled by a random permutation ρ chosen by the sender in our cut-and-choose game, the sender needs to ensure that these T-element permutations still form the pre-chosen n-element permutations after the shuffling. Our method is that the sender uses the reverse permutation ρ^{-1} to pre-organize all T-element permutation *before* the cut-and-choose game so that they can still form the desired random n-element permutations when they are combined together following the GBN structure, after the cut-and-choose tuple generation phase.

E. Attacks to Laud's SSS protocol [3]

Laud [3] proposed an oblivious permutation protocol by combining shuffle tuples with authenticated secret sharing. The protocol exploits the idea from the CGP protocol to perform shuffling. We review the protocol and show our attacks.

Protocol description. Laud's SSS protocol [3] assumes that an ideal functionality $\mathcal{F}_{\mathsf{prep}}.\mathsf{shuffle}$ can generate the desired CGP shuffle tuples. $\mathcal{F}_{\mathsf{prep}}.\mathsf{shuffle}$ receives a permutation π_i from P_i, generates a shuffle tuple, and sends $\vec{z}^{(i,j)}$ to P_i and $\vec{x}^{(i,j)}, \vec{y}^{(i,j)}$ to P_j such that $\vec{z}^{(i,j)} = \pi_i(\vec{x}^{(i,j)}) - \vec{y}^{(i,j)}$.

In the online phase, the parties essentially permute the secrets and the corresponding MAC values using two independent instances of shuffle tuples with the same underlying permutation; one is applied over the shared secrets, and another is applied over the MAC values of the secrets. The parties may not follow the protocol honestly, *e.g.*, adding errors to protocol messages. In [3], the authors use a polynomial-based check technique to enforce honest behaviors. In particular, to check whether $\vec{v}^{(k)} = \pi(\vec{v}^{(0)})$, the parties use a polynomial $f(\vec{x}, r) = \prod_{i \in [n]}(\vec{x}_i - r)$, where $r \xleftarrow{\$} \mathbb{F}$. Therefore, if $f(\vec{v}^{(0)}, r) = f(\vec{v}^{(k)}, r)$, we have $\vec{v}^{(k)} = \pi(\vec{v}^{(0)})$ except with probability $n/|\mathbb{F}|$, by the Schwartz–Zippel lemma. In the protocol, the parties use $\langle r' \rangle$ to hide the revealed value, ensuring no additional information is leaked from the check.

Attacks. The construction of [3] relies on a correlation generation protocol to securely realize $\mathcal{F}_{\mathsf{prep}}.\mathsf{shuffle}$, but [3] didn't give a concrete protocol for $\mathcal{F}_{\mathsf{prep}}.\mathsf{shuffle}$. As we showed in §III-B, even if the correlations are correctly and securely generated, simply combining shuffle tuples with authenticated

Protocol $\Pi_{\mathsf{ObliviousShuffle}}$

[Offline]: Each pair of two parties (P_i, P_j) generate sufficient number of shuffle tuples:

1. Each party P_i selects a random permutation of length n:
2. Each pair of parties P_i and P_j, runs two instances of $\mathcal{F}_{\mathsf{prep}}.\mathsf{shuffle}$, with P_i providing the input π_i. Let the result of t-th instance be $\vec{z}_t^{(i,j)}$ for P_i and $\vec{x}_t^{(i,j)}$, $\vec{y}_t^{(i,j)}$ for P_j.

[Online] On inputting a secret-shared vector $\langle \vec{v}^{(0)} \rangle$ of length n:

1. For $i \in [k]$, do the following:
   a) For $j \neq i$, do the following:
      i) P_j sends $\vec{w}_1^{(i,j)} \leftarrow [\![\vec{v}^{(i-1)}]\!]_j - \vec{x}_1^{(i,j)}$ and $\vec{w}_2^{(i,j)} \leftarrow [\![\gamma(\vec{v}^{(i-1)})]\!]_j - \vec{x}_2^{(i,j)}$
      ii) P_i computes $\vec{r}_{ij} \leftarrow \pi_i(\vec{w}_1^{(i,j)}) + \vec{z}_1^{(i,j)}$ and $\vec{s}_{ij} \leftarrow \pi_i(\vec{w}_2^{(i,j)}) + \vec{z}_2^{(i,j)}$.
      iii) P_j defines shares of next round: $[\![\vec{v}^{(i)}]\!]_j = \vec{y}_1^{(i,j)}$ and $[\![\gamma(\vec{v}^{(i)})]\!]_j = \vec{y}_2^{(i,j)}$.
   b) P_i defines $\vec{r}_{ii} \leftarrow \pi_i([\![\vec{v}^{(i-1)}]\!]_i)$ and $\vec{s}_{ii} \leftarrow \pi_i([\![\gamma(\vec{v}^{(i)})]\!]_i)$.
   c) P_i defines shares for next round: $[\![\vec{v}^{(i)}]\!]_i = \sum_j \vec{r}_{ij}^{(i)}$ and $[\![\gamma(\vec{v}^{(i)})]\!]_j = \sum_j \vec{s}_{ij}^{(i)}$.
2. The parties pick fresh random $\langle r \rangle$ and $\langle r' \rangle \in \mathbb{F}$ and reveal r.
3. The parties check whether $\langle r' \rangle \cdot (\prod_{i=1}^{n}(r - \langle \vec{v}_i^{(k)} \rangle) - \prod_{i=1}^{n}(r - \langle \vec{v}_i^{(0)} \rangle)) = 0$ using standard SPDZ revealing protocol.
4. If not abort, output $\langle \vec{v}^{(k)} \rangle$

Fig. 21: k-party secret-shared shuffle protocol [3]

secret sharing does not lead to a maliciously secure shuffle protocol. Following the previous discussion, a malicious P_j may perform an online selective failure attack to learn sensitive information about the secret permutation provided by P_i. In particular, P_j may send $\vec{w}_2^{(i,j)} = [\![\gamma(\vec{v}^{(i-1)})]\!]_j - \vec{x}_2^{(i,j)} + \vec{e}$, where $\vec{e} = (0, \cdots, 0, e, \cdots, 0)$ is a weight-1 error vector with non-zero element at position q. Therefore, P_j can guess $\pi_i(\vec{e})$ and set $[\![\vec{v}^{(i)}]\!]_j \leftarrow \vec{y}_1^{(i,j)} - \pi_i(\vec{e})$. For weight-1 vector $\vec{e}$, P_j only needs to guess p such that $q = \pi_i(p)$ to guess $\pi_i(\vec{e})$, with a success probability of $1/n$. If the adversary guesses $q = \pi_i(p)$ correctly, this attack won't be detected by the polynomial-based check in step 3 of $\Pi_{\mathsf{ObliviousShuffle}}$ and the adversary learns information about π_i for free. This selective failure attack proves that [3] cannot achieve full privacy.

F. Attacks to the Eskandarian-Boneh SSS protocol [2]

Eskandarian and Boneh [2] recently proposed an anonymous communication protocol with malicious security. The core of this protocol is a maliciously secure shuffle protocol adapted from [1]. In order to gain more efficiency, its shuffle correlation is different from the one in the original CGP protocol. Nevertheless, we show our attacks discussed in §III still work to [2] by slight modifications.

Three-party shuffle protocol and our attacks. The three-party shuffle protocol relies on a new two-party shuffle correlation such that P_1 holds random vectors $\vec{a}_2', \vec{b}_2 \in \mathbb{F}^n$ and a random permutation $\pi_1 \in \mathbf{S}_n$. P_2 holds a random vector $\vec{a}_1 \in \mathbb{F}^n$, a random permutation $\pi_2 \in \mathbf{S}_n$, and a vector $\vec{\Delta}_2 \in \mathbb{F}^n$ such that $\vec{\Delta}_2 = \pi_2(\pi_1(\vec{a}_1) + \vec{a}_2') - \vec{b}_2$. [2] relies on a third party P_3 to compute $\vec{\Delta}_2$. In particular, P_1 chooses a random seed that extends to get $\pi_1 \in \mathbf{S}_n$ and $\vec{a}_2', \vec{b}_2 \in \mathbb{F}^n$, and sends the seed to P_3, who can use the seed to recover the same values. Similarly, P_2 samples a seed to generate $\pi_2, \vec{a}_1$ and gives the seed to P_3. P_3 computes $\vec{\Delta}_2 \leftarrow \pi_2(\pi_1(\vec{a}_1) + \vec{a}_2') - \vec{b}_2$ and sends it to P_2. P_1 and P_2 also agree on a secret seed and generate a permutation $\pi_{12} \in \mathbf{S}_n$, which is used to hide the underlying permutation from P_3.

Using the above shuffle correlation, P_1 and P_2 can perform shuffle over a secret vector $[\![\vec{x}]\!] = [\![\vec{x}]\!]_1 + [\![\vec{x}]\!]_2$ shared between them (P_3 does not hold shares). In particular, the shuffle protocol runs as follows:

- P_2 sends $\vec{z}_2 \leftarrow [\![\vec{x}]\!]_2 - \vec{a}_1$ to P_1.
- P_1 sends $\vec{z}_1 \leftarrow \pi_1(\vec{z}_2 + [\![\vec{x}]\!]_1) - \vec{a}_2'$ to P_2, and sets $[\![\vec{s}]\!]_1 \leftarrow \vec{b}_2$.
- P_2 sets $[\![\vec{s}]\!]_2 \leftarrow \pi_2(\vec{z}_1) + \vec{\Delta}_2$.

Clearly, $[\![\vec{s}]\!]_1 + [\![\vec{s}]\!]_2 = \vec{b}_2 + \pi_2(\vec{z}_1) + \vec{\Delta}_2 = \vec{b}_2 + \pi_2(\pi_1([\![\vec{x}]\!]_2 - \vec{a}_1 + [\![\vec{x}]\!]_1) - \vec{a}_2') + \pi_2(\pi_1(\vec{a}_1) + \vec{a}_2') - \vec{b}_2 = \pi_2 \circ \pi_1(\vec{x})$. Since P_3 knows π_1 and π_2, P_1 and P_2 additionally use π_{12} to shuffle over each party's share using π_{12}. In this manner, neither party knows the underlying permutation. [2] extended the above idea to shuffle authenticated secret-shared vectors, where MACs are used to detect possible errors. To check whether any party deviates from the protocol, P_1 and P_2 run the MAC check protocol at the end of the protocol and abort if the check fails.

We show that a similar online-phase attack from §III-B can be adapted to attack the three-party shuffle protocol over authenticated secret sharings. Suppose P_2 is corrupted. Now P_2 can instead send $\vec{z}_2 \leftarrow [\![\vec{x}]\!]_2 - \vec{a}_1 + \vec{e}$ where $\vec{e} = (0, \cdots, 0, e, \cdots, 0)$ is a weight-1 error vector with non-zero element at position q. After receiving $\vec{z}_1$ from the honest P_1, P_2 can guess p with $\pi_1(p) = q$ and subtract e from $\vec{z}_1[p]$ *before* setting $[\![\vec{s}]\!]_2$. If P_2 guessed $\pi_1(p) = q$ correctly, P_2 would just remove the introduced error, and the parties would still share a well-formed permutated authenticated vector; otherwise, the protocol would abort from MAC check because the integrity of the permutated vector is undermined by the introduced errors. In the above attack, the probability for P_2 to guess $\pi_1(p)$ correctly is $1/n$. Also note that the composed permutation $\pi = \pi_{12} \circ \pi_2 \circ \pi_1$ and P_2 knows π_2 and π_{12}, which means P_2 can learn $\pi(p)$ in a successful attack without been caught.

k-party shuffle protocol and our attacks. In a k-party shuffle correlation, each P_i for $i \in [1, k]$ holds a random permutation $\pi_i \in \mathbf{S}_n$ and random vectors $\vec{a}, \vec{b}, \vec{a}' \in \mathbb{F}^n$, and P_k additionally holds a vector $\vec{\Delta}_k \in \mathbb{F}^n$ such that $\vec{\Delta}_k = \pi_k(\cdots \pi_2(\pi_1(\sum_{i=2}^{k} \vec{a}_i) + \vec{a}_1') + \vec{a}_2' \cdots + \vec{a}_{k-1}') - \sum_{i=1}^{k-1} \vec{b}_i$.

The Eskandarian-Boneh SSS protocol [2] proposed a candidate protocol for generating k-party shuffle correlation by adapting the CGP correlation generation protocol [1]. The parties firstly run the CGP shuffle tuple generation protocol to generate tuple $(\vec{a}_{i,j}, \vec{b}_{i,j}, \vec{\Delta}_{i,j})$ for all $i, j \in [k]$, $i \neq j$ such that $\vec{\Delta}_{i,j} = \pi_i(\vec{a}_{i,j}) - \vec{b}_{i,j}$, with P_i holding $(\pi_i, \vec{\Delta}_{i,j})$ and P_j holding $(\vec{a}_{i,j}, \vec{b}_{i,j})$. The only difference is that [2] replaces semi-honest OTs in the CGP protocol with maliciously secure OTs.

Using shuffle tuples shared between each pair of two parties, the parties can jointly generate an k-party shuffle correlation. In particular, the parties have to produce vectors $\vec{a}_i, \vec{b}_i, \vec{a}'_i$ for each P_i and additionally $\vec{\Delta}_k$ for P_k. The k-party shuffle correlation from the CGP shuffle correlation is computed as follows:

- $\vec{a}_i \leftarrow \vec{a}_{1,i}$, $\vec{b}_i \leftarrow \vec{b}_{1,i}$
- $\vec{a}'_i \leftarrow \sum_{j \neq i} \vec{b}_{i,j} + \vec{\Delta}_{i,j} + \vec{a}_{i+1,j}$
- $\vec{\Delta}_k \leftarrow \sum_{j \in [1,k-1]} \vec{\Delta}_{k,j}$

Note that here computing $\vec{a}_i$, $\vec{b}_i$ and $\vec{\Delta}_k$ only requires local computation, but computing $\vec{a}'_i$ is done by adding secrets held by different parties.

The parties can use such k-party shuffle correlation to shuffle a secret-shared vector $[\![\vec{x}]\!]$ and output $[\![\vec{s}]\!]$, where $\vec{s} = \pi(\vec{x})$ and $\pi = \pi_k \circ \pi_{k-1} \cdots \circ \pi_1$. The protocol uses k-party shuffle correlation to perform shuffle as follows:

- For each party P_i, where $i \neq 1$, computes $\vec{z}_i \leftarrow [\![\vec{x}]\!]_i - \vec{a}_i$ and sends $\vec{z}_i$ to P_1.
- P_1 computes $\vec{z}'_1 \leftarrow \pi_1(\sum_{i=2}^{n} \vec{z}_i) - \vec{a}'_1$ and sends $\vec{z}'_1$ to P_2. P_1 sets its output to $[\![\vec{s}]\!]_1 \leftarrow \vec{b}_1$.
- For $i \in [2, k-1]$, P_i computes $\vec{z}'_i \leftarrow \pi_i(\vec{z}'_{i-1}) - \vec{a}'_i$ and sends it to P_{i+1}. P_i sets its output to $[\![\vec{s}]\!]_i \leftarrow \vec{b}_i$.
- P_k outputs $[\![\vec{s}]\!]_k \leftarrow \pi_k(\vec{z}'_{k-1}) + \vec{\Delta}_k$.

To achieve malicious security, [2] uses the above protocol to simultaneously shuffle authenticated secret-shared vector $\langle \vec{s} \rangle = ([\![\vec{s}]\!], [\![\gamma(\vec{s})]\!])$. At the end of the protocol, the parties run MAC check over the shuffled authenticated vector $\langle \vec{s} \rangle$ to check integrity. As we showed in §III-B, a post-execution check alone cannot ensure full privacy. Such k-party shuffle protocol is still leaky due to the online-phase selective failure attack. Besides, [2] does not check the correctness of correlations used for shuffling, thus our proposed malicious attacks to the offline phase (refer to §III-C) still work over [2], which break privacy. In particular, we have the following offline-phase and online-phase attacks to the k-party shuffle protocol [2], by slightly modifying our attacks from §III-B and §III-C.

Offline-phase attacks. [2] does not check the correctness of correlations in the offline phase, hence we can modify our offline attacks from §III-C to attack [2].

First, using maliciously secure OT for the CGP correlation generation cannot guarantee full privacy because such enhanced protocol still suffers from a selective failure attack from the OT sender via manipulating OT messages (*i.e.*, by providing an inconsistent GGM tree), which can leak information about honest parties' permutation in a successful attack (refer to OPV attack in §III-C). In particular, suppose the adversary corrupts $k-1$ parties and we denote the non-corrupted party as P_h for $h \in [1, k]$. The adversary can provide non-well-formed GGM trees when running a two-party CGP shuffle correlation generation protocol with P_h. If the attack was successful, the parties would still share shuffle tuples of the correct form. We note that [2] does not check the correctness of shuffle tuples in the offline phase, thus [2] does not reveal the information in the offline phase. Nevertheless, the adversary can still learn whether its offline-phase attack is successful or not from the online-phase MAC check result. If the adversary guessed correctly in the selective failure attack, it could pass the MAC check and learn sensitive information about the secret permutation π_h held by the honest party P_h.

In addition, [2] does not check whether an OPM receiver provides a valid permutation in the OPM generation protocol. Therefore, our proposed OPM attack still applies to [2], which breaks privacy (refer to §III-C). In particular, suppose the adversary corrupts $k-1$ parties and we denote the non-corrupted party as P_h for $h \in [1, k]$. The adversary, when playing the role of an OPM receiver with P_h, can run the OPM attack from §III-C with the honest party P_h, which can learn $n-1$ secret shares and mac shares of P_h. Since the adversary corrupted the other $k-1$ parties, the parties can now recover $n-1$ secrets and their MACs.

Online-phase attacks. The adversary can perform selective failure attacks in the online shuffle phase. Again, assume the adversary corrupts $k-1$ parties and denote the non-corrupted party as P_h for $h \in [1, k]$. The adversary can attack as follows:

- Case 1: $h = 1$. In this case, the adversary, on behalf of P_2, can instead send $\vec{z}_2 \leftarrow [\![\vec{x}]\!]_i - \vec{a}_i + \vec{e}$ to P_1, where $\vec{e} = (0, \cdots, 0, e, \cdots, 0)$ is a weight-1 error vector with non-zero element at position q. After receiving $\vec{z}'_1$ from P_1, the adversary will subtract e from $\vec{z}'_1[p]$ before computing $\vec{z}'_2$. In this way, the adversary guesses $q = \pi_1(p)$. Now if the adversary guessed correctly, the protocol would proceed normally because the attacker successfully removed the error; otherwise, the error would still remain in the shuffled vector, and the protocol would abort from the MAC check.
- Case 2: $h \in [2, k-1]$. In this case, the adversary can add a weight-1 error vector into the protocol message $\vec{z}'_{h-1}$ and remove the (permuted) error vector from $\vec{z}'_{h+1}$. Specifically, the adversary, on behalf of P_{h-1}, can corrupt $\vec{z}'_{h-1}$ by updating $\vec{z}'_{h-1} = \vec{z}'_{h-1} + \vec{e}$ where $\vec{e} = (0, \cdots, 0, e, \cdots, 0)$ is a weight-1 error vector with non-zero element at position q. After receiving $\vec{z}'_h$ from P_h on behalf of P_{h+1}, the adversary, before computing $\vec{z}'_{h+1}$, can remove e from $\vec{z}'_h[p]$ for $p \in [m]$. In this way, the adversary guesses $q = \pi_h(p)$.
- Case 3: $h = k$. The adversary can add an error into the protocol message $\vec{z}'_{k-1}$ and remove the (permuted) error from $\vec{b}_{k-1}$. In particular, the adversary, on behalf of P_{k-1}, can compute $\vec{z}'_{k-1}$ normally but add an error into $\vec{z}'_{k-1}$ by updating $\vec{z}'_{k-1} \leftarrow \vec{z}'_{k-1} + \vec{e}$ where $\vec{e} = (0, \cdots, 0, e, \cdots, 0)$ is a weight-1 error vector with non-zero element at position q. Then P_h computes $[\![\vec{s}]\!]_{k-1} \leftarrow \vec{b}_{k-1}$ normally but update $[\![\vec{s}]\!]_h[q] \leftarrow [\![\vec{s}]\!]_h[q] - e$. In this way, the adversary guesses $q = \pi_h(p)$.

In the above attack, the adversary surrounds the honest participant P_h. When playing the predecessor of P_h, $\mathcal{A}$ corrupts one of P_h's inputs by adding error e. Then, as P_h's post-processor, $\mathcal{A}$ corrupts the message from P_h by subtracting e before processing. For either case in the above, the adversary can learn sensitive information about the secret permutation held by the honest party P_h in successful attacks.

Summary. Prior offline and online attacks prove that [2] is not maliciously secure both in correlation generation and shuffle phases. [2] cannot preserve full privacy due to offline and online selective failure attacks from a malicious sender(s) and the OPM attack from a malicious receiver(s).

G. Proof of Theorem 1

Let $\mathcal{A}$ be any PPT adversary who is allowed to corrupt either the sender or the receiver. We construct a PPT simulator $\mathcal{S}$ that can simulate the adversary's view by accessing the functionality $\mathcal{F}_{\mathsf{OT}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Corrupted receiver. The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ and interacts with the adversary $\mathcal{A}$ as follows:

- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ and receives $\overline{\alpha_i}$ from $\mathcal{A}$ for $i \in [1,h]$. $\mathcal{S}$ computes α from $\{\overline{\alpha_i}\}_{i\in[1,h]}$ and sends α to $\mathcal{F}_{\mathsf{GenV}}$. $\mathcal{S}$ receives $K^* = \{K^{(i)}_{1-\alpha_i}\}_{i\in[1,h]} \cup \{K^{(h+1)}_1\}$ from $\mathcal{F}_{\mathsf{GenV}}$.
- For $i \in [1,h]$, $\mathcal{S}$ defines $s^i_{\alpha_a\cdots\alpha_{i-1}\overline{\alpha_i}} = K^{(i)}_{1-\alpha_i}$ and $s^{h+1}_{\alpha||1} = K^{(h+1)}_1$. $\mathcal{S}$ sets $K^1_{\overline{\alpha_1}} = s^1_{\overline{\alpha_1}}$. Then, for $i \in [2,h+1]$, $\mathcal{S}$ computes $(s^i_{2j}, s^i_{2j+1}) \leftarrow G(s^{i-1}_j)$, for $j \in [0,2^{i-1}] \setminus \{\alpha_1\cdots\alpha_{i-1}\}$.
- For $i \in [1,h+1]$, $\mathcal{S}$ computes $K^i_{\overline{\alpha_i}} \leftarrow \bigoplus_{j\in[0,2^{i-1}]} s^i_{2j+\overline{\alpha_i}}$ where $\alpha_{h+1} = 0$. $\mathcal{S}$ sends $\{K^i_{\overline{\alpha_i}}\}_{i\in[1,h]}$ (as the messages from $\mathcal{F}_{\mathsf{OT}}$) and K^{h+1}_1 to $\mathcal{A}$.
- Set $\gamma_j = s^{h+1}_{2j+1}$ for $j \in [N]$, and compute $\tau = h(\gamma_0, \gamma_1, \cdots, \gamma_{n-1})$. $\mathcal{S}$ sends τ to $\mathcal{A}$.
- $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Below we show the simulated execution is indistinguishable from the real-world execution. First note that the simulator $\mathcal{S}$ receives K^* corresponding to OPV key K held by the honest $\mathtt{S}$, and $\mathcal{S}$ computes $\{K^i_{\overline{\alpha_i}}\}_{i\in[1,h+1]}$ in the exact way as the real-world execution. Similarly, $\mathcal{S}$ can compute all right children in the $h+1$ level using K^*, thus $\mathcal{S}$ can compute the exact τ corresponding to the key K held by the honest sender. Overall, the above simulation is perfectly indistinguishable from real-world execution in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.

Corrupted sender. The simulator $\mathcal{S}$ for corrupted sender $\mathtt{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ and interacts with $\mathcal{A}$ as follows:

- $\mathcal{S}$ receives OT messages $\{(K^i_0, K^i_1)\}_{i\in[h]}$, K^{h+1}_1 and τ from $\mathcal{A}$.
- $\mathcal{S}$ computes $K^* \leftarrow F.\mathsf{Puncture}^*(\{K^i_{\overline{\alpha_i}}\}_{i\in[h+1]}, \alpha||0)$ and $\{s_j(\alpha)\}_{j\in[2n]\setminus\{\alpha||0\}} \leftarrow F'.\mathsf{FullEval}\ (K, \alpha||0)$ for each $\alpha \in [n]$. Set $\gamma_j(\alpha) = s_{2j+1}(\alpha)$ for $j \in [n]$. Compute $\tau'(\alpha) \leftarrow h(\gamma_0(\alpha), \cdots, \gamma_{n-1}(\alpha))$.
- Let I be the set of α such that $\tau'(\alpha) = \tau$, *i.e.*, $I = \{\alpha \in [n] \,|\, \tau'(\alpha) = \tau\}$.
- $\mathcal{S}$ sends $K^* = \{(K^i_0, K^i_1)\}_{i\in[h]} \cup \{(0, K^{h+1}_1)\}$ and I to $\mathcal{F}_{\mathsf{GenV}}$.

The punctured key computed by the functionality is exactly the key computed in real-world execution. Also, it shows in [30] (Proof of Theorem 15, [30]) that except with negligible probability, all choices of α, $\alpha' \in I$ lead to the same vector $\vec{s} = (s_0, \cdots, s_{n-1})$. Following this fact, the receiver aborts in the real protocol execution when $\tau' \neq \tau$, which is exactly the case where $\alpha \notin I$. Therefore, the functionality aborts if and only if the real protocol execution aborts. Overall, the above simulation is perfectly indistinguishable from real-world execution in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.

H. Proof of Theorem 2

For any PPT adversary $\mathcal{A}$, we construct a PPT simulator $\mathcal{S}$ that can simulate the adversary's view by accessing the functionality $\mathcal{F}_{\mathsf{GenM}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. In the following, we prove the security of our protocol for two cases of a malicious sender and a malicious receiver, respectively.

Corrupted sender. $\mathcal{S}$ emulates functionalities $\mathcal{F}_{\mathsf{GenV}}$ and random oracles H_1. $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

- $\mathcal{S}$ receives $\{I_i, K^*_i\}_{i\in[n]}$ from $\mathcal{A}$ by emulating $\mathcal{F}_{\mathsf{GenV}}$. $\mathcal{S}$ checks $\mathsf{Ver}(K^*_i, I_i) = 1$ for $i \in [n]$. If any check fails, $\mathcal{S}$ sends abort to $\mathcal{A}$ and terminates.
- $\mathcal{S}$ receives $\{\omega_j\}_{j\in[n]}$ and τ from $\mathcal{A}$ by emulating $\mathcal{F}_{\mathsf{eq}}$. $\mathcal{S}$ checks local table for simulating random oracle H_1 and finds if there exists an input $\mathbf{C}$ such that $\tau = H_1(\mathbf{C})$. If such $\mathbf{C}$ does not exist or $\{\omega_j\}_{j\in[n]}$ does not consiste with $\mathbf{C}$, $\mathcal{S}$ just aborts.
- From $\{\omega_j\}_{j\in[n]}$, $\mathbf{C}$, and $\{I_i, K^*_i\}_{i\in[n]}$, $\mathcal{S}$ extracts and updates $\mathcal{A}$'s selective failure attack strategy as follows:
  - If $|I_i| \geq 2$, $\mathcal{S}$ can fully recover the i-th row of OPM matrix using the following strategy: $\mathcal{S}$ randomly samples two different indexes $\alpha, \alpha \in I_i$. $\mathcal{S}$ computes two punctured keys $K^*\{\alpha\}, K^*\{\alpha'\}$ by calling
  $$K^*\{\alpha\} \leftarrow F'.\mathsf{Puncture}(K^*, \alpha),$$
  $$K^*\{\alpha'\} \leftarrow F'.\mathsf{Puncture}(K^*, \alpha'),$$
  respectively. $\mathcal{S}$ can recover $\{\mathbf{M}^{(1)}_{i,j}\}_{j\in[n]\setminus\{\alpha\}}$ using $K^*\{\alpha\}$ and $\mathbf{M}^{(1)}_{i,\alpha}$ using $K^*\{\alpha'\}$.
  - If $|I_i| = 1$, $\mathcal{S}$ parses $I_i = \{\alpha\}$ and computes a punctured key $K^*\{\alpha\}$. $\mathcal{S}$ computes $\{\mathbf{M}^{(1)}_{i,j}\}_{j\in[n]\setminus\{\alpha\}}$ using K^*_α. Note that in this case, $\mathbf{M}^{(1)}_{i,\alpha}$ is undefined (*i.e.*, $\mathbf{M}^{(1)}_{i,\alpha} = \perp$).
- From $\{\omega_j\}_{j\in[n]}$, $\mathbf{C}$, and $\mathbf{M}^{(1)}$, $\mathcal{S}$ extracts and updates $\mathcal{A}$'s selective failure attack strategy as follows: for row $i \in [n]$ and $j \in [n]$ such that $\mathbf{C}_{i,j} \neq \perp$, if $\mathbf{C}_{i,j} \neq \mathbf{M}^{(1)}_{i,j}$, $\mathcal{S}$ updates $I_i \leftarrow I_i \cap \{j\}$.
- $\mathcal{S}$ forwards $\{I_i, K^*_i\}_{i\in[n]}$ to functionality $\mathcal{F}_{\mathsf{GenM}}$. If $\mathcal{F}_{\mathsf{GenM}}$ returns abort, $\mathcal{S}$ sends abort to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends OK to $\mathcal{A}$ by emulating $\mathcal{F}_{\mathsf{eq}}$.
- $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We first show the view simulated by $\mathcal{S}$ is indistinguishable from real protocol execution. The only message from $\mathtt{R}$ to $\mathtt{S}$ is τ. In the real world, the adversary receives the same τ as he committed if 1) the adversary honestly follows the protocol; 2) the adversary performs a selective failure attack and succeeds in the attack. In either case, sending τ back to $\mathcal{A}$ in the ideal world is indistinguishable from real-world execution.

The remaining question is to show that the ideal world aborts with indistinguishable probability as the real-world execution. Since the protocol is designed in $\mathcal{F}_{\mathsf{GenV}}$-hybrid model, the simulator $\mathcal{S}$ has already obtained $\mathcal{A}$'s selective failure attack strategy to each row of the generated OPM. $\mathcal{A}$ may update its strategy in the OPM check protocol. The only possible approach for $\mathcal{A}$ is adding errors into $\{\omega_j\}_{j\in[n]}$. The simulator has to extract $\mathcal{A}$'s selective failure strategy in the OPM check and updates $\{I_i\}_{i\in[n]}$ correspondingly. Note that $\mathcal{S}$ has already obtained all OPV keys from $\mathcal{A}$, thus $\mathcal{S}$ can fully determine row i of the OPM $\mathbf{M}^{(1)}$ with $|I_i| \neq 1$, and $n-1$ elements for row i with $|I_i| = 1$. This means $\mathcal{S}$ can recover almost all elements except these undefined ones (they are at most n). $\mathcal{S}$ can extract $\mathbf{C}$ that $\mathcal{A}$ queries random oracle

H_1 and compares with $\mathbf{M}^{(1)}$ to extract whether $\mathcal{A}$ updates its selective failure attack in the OPM check phase. First, $\mathcal{S}$ checks whether $\{\omega_j\}_{j\in[n]}$ is consistent with $\mathbf{C}$. In the real world, this corresponds to the fact that $\mathcal{A}$ sends inconsistent messages with its committed matrix. In the real world, the protocol would abort except due to collusion from H_1. In the ideal world, $\mathcal{S}$ always aborts when seeing such inconsistency. The simulation is computationally indistinguishable from real protocol execution due to the collusion resistance of H_1. Second, if $\{\omega_j\}_{j\in[n]}$ is consistent with $\mathbf{C}$, the simulator can extract $\mathcal{A}$'s strategy using $\mathbf{C}$ and its recovered $\mathbf{M}$. In particular, for any $i, j \in [n]$ with $\mathbf{C}_{i,j} \neq \perp$, $\mathcal{S}$ checks whether $\mathbf{C}_{i,j} \neq \mathbf{M}_{i,j}$. If true, $\mathcal{S}$ updates $I_i \leftarrow I_i \cap \{j\}$. In this manner, $\mathcal{S}$ fully extracts $\mathcal{A}$'s updated attack strategy in the OPM check phase, and $\mathcal{S}$ can simply forward $\{I_i, K_i^*\}_{i\in[n]}$ to functionality $\mathcal{F}_{\mathsf{GenM}}$. Since $\mathcal{S}$ fully extracts $\mathcal{A}$'s selective failure attack strategy, the ideal world will abort with the same probability as the real protocol. Overall, the simulator aborts with computationally indistinguishable probability as the real-world protocol does.

Corrupted receiver. $\mathcal{S}$ emulates functionalities $\mathcal{F}_{\mathsf{GenV}}$ and random oracles H_1. $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

- $\mathcal{S}$ defines $\mathsf{abort} \leftarrow 0$.
- For $i \in [n]$, $\mathcal{S}$ receives $\mathcal{A}$'s input $\pi(i)$ to $\mathcal{F}_{\mathsf{GenV}}$. If n punctured points correspond to a valid permutation $\pi \in \mathbf{S}_n$, $\mathcal{S}$ forwards π to $\mathcal{F}_{\mathsf{GenM}}$ and receives $\{K_i^*\}_{i\in[n]}$ from $\mathcal{F}_{\mathsf{GenM}}$. Otherwise, $\mathcal{S}$ sets $\mathsf{abort} \leftarrow 1$ and samples $K_i \xleftarrow{\$} \{0,1\}^\kappa$ and produces punctured key $K_i^* \leftarrow F'.\mathsf{Puncture}(K_i, \pi(i))$.
- $\mathcal{S}$ samples $(\omega_0, \omega_1, \cdots, \omega_{n-1}) \xleftarrow{\$} (\{0,1\}^\kappa)^n$ and returns these values to $\mathcal{A}$.
- $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{eq}}$ and receives $\widetilde{\tau}$. $\mathcal{S}$ recomputes τ from $\{K_i^*\}_{i\in[n]}$, $\{\omega_i\}_{i\in[n]}$ and π. $\mathcal{S}$ checks whether the recomputed value matches the one sent from $\mathcal{A}$. If false, $\mathcal{S}$ sets $\mathsf{abort} \leftarrow 1$. $\mathcal{S}$ sends the τ back to $\mathcal{A}$.
- $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The simulated view is indistinguishable from a real protocol execution, and the simulation aborting is negligible close to a real protocol execution. First note when π is not a valid permutation, the simulator generates PPRF keys for $\mathcal{A}$ in the same way as the ideal functionality $\mathcal{F}_{\mathsf{GenV}}$ does, thus the simulation for OPV key generation is indistinguishable from $\mathcal{A}$ no matter π is valid or not.

Another difference is from the way of generating $\{\omega_i\}_{i\in[n]}$. In real protocol execution, $\{\omega_i\}_{i\in[n]}$ is computed from the matrix recovered from n OPV master keys, while in our simulation $\mathcal{S}$ just samples these values randomly. By a hybrid argument, one can conclude that the simulated view is indistinguishable from real protocol execution from the selective security of F'. In particular, we show that any PPT adversary that distinguishes the two executions with probability ϵ can be used to win the PPRF selective security with advantage $\frac{\epsilon}{n}$ as follows: Construct a sequence of hybrid games $(\mathsf{H}_0, \mathsf{H}_1, \cdots, \mathsf{H}_n)$, where H_i represents that the first i protocol messages are randomly sampled, and the remaining $n-i$ values are computed from true PPRF outputs. Obviously, H_0 is the real execution and H_n denotes the simulated execution. If the adversary can distinguish H_0 and H_n with advantage ϵ, then it follows that there exists i such that $\mathsf{Pr}[\mathsf{H}_i] - \mathsf{Pr}[\mathsf{H}_{i+1}] \geq \epsilon/n$, which wins the selective security of F' with advantage ϵ/n.

It is necessary to argue that $\mathcal{S}$ aborts with the same probability as the real-world execution. In the simulation, $\mathcal{S}$ aborts if 1) π is not well-formed or 2) $\mathcal{A}$ does not send back the correct $\widetilde{\tau}$ under the condition that π is well-formed. Case 2 is easy to simulate. For case 1, $\mathcal{A}$ cannot compute the correct $\widetilde{\tau}$ in the real protocol because $\mathcal{A}$ must have missed at least two elements for some columns of its local OPM matrix. In this case, $\mathcal{A}$ can only use a random element for the equality check in the hope of passing the check by chance. However, this is infeasible from the security of PPRF and H_1. In the ideal world, the simulator always aborts. Thus, the simulation is computationally indistinguishable from real-world execution.

I. Proof of Theorem 3

Let $\mathcal{A}$ be the adversary who is allowed to corrupt either S or R. We construct a PPT simulator $\mathcal{S}$ that can simulate the adversary's view by accessing the functionality $\mathcal{F}_{\mathsf{GenM}}$. In the cases where $\mathcal{S}$ aborts or terminates the simulation, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Corrupted S. The offline phase is easy to simulate: the simulator uses the existing simulation strategy for OPM generation and sends a randomly sampled seed $s \in \{0,1\}^\kappa$ back to $\mathcal{A}$. In the following, we only focus on the online phase simulation. For simplicity, we assume the simulator sees the shares held by the adversary.[2]

- $\mathcal{S}$ samples B randomly sampled $\{\vec{\delta}^{(i)}\}$ and sends them to $\mathcal{A}$.
- $\mathcal{S}$ receives $\mathcal{A}$'s messages in MAC check protocol. $\mathcal{S}$ can check whether $\mathcal{A}$ sends the correct messages since $\mathcal{S}$ has all the necessary information held by S. Therefore, if $\mathcal{A}$ sends the incorrect message in the MAC check, $\mathcal{S}$ just aborts.

The view from protocol messages for $\mathcal{A}$ is indistinguishable from the real protocol execution. First, since $\mathcal{S}$ works in exactly the same way as $\mathcal{F}_{\mathsf{GenM}}$ does, the aborting probability from $\mathcal{F}_{\mathsf{GenM}}$ is indistinguishable from real-protocol execution. The simulation for the online phase is also straightforward because $\mathcal{S}$ has all the shares held by $\mathcal{A}$. This means $\mathcal{S}$ knows $\mathcal{A}$ deviates the protocol by checking whether $\mathcal{A}$ sends the correct protocol messages. Therefore, $\mathcal{S}$ can always abort with indistinguishable probability as the real protocol does.

Corrupted R. The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{GenM}}$ and interacts with $\mathcal{A}$ as follows:

- Whenever simulating $\mathcal{F}_{\mathsf{GenM}}$, $\mathcal{S}$ receives $\{K_i, I_i\}_{i\in[n]}$ from the adversary $\mathcal{A}$. $\mathcal{S}$ randomly samples a permutation $\pi \in \mathbf{S}_n$ and checks $\pi(i) \in I_i$ and $\mathsf{Ver}(K_i^*, I_i) = 1$ for all $i \in [n]$. If any check fails, $\mathcal{S}$ sends abort to $\mathcal{A}$.
- In the online phase, $\mathcal{S}$ receives $\{\vec{\delta}^{(i)}\}_{i\in[n]}$ from $\mathcal{A}$. Now, $\mathcal{S}$ can check whether $\mathcal{A}$ adds error $\vec{e}$ into $\{\vec{\delta}^{(i)}\}_{i\in[n]}$. $\mathcal{S}$ then uses π_i to compute $\pi_i(\vec{e})$, and uses $\vec{b}^{(i)} - \pi_i(\vec{e})$ to recompute $\mathcal{A}$'s MAC check share. If the share does not match the MAC check share from $\mathcal{A}$, $\mathcal{S}$ aborts. If $\mathcal{A}$ behaves honestly or

[2] In the proof of SPDZ protocol, the simulator can learn the shares of corrupted parties by emulating the SPDZ preprocessing functionality. For example, the simulator for the proof of [35] maintains a database CS to store the sums of shares of corrupted parties, which is computed by summing up each corrupted party's share extracted by the simulator. In the following proof, we will assume this without further elaboration.

$\mathcal{A}$ sends the correct MAC check message $[\![m]\!]_1$, $\mathcal{S}$ sends $-[\![m]\!]_1$ to $\mathcal{A}$.

In the above simulation, $\mathcal{S}$ works exactly as $\mathcal{F}_{\mathsf{GenM}}$ works, thus this part of view simulation is indistinguishable from real-world execution. Second, $\mathcal{S}$ can abort with indistinguishable probability as the real protocol execution by checking whether $\mathcal{A}$ sends the desired message in MAC check. For this part, $\mathcal{S}$ can do the check since it has all available information about $\mathcal{A}$, thus $\mathcal{S}$ can recompute by itself to see whether the simulation should abort or proceed. Due to repeated execution, when $\mathcal{A}$ passed the check, no matter whether he honestly followed the protocol or successfully attacked some of the permutations, the composited permutation is still random in the view of $\mathcal{A}$. However, this holds with statistical error $2^{-\lambda}$ as we shown in §B.