Scalable Collaborative zk-SNARK: Fully Distributed Proof Generation and Malicious Security

Xuanming Liu [*,‡], Zhelei Zhou [*,‡,§], Yinghao Wang[‡], Bingsheng Zhang[‡,§], and Xiaohu Yang [†,‡]

[‡]Zhejiang University, The State Key Laboratory of Blockchain and Data Security
[§]ZJU-Hangzhou Global Scientific and Technological Innovation Center

February 1, 2024

Abstract

The notion of collaborative zk-SNARK is introduced by Ozdemir and Boneh (USENIX 2022), which allows multiple parties to jointly create a zk-SNARK proof over distributed secrets (also known as the witness). This approach ensures the *privacy* of the witness, as no corrupted servers involved in the proof generation can learn anything about the honest servers' witness. Later, Garg et al. continued the study, focusing on how to achieve faster proof generation (USENIX 2023). However, their approach requires a powerful server that is responsible for the most resource-intensive computations and communications during the proof generation. This requirement results in a scalability bottleneck, making their protocols unable to handle large-scale circuits.

In this work, we address this issue by lifting a zk-SNARK called Libra (Crypto 2019) to a collaborative zk-SNARK and achieve a *fully distributed* proof generation, where all servers take roughly the same portion of the total workload. Further, our protocol can be adapted to be secure against a malicious adversary by incorporating some verification mechanisms. With 128 consumer machines and a 4Gbps network, we successfully generate a proof for a data-parallel circuit containing 2^{23} gates in merely 2.5 seconds and take only 0.5 GB memory for each server. This represents a $19\times$ speed-up, compared to a local Libra prover. Our benchmark further indicates an impressive $877\times$ improvement in running time and a $992\times$ enhancement in communication compared to the implementation in previous work. Furthermore, our protocol is capable of handling larger circuits, making it scalable in practice.

[*]The co-first authors.
[†]The corresponding author: Xiaohu Yang, email: yangxh@zju.edu.cn.

Contents

1 Introduction

The cryptographic primitive, known as *Zero-Knowledge Succinct Non-interactive Arguments of Knowledge* (zk-SNARK), allows a prover to produce short proofs that convince a verifier of the knowledge of a witness attesting the validity of an NP relation, without revealing any extra information about the underlying witness. This technique has broad applications in various scenarios, such as blockchain [XZC+22, But21, But22] and verifiable machine learning [ZFZS20, LXZ21]. However, a significant limitation of existing zk-SNARKs is that the entire computational burden of proof generation falls on a single prover. This procedure can be notably time-consuming and demands significant memory, especially in the case of large-scale circuits.

A natural solution to this problem involves distributing the proof generation process across multiple servers, where each server is responsible for only a portion of the total computation. This method has been explored in various studies [WZC+18, XZC+22, LXZ+23]. However, these works assume that the servers are "harmless", allowing for the direct disclosure of the witness to them. This assumption may not always be adopted, especially in situations where the witness is sensitive and should be kept from the servers.

To protect privacy of the witness, Ozdemir and Boneh [OB22] introduced an innovative framework known as *collaborative zk-SNARK*. This framework enables multiple servers, each holding parts of the witness, to collaboratively generate a proof without revealing the witness to others. This concept was further developed by Garg et al. [GGJ+23], who proposed a framework called *zk-SNARK-as-a-Service* (zkSaaS), primarily focusing on the outsourcing scenario, where a client wishes to outsource the proof generation task to a group of servers without revealing the witness. Notice that, in both [OB22] and [GGJ+23], the authors employ secret sharing techniques to prevent the corrupted servers from knowing anything about the honest servers' private input. Central to their works is a secure Multi-Party Computation (MPC) protocol, letting the servers collaboratively compute the proof. Both [OB22] and [GGJ+23] lay a solid foundation for collaborative zk-SNARK; however, there is an important issue being unsolved, and we list it in the following.

Lack of fully distributed proof generation. In [OB22], multiple servers are necessitated to execute a generic MPC protocol [DPSZ12, GSZ20], each of them undertakes slightly larger workload than the single prover in the original zk-SNARK. The efficiency is improved by Garg et al. [GGJ+23], as they claimed that using 128 servers to collaboratively generate a proof can achieve a better efficiency than a local prover, for zk-SNARKs called Groth16 [Gro16] and Plonk [GWC19]. However, the proof generation in [GGJ+23] is only *partially distributed*, as there is a king server that undertakes larger computationally intensive tasks and requires much larger memory resources than other normal servers. Jumping ahead, our benchmark indicates that this requirement becomes a critical bottleneck when scaling to large-scale circuits. How to eliminate the need for such a powerful king server is left as an open question in [GGJ+23].

We find the concept of *fully distributed* proof generation to be of great importance, which means each server incurs equal computation and space complexities, proportional to the total overhead. This property is particularly vital for achieving scalability, as it ensures an even distribution of the workload across all servers and enables the cluster to handle larger circuits. Since the prior works cannot achieve this property, we here ask our main research question:

> *Can we achieve fully distributed proof generation for a collaborative zk-SNARK, where the total workload is evenly distributed among all servers?*

In addition to the above question, we also care about the security model. The work by Garg et al. [GGJ+23] is proven to be secure against a *semi-honest* adversary, where all the servers are assumed to follow the protocol instructions, even if they are corrupted. In contrast, the work by Ozdemir and Boneh [OB22] is secure against a *malicious* adversary, who can let the corrupted servers deviate from the protocol instructions. There is no doubt that assuming the presence of a malicious adversary is more realistic in practice. In this work, we consider both semi-honest security and malicious security, and we focus on the *honest majority* setting as in [GGJ+23], where a minority of the servers can be corrupted.

1.1 Our contributions

We summarize the contributions as follows:

1. **Two fully distributed primitives.** We first design two primitives against a semi-honest adversary: (i) a fully distributed sumcheck protocol, (ii) and a fully distributed polynomial commitment scheme. These are key building blocks for the zk-SNARK we will focus on. For both primitives, we can achieve $O(\frac{n}{N})$ computation and space complexities, where n is the input size and N is the number of servers.

2. **Fully distributed proof generation for** Libra**.** We provide a positive answer to the aforementioned question by adapting a zk-SNARK called Libra [XZZ+19] to a collaborative zk-SNARK. We achieve this by making use of our newly designed primitives. For data-parallel circuits (circuits that contain multiple identical sub-copies) with $|\mathcal{C}|$ gates, we can achieve fully distributed proof generation against a semi-honest adversary, where each server has the same computation and space complexities $O(\frac{|\mathcal{C}|}{N})$.

3. **Achieving malicious security.** We provide some lightweight verification protocols to detect malicious behaviors during the proof generation process. As a result, for data-parallel circuits, the computation complexity of each server in our malicious secure distributed proof generation is $\tilde{O}(\frac{|\mathcal{C}|}{N})$ [1] while the space complexity is the same as the semi-honest protocol.

4. **Implementation and evaluation.** We have conducted a proof-of-concept implementation for our semi-honest protocols to validate our results. Performance evaluations indicate that our protocols are both efficient and scalable:

   - *More efficient than local prover [XZZ+19].* Our protocol achieves a 19× speed-up than a local Libra prover [XZZ+19], when generating a proof for a circuit with 2^{23} gates, utilizing 128 servers.
   - *More efficient than [GGJ+23].* Our protocol makes an improvement of roughly 877× in running time and 992 × in communication over the distributed Plonk [GGJ+23], when generating a proof for a circuit with 2^{23} gates over a 4Gbps network, utilizing 128 servers for both cases.
   - *Less cost than [GGJ+23].* When there are 128 servers collaboratively generating proof for a circuit with 2^{19} gates running over a 4Gbps network, our financial cost is only 7% of that of [GGJ+23].
   - *Handling larger scale circuits than [GGJ+23].* When provided with a powerful server with 96 GB memory and 127 servers with 4 GB memory, the distributed Plonk [GGJ+23] can only handle up to a circuit with 2^{24} gates. In contrast, when provided with 128 servers with only 4 GB memory, our protocol can handle a larger circuit with 2^{27} gates.

1.2 Applications

Here, we explore some potential applications for our collaborative zk-SNARK and we list them in the following.

Distributed proof generation of cross-chain bridges. Cross-chain bridges are crucial in blockchain, enabling asset transfers across different chains. Xie et al. in [XZC+22] introduced zkBridge, a system allowing smart contracts on various blockchains to transfer states securely. Their method involves distributed proof generation for efficiently validating transactions in batches, using data-parallel circuits. However, this process is often centralized, executed by a single entity (with multiple servers); therefore, the transactions are disclosed to the single entity. Our protocols differ by distributing proof generation across multiple entities, enhancing security by reducing single-entity dependence. It matches the efficiency of [XZC+22]'s method while preserving the transactions' privacy. Additionally, it is adaptable to other applications prioritizing decentralization and privacy, like zkRollup [But21] and zkEVM [But22].

Verifiable machine learning. A user can commit to a Machine Learning (ML) model and provide a proof of inference, which is used to verify the accuracy of this model. Several works address various ML models like decision trees [ZFZS20], neural networks [ZCL+21], and convolutional neural networks [LXZ21]. As suggested in [GGJ+23], collaborative zk-SNARK can be a solution to this task, since the circuits for ML inference can be quite large as the ML models grow in size, which may become impractical for a local zk-SNARK prover to prove. Since our protocol has better scalability than [GGJ+23], our protocol can handle larger ML models.

Collaborative auditing for efficient MPC. Our framework also finds an application in Publicly-Auditable MPC (PA-MPC) [BDO14], as highlighted in [OB22]. PA-MPC allows public verification of MPC protocol outcomes. Recent works [BGJK21, EGPS22, GPS22, EGP+23] demonstrates the practicality of multiple parties collectively computing a circuit using efficient MPC protocols. They offer greater online efficiency than traditional protocols like [DPSZ12]. As shown in [OB22], PA-MPC can be obtained by letting the parties run an MPC protocol for computation and invoke the collaborative zk-SNARK for generating proofs that shows the correctness of the computation. Our result provides a compiler for these efficient MPC protocols to achieve PA-MPC.

[1] The $\tilde{O}$ notation ignores logarithmic factors.

1.3 Related works

GKR-based interactive proofs. There are many existing zero-knowledge proof systems, or zk-SNARKs [PHGR13, Gro16, AHIV17, GWC19, MBKM19, XZZ+19, Set20, ZXZS20, WYX+21, XZS22]. Among them, a noticeable line of works are based on the initial work by Goldwasser, Kalai, and Rothblum [GKR08] (here after, GKR), which introduced a doubly-efficient interactive proof system for layered arithmetic circuits. Subsequent developments, such as those in [CMT12, Tha13, WJB+17, ZGK+18], have optimized the prover time for circuits with specific structures. Thaler, in [Tha13], introduced a linear-time sumcheck protocol particularly effective for matrix multiplication tasks. Zhang et al. in [ZGK+17a] extended the GKR protocol into an argument system by integrating it with a polynomial commitment scheme. The work of Xie et al. in [XZZ+19] presented Libra, a zk-SNARK, achieving $O(|\mathcal{C}|)$ prover time for arbitrary layered arithmetic circuits, where $|\mathcal{C}|$ denotes the size of the arithmetic circuit. This framework has been the foundation for subsequent research efforts such as [ZXZS20, ZLW+21, LXZ21, XZC+22], which either enhanced the efficiency of the protocol or developed efficient zero-knowledge argument systems for a range of applications.

In this work, we continue the study of GKR-based zk-SNARK but focus on a different setting: we aim to securely distribute the proof generation phase of zk-SNARK to a set of servers, and try to design an MPC protocol that can be invoked by the servers to jointly generate proof. We require this protocol (i) to be *fully distributed*, meaning each server bears the same computation complexity and space complexity, proportional to the overall overhead, (ii) and to be secure against a *malicious* adversary who can only corrupt a minority of the servers, meaning the adversary can not obtain any additional information about the witness if only a minority of the servers are corrupted.

Distributed proof generation. There are a beautiful line of works in the literature (e.g. [SVdV16, WZC+18, XZC+22, DPP+22, OB22, GGJ+23, CLMZ23, LXZ+23]) focus on how to let a group of servers to jointly generate a zk-SNARK proof. We observe that these works can be divided into two types: (i) they assume the servers are all honest and lets the servers know the witness directly; (ii) they assume the adversary can corrupt some servers and let the servers obtain the secret shares of the witness, so the servers know nothing about the witness. We present more details about these works in the following.

Type I: the servers know the witness. Since generating zk-SNARK proofs for large-scale circuits is quite time-consuming, a lot of works [WZC+18, XZC+22, LXZ+23] have contemplated making the proof generation process distributed, involving multiple servers working in tandem to generate the proof, thereby speeding up the proof generation and reducing the overhead for each server. Their work is orthogonal to ours. They employ certain techniques to universally improve the efficiency of proof generation by assuming all the servers are honest, and thus the witness can be disclosed to the servers directly.

A closely related work is the work by Zhang et al. [XZC+22]. They designed distributed proof generation process based on a GKR-based zk-SNARK called Virgo [ZXZS20]. Their algorithm can generate proofs for multiple identical circuits simultaneously, thus significantly improves efficiency by N times, where N is the number of servers. However, their protocol allows the servers to know the entire witness. Therefore, their application scenario is limited and the security of their protocol would be compromised if an adversary corrupts a portion of servers. In this work, we aim to achieve the similar efficiency improvements while keeping the witness hidden from the servers.

Type II: the servers do not know the witness. Recent years there is a series of works [SVdV16, CLMZ23, DPP+22, OB22, GGJ+23] have discussed the topic that let several servers generate the proof collaboratively, *without* disclosing the whole witness to the servers.

Both [OB22] and [GGJ+23] are representative works in this area. In both works, the process can be divided into two steps: Firstly, each server obtains secret-shared witness, rather than the entire witness. Subsequently, the servers execute some MPC protocols to complete the various modules (e.g., FFT and MSM) in the proof generation process, ultimately obtaining the proof.

The main difference between these two works lies in the motivations. Collaborative zk-SNARK, by Ozdemir and Boneh [OB22], considered the case where each server obtains a different witness, for instance, the i-th server obtains w_i, and the servers wants to jointly compute a proof that satisfies an NP relation $\mathcal{R}(x, w_1, \cdots, w_N) = 1$ where x is the statement and N is the number of servers. They implemented their protocols using the generic MPC protocols in the honest majority setting [GSZ20] or in the dishonest majority setting [DPSZ12]. However, this generic approach cannot improve the efficiency of the proof generation phase.

On the other hand, the work by Garg et al. [GGJ+23] focused on the case where a client wishes to outsource the proof generation process to multiple servers, and they formalized a framework called zkSaaS. They designed some tailored-made MPC protocols for three zk-SNARKs: Groth16 [Gro16], Marlin [CHM+20] and

Plonk [GWC19], and they employed the packed Shamir's secret sharing scheme [FY92] to improve the efficiency. However, their protocols are only proven secure in the *semi-honest* model. Furthermore, in their protocols, there is special server that requires much more computation and space sources than other servers, since they let this special server perform some hard computation tasks. Thus, their protocol cannot be *fully distributed*, and they leaved an open question about how to design a fully distributed protocol for the distributed proof generation. In this work, we aim to (i) provide an efficient solution to the aforementioned open question they leaved in [GGJ+23]; (ii) and make our solution malicious-secure.

We note that, there are some works in the literature [CLMZ23, DPP+22] try to distribute the proof generation in a different setting: they assume that there is a special party (in [CLMZ23], they called it delegator; while in [DPP+22], they called it aggregator) who is always semi-honest, while the rest of the servers can be corrupted by a malicious adversary. The main difference between their works and the zkSaaS [GGJ+23] is that: in zkSaaS, the semi-honest client only sends messages to the servers in the first round, then the client is no longer required to stay online, since the rest of the computation is conducted among the servers; in contrast, in [CLMZ23, DPP+22], their semi-honest special party has to stay online and interact with the servers during the entire computation. In our work, we do not need to assume a special server to be always semi-honest during the proof generation process as in [CLMZ23, DPP+22]. In our malicious secure protocol, any server can be corrupted by the malicious adversary.

1.4 Paper organization

Section 2 introduces the preliminaries. Section 3 provides a technique overview of this work. Section 4 provides formal definitions of collaborative zk-SNARKs and the security model upon which our work is based. The fully distributed sumcheck protocol is detailed in Section 5. In Section 6, we integrate the aforementioned protocol with a fully distributed polynomial commitment to establish a collaborative proof generation process. Section 7 expands these protocols to accommodate the malicious security setting. Finally, Section 8 explores the specifics of our implementation and evaluates the results of our experiments.

2 Preliminaries

We put more preliminaries including zero-knowledge arguments, zk-SNARK, and MPC in Appendix A.

2.1 Notation

In this paper, we use λ to denote the security parameter, and $\mathsf{negl}(\lambda)$ to denote a negligible function in λ. "PPT" stands for probabilistic polynomial time. We use bold letters, e.g., $\boldsymbol{x}$, to denote vectors. For a positive integer $n > 1$, we use $[n]$ to denote the set $\{1, \ldots, n\}$. For positive integers a, b such that $a < b$, we use $[a, b]$ to denote the set $\{a, \ldots, b\}$. We use $[\![\boldsymbol{x}]\!]_d$ to denote a degree-d packed secret sharing of $\boldsymbol{x}$ and may omit the subscript d if the context is clear. Similarly, we use $\langle x \rangle$ to denote a Shamir's secret sharing. Let $\mathbb{F}$ be a large finite field with a prime order such that $|\mathbb{F}|^{-1} = \mathsf{negl}(\lambda)$.

Bilinear Groups. Let $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ denote an efficiently computable and non-degenerate bilinear pairing, such that $e(h_1^\alpha, h_2^\beta) = e(h_1, h_2)^{\alpha\beta}$, for all $\alpha, \beta \in \mathbb{F}_q$, and all $h_1, h_2 \in \mathbb{G}$.

2.2 Polynomial commitment scheme

A polynomial commitment (PC) scheme is a cryptographic primitive that allows a prover to commit to a polynomial f and later open the commitment to a point x and prove that the commitment is indeed a valid evaluation of f at x.

Definition 1. *A* PC *scheme for ℓ-variates polynomials $\mathcal{F}$ is a tuple of PPT algorithms (*PC.Setup, PC.Commit, PC.Open, PC.Verify*):*

- PC.Setup$(1^\lambda, \mathcal{F}) \to$ pp*: It takes as input a security parameter λ, and outputs a public parameter* pp.
- PC.Commit$(f, \mathsf{pp}) \to \mathsf{com}_f$*: It takes as input the ℓ-variates polynomial $f(\boldsymbol{x})$ where $\boldsymbol{x} = (x_1, ..., x_\ell)$, and outputs a commitment* com_f.

- $\mathsf{PC.Open}(f, \boldsymbol{x}, \mathsf{pp}) \to (z, \pi)$: *It evaluates f at a point $\boldsymbol{x}$ and outputs an evaluation $z := f(\boldsymbol{x})$ and a corresponding proof π.*
- $\mathsf{PC.Verify}(\mathsf{com}_f, \boldsymbol{x}, z, \pi, \mathsf{pp}) \to \{0, 1\}$: *It verifies the proof π using $\mathsf{pp}, \mathsf{com}_f, z$ and outputs 1 if the proof is verified.*

The PC scheme satisfies the following properties:

- ***Completeness**. For any polynomial $f \in \mathcal{F}$ and any point $\boldsymbol{x} \in \mathbb{F}^\ell$, if $\mathsf{PC.Setup}(1^\lambda, \mathcal{F}) \to \mathsf{pp}, \mathsf{PC.Commit}(f, \mathsf{pp}) \to \mathsf{com}_f$,*
  $\mathsf{PC.Open}(f, \boldsymbol{x}, \mathsf{pp}) = (z, \pi)$, then the following probability is 1.

$$\Pr\left[\mathsf{PC.Verify}(\mathsf{com}_f, \boldsymbol{x}, z, \pi, \mathsf{pp}) = 1\right]$$

- ***Knowledge soundness**. For any PPT adversary P^*, there exists a PPT extractor $\mathcal{E}$ with access to P^*'s messages during the protocol. If $\mathsf{PC.Setup}(1^\lambda, \mathcal{F}) \to \mathsf{pp}, (z^*, \boldsymbol{x}^*, \mathsf{com}_f^*, \pi^*) \leftarrow \mathsf{P}^*(1^\lambda, \mathsf{pp}), f^* \leftarrow \mathcal{E}^{\mathsf{P}^*}(1^\lambda, \mathsf{pp})$, then the following probability is $\mathsf{negl}(\lambda)$.*

$$\Pr\begin{bmatrix}\mathsf{PC.Verify}(\mathsf{com}_f^*, \boldsymbol{x}^*, z^*, \pi^*, \mathsf{pp}) = 1 \wedge \\ \mathsf{com}^* = \mathsf{PC.Commit}(f^*, \mathsf{pp}) \wedge \\ f^*(\boldsymbol{x}^*) \neq z^*\end{bmatrix}$$

*Informally, a polynomial commitment scheme is **zero-knowledge**, if the commitment and the openings reveal no information about the polynomial f. This property can be achieved by adding randomness [ZGK+17b].*

KZG commitment and its variants. In this work, we mainly focus on a variant of the well-known KZG PC scheme [KZG10, PST13] for *multilinear polynomials*, which are multivariate polynomials whose degree in each variable is at most one (Hereafter, we call it mKZG scheme). Given an ℓ-variate multilinear polynomial $f \in \mathcal{F}$, the multilinear KZG (mKZG) scheme is defined by four algorithms:

- $\mathsf{mKZG.Setup}(1^\lambda, \mathcal{F}) \to \mathsf{pp}$: Sample $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}^\ell$ and output $\mathsf{pp} = \{\{g^{\prod_{i \in W} s_i}\}_{W \in \mathcal{W}_\ell}\}$, where $\mathcal{W}_\ell$ is the collection of all subset of $\{1, ..., \ell\}$. Note that, $\boldsymbol{s}$ is the trapdoor.
- $\mathsf{mKZG.Commit}(f, \mathsf{pp}) \to \mathsf{com}_f$: Output $\mathsf{com}_f = g^{f(\boldsymbol{s})}$.
- $\mathsf{mKZG.Open}(f, \boldsymbol{u}, \mathsf{pp}) \to (z, \pi)$: Evaluate f at $\boldsymbol{u}$ as $z = f(\boldsymbol{u})$. Compute polynomials $\{Q_i\}_{i \in [\ell]}$ that satisfy $f(\boldsymbol{x}) = \sum_{i=1}^{\ell} (x_i - u_i) \cdot Q_i(x_{i+1}, ..., x_\ell) + z$. Finally, output $\pi = \{\pi_i\}_{i \in [\ell]}$, where $\pi_i = g^{Q_i(s_{i+1}, ..., s_\ell)}$.
- $\mathsf{mKZG.Verify}(\mathsf{com}_f, \boldsymbol{u}, z, \pi, \mathsf{pp}) \to \{0, 1\}$: Check $e(\frac{\mathsf{com}_f}{g^z}, g) = \prod_{i=1}^{\ell} e(\pi_i, g^{s_i - u_i})$, and output 1 if and only if the check passes.

Note that the evaluations of a polynomial on the power can be computed efficiently using pp, and $\{Q_i\}_{i \in [\ell]}$ can be computed using polynomial divisions for ℓ times. We kindly refer the readers to [PST13] for the security analysis of the above scheme.

2.3 The GKR protocol

In [GKR08], Goldwasser, Kalai, and Rothblum proposed a protocol, hereafter denoted as the GKR protocol, which enables a prover to convince a verifier that the output of a layered arithmetic circuit is correct. The protocol is founded upon the sumcheck protocol [LFKN90]. In this subsection, we give a brief introduction to the GKR protocol and its applications to zk-SNARKs. Before that, we present an important notion that will be used in the GKR protocol in the following.

2.3.1 Multilinear extension

In the GKR-based protocol, we typically engage the *multilinear extension* of the witness in the computation. In this paper, we define the multilinear extension of a vector $V : \{0, 1\}^\ell \to \mathbb{F}$ as $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ such that $\tilde{V}(\boldsymbol{x}) = V(\boldsymbol{x})$ for any $\boldsymbol{x} \in \{0, 1\}^\ell$. More concretely, $\tilde{V}$ can be expressed as:

$$\tilde{V}(\boldsymbol{x}) = \sum_{\boldsymbol{b} \in \{0,1\}^\ell} (\prod_{i=1}^{\ell} \beta_{b_i}(x_i)) \cdot V(\boldsymbol{b}) \quad , \tag{1}$$

where $\beta_{b_i}(x_i) = (1 - x_i)(1 - b_i) + x_i b_i$ and b_i is i-th bit of $\boldsymbol{b}$. Note that $\tilde{V}$ is a multilinear polynomial.

2.3.2 The sumcheck protocol

Given a multivariate polynomial $f : \mathbb{F}^\ell \rightarrow \mathbb{F}$, the sumcheck protocol [LFKN90] enables the prover to convince the verifier that $H = \sum_{b_1,b_2,...,b_\ell \in \{0,1\}} f(b_1, b_2, ..., b_\ell)$ is correct. In [LFKN90], the authors proposed a protocol that requires ℓ rounds of communication. This protocol is presented in Figure 1. The communication cost and the verification time are both of $O(d \cdot \ell)$, where d represents the degree of the polynomial f. In [Tha13], Thaler introduced an algorithm designed to execute the prover algorithm in $O(2^\ell)$ time. As our work builds upon this algorithm, we have provided a concise overview of it in Section 5.1. A zero-knowledge sumcheck can be constructed by adding masking polynomials, which is described in [CFS17, ZGK⁺17b, XZZ⁺19].

Protocol Π_{sumcheck}

The protocol is an interactive protocol between P and V proceeding in ℓ rounds.

- In the first round, P sends a univariate polynomial $f_1(x_1) \overset{def}{=} \sum_{b_2,...,b_\ell \in \{0,1\}} f(x_1, b_2, ..., b_\ell)$. Then V checks $H \overset{?}{=} f_1(0) + f_1(1)$. V then sends a random value $r_1 \in \mathbb{F}$ to P.
- In the i-th round ($2 \le i \le \ell - 1$), P sends a univariate polynomial $f_i(x_i) \overset{def}{=} \sum_{b_{i+1},...,b_\ell \in \{0,1\}} f(r_1, ..., r_{i-1}, x_i, b_{i+1}, ..., b_\ell)$. Then V checks $f_{i-1}(r_{i-1}) \overset{?}{=} f_i(0) + f_i(1)$. V then sends a random value $r_i \in \mathbb{F}$ to P.
- In the ℓ-round, P sends a univariate polynomial $f_\ell(x_\ell) \overset{def}{=} f(r_1, ..., r_{\ell-1}, x_\ell)$. Then V checks $f_{\ell-1}(r_{\ell-1}) \overset{?}{=} f_\ell(0) + f_\ell(1)$. If the check passes, V generates a random challenge $r_\ell \in \mathbb{F}$ and check that $f_\ell(r_\ell) \overset{?}{=} f(r_1, r_2, ..., r_\ell)$. V gets the evaluation of f on random vector $\boldsymbol{r} = \{r_1, r_2, ..., r_\ell\}$ from an oracle. V accepts the proof if and only if all the checks pass.

Figure 1: The Sumcheck Protocol Π_{sumcheck}

2.3.3 GKR protocol

Using the sumcheck protocol as a building block, the GKR protocol enables a prover to efficiently convince a verifier that the output of a layered arithmetic circuit is correct. Let $\mathcal{C}$ be an arithmetic circuit with depth d over a finite field $\mathbb{F}$. Without loss of generality, we suppose there are S gates in each layer i and let $S = 2^m$ for some integer m, where each gate takes inputs from two gates in the $(i+1)$-layer and outputs a value. Layer-0 is the output layer while layer-d is the input layer. We then define a function $V_i : \{0,1\}^m \rightarrow \mathbb{F}$ that takes input a gate label $\boldsymbol{b} \in \{0,1\}^m$ and returns the value which is the output of the specific gate. With this definition, V_0 corresponds to the output of the circuit and V_d represents the input. We then define the multilinear extension of V_i as $\tilde{V}_i : \mathbb{F}^m \rightarrow \mathbb{F}$ such that $\tilde{V}_i(\boldsymbol{x}) = V_i(\boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^m$. We define two additional functions $add_i, mult_i : \{0,1\}^{3m} \rightarrow \{0,1\}$, referred to as *wiring predicates* in the literature. $add_i(mult_i)$ takes one gate label $\boldsymbol{g} \in \{0,1\}^m$ in layer $i-1$ and two gate labels $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^m$ in layer i, and outputs 1 if and only if the gate corresponding to $\boldsymbol{b}$ is an addition(multiplication) gate that takes the output of gate $\boldsymbol{x}, \boldsymbol{y}$ as input.

With the above definitions, we can express the evaluations of $\tilde{V}_i$ as a summation of evaluations of $\tilde{V}_{i+1}$:

$$\begin{aligned} \alpha_i \tilde{V}_i(\boldsymbol{u}^{(i)}) + \beta_i \tilde{V}_i(\boldsymbol{v}^{(i)}) = \sum_{\boldsymbol{x},\boldsymbol{y} \in \{0,1\}^m} f_i(\tilde{V}_{i+1}(\boldsymbol{x}), \tilde{V}_{i+1}(\boldsymbol{y})) = \\ \sum_{\boldsymbol{x},\boldsymbol{y} \in \{0,1\}^m} (\alpha_i \tilde{add}_{i+1}(\boldsymbol{u}^{(i)}, \boldsymbol{x}, \boldsymbol{y}) + \beta_i \tilde{add}_{i+1}(\boldsymbol{v}^{(i)}, \boldsymbol{x}, \boldsymbol{y}))(\tilde{V}_{i+1}(\boldsymbol{x}) + \tilde{V}_{i+1}(\boldsymbol{y})) \\ + (\alpha_i \tilde{mult}_{i+1}(\boldsymbol{u}^{(i)}, \boldsymbol{x}, \boldsymbol{y}) + \beta_i \tilde{mult}_{i+1}(\boldsymbol{v}^{(i)}, \boldsymbol{x}, \boldsymbol{y}))\tilde{V}_{i+1}(\boldsymbol{x})\tilde{V}_{i+1}(\boldsymbol{y}) \ , \end{aligned} \tag{2}$$

where $\boldsymbol{u}^{(i)}, \boldsymbol{v}^{(i)} \in \mathbb{F}^m$ are random vectors and $\alpha_i, \beta_i \in \mathbb{F}$ are random values. Note here f_i depends on $\alpha_i, \beta_i, \boldsymbol{u}^{(i)}, \boldsymbol{v}^{(i)}$, and we omit the subscripts for easy interpretation.

Then we let P and V execute the sumcheck protocol on Equation 2. At the end of the sumcheck protocol, V receives two claims from P: $\tilde{V}_{i+1}(\boldsymbol{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\boldsymbol{v}^{(i+1)})$. For checking the correctness of these two claims, V then computes the random linear combination like on the left side of Equation 2 and proceeds to the subsequent layer recursively until the input layer. We give a detailed description of the GKR protocol in Protocol 2.

Extending the GKR protocol to a zk-SNARK. The GKR protocol in Figure 2 has no zero-knowledge properties and is not even an argument system. There is a line of work [ZGK$^+$17a, ZGK$^+$17b, XZZ$^+$19, ZXZS20] in the literature that shows how to address these issues. Here we provide a quick recap.

To extend the GKR protocol to an argument system, Zhang et al. [ZGK$^+$17a] proposed an idea that combines a PC scheme with the GKR protocol. More precisely, the prover P initially commits to the multilinear extension of variables in the input layer (denoted as $\tilde{V}_d$) to V using PC.Commit. At the end of the protocol, the verifier V can ask the prover P to open the committed polynomial at randomly selected points by leveraging PC.Open. After checking the validity of the prover's opening messages using PC.Verify, the verifier V can finish the entire GKR protocol using the opened evaluations. To obtain the zero-knowledge property, [XZZ$^+$19] demonstrates that the zero-knowledge property can be achieved by integrating some masking (random) polynomials into the original GKR protocol. Furthermore, [XZZ$^+$19] shows that their approaches can result in a very efficient zk-SNARK called Libra. We refer interested readers to see more details about the GKR-based zk-SNARKs in [XZZ$^+$19, ZXZS20].

Notice that, in this work, we focus on a GKR-based zk-SNARK called Libra [XZZ$^+$19], whose prover time is $O(|\mathcal{C}|)$, where $|\mathcal{C}|$ is the circuit size.

Protocol Π_{GKR}

Let $\mathbb{F}$ be a finite field. Let $\mathcal{C} : \mathbb{F}^S \to \mathbb{F}^S$ be a d-depth layered arithmetic circuit. P wants to convince that $\mathcal{C}$ is satisfied by its input and output.

1. V choose a random vector $\boldsymbol{g} \in \mathbb{F}^m$ and sends in to P.
2. P and V run a sumcheck protocol on
$$\tilde{V}_0(\boldsymbol{g}) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^m} \tilde{add}_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})(\tilde{V}_1(\boldsymbol{x}) + \tilde{V}_1(\boldsymbol{y})) + \tilde{mult}_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})\tilde{V}_1(\boldsymbol{x})\tilde{V}_1(\boldsymbol{y})$$
At the end of the protocol, V needs to check the correctness of two claims $\tilde{V}_1(\boldsymbol{u}^{(1)})$ and $\tilde{V}_1(\boldsymbol{v}^{(1)})$ from P. V proceeds using a random linear combination like in Step 3.
3. In each layer $i = 1, ..., d-1$:
   - V randomly select $\alpha_i, \beta_i \in \mathbb{F}$ and sends them to P.
   - P and V run a sumcheck protocol on Equation 2:
$$\alpha_i\tilde{V}_i(\boldsymbol{u}^{(i)}) + \beta_i\tilde{V}_i(\boldsymbol{v}^{(i)}) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^m} f_i(\tilde{V}_{i+1}(\boldsymbol{x}), \tilde{V}_{i+1}(\boldsymbol{y}))$$
   - At the end of the sumcheck protocol, V needs to check the correctness of two claims $\tilde{V}_{i+1}(\boldsymbol{u}^{(i+1)})$ and $\tilde{V}_{i+1}(\boldsymbol{v}^{(i+1)})$ from P. To validate these claims, V conducts a recursive check on the subsequent layer.
4. At the input layer-d, V has two claims $\tilde{V}_d(\boldsymbol{u}^{(d)})$ and $\tilde{V}_d(\boldsymbol{v}^{(d)})$ from P. To check the claims, V uses the multilinear extension of the inputs and evaluates on $\boldsymbol{u}^{(d)}, \boldsymbol{v}^{(d)}$. V outputs 1 if and only if the claims are correct.

Figure 2: The Protocol Π_{GKR}

2.4 Packed secret sharing

In this work, we focus on the *Packed Secret Sharing* (PSS) scheme [FY92], which allows a dealer to pack several secrets at one time and distribute them among multiple parties, using the well-known *Shamir's secret sharing* scheme [Sha79].

Suppose $\boldsymbol{x} = \{x_1, ..., x_k\}$ is a vector of k secrets to be shared. The dealer can pick a *degree-d* ($d \geq k-1$) polynomial f ($d \geq k-1$) such that $f(-i+1) = x_i$ for $i \in [k]$. Each share is then calculated as $f(i)$ and sent to the i-th party for $i \in [N]$. Any $d+1$ parties can reconstruct $\boldsymbol{x}$ by Lagrange interpolation.

In this work, we use $[\![\boldsymbol{x}]\!]_d$ to denote a degree-d packed secret sharing of $\boldsymbol{x}$ and may omit the subscript d if the context is clear. We use $\langle x\rangle$ to denote a regular threshold sharing (i.e., Shamir's secret sharing). We recall two properties of PSS in the following. More concretely, for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$:

- Linear homomorphism: $[\![\boldsymbol{x}+\boldsymbol{y}]\!]_d = [\![\boldsymbol{x}]\!]_d + [\![\boldsymbol{y}]\!]_d$.

- Multiplication: $[\![\boldsymbol{x} * \boldsymbol{y}]\!]_{2d} = [\![\boldsymbol{x}]\!]_d \cdot [\![\boldsymbol{y}]\!]_d$, where $*$ represents a coordinate-wise multiplication.

The first property implies that, for all scalar $c \in \mathbb{F}$, all parties can locally compute $[\![\boldsymbol{c} * \boldsymbol{x}]\!]_d = c \cdot [\![\boldsymbol{x}]\!]_d$, where $\boldsymbol{c} := (c, c, \dots, c) \in \mathbb{F}^k$. We also note that, if we denote by t the number of corrupted parties, then $t = d - k + 1$; in other words, the PSS scheme above is secure against $d - k + 1$ corrupted parties.

3 Technical Overview

The primary goal of this work is to eliminate the requirement of a powerful server in the proof generation process, thereby ensuring an even distribution of workload across all servers. This enables the proof generation to scale effectively to larger circuits.

SIMD-friendly zk-SNARKs. Similar to [GGJ+23], our main idea is to utilize the *Single Instruction, Multiple Data* (SIMD) structure in the computation of zk-SNARK proofs, and employ the PSS technique [FY92] to enhance the efficiency. We note that the primary reason for the partially distributed proof generation in [GGJ+23], stems from a primitive called Fast Fourier Transform (FFT), which is a key component of some zk-SNARKs such as Groth16 [Gro16] and Plonk [GWC19]. In [GGJ+23], the authors find it is very hard to evenly distribute the computation task of FFT; therefore, they design a protocol for *partially* distributing FFT, in which there is a powerful server that requires much more computation and memory resources than others.

In contrast, in this work, we focus on distributing a particular category of zk-SNARKs which avoids the use of FFT, i.e., GKR-based zk-SNARKs. Our key observation is that: the sumcheck protocol [LFKN90], which is a fundamental building block of GKR-based zk-SNARKs [WJB+17, WTs+18, ZGK+18, XZZ+19, ZXZS20], is somewhat "SIMD-friendly".

Fully distributed sumcheck protocol. As described in Section 2.3.2, the sumcheck protocol allows a prover to convince a verifier of a claim $H = \sum_{\boldsymbol{b} \in \{0,1\}^\ell} f(\boldsymbol{b})$ is correct, with $f : \mathbb{F}^\ell \to \mathbb{F}$ being an ℓ-variate polynomial. Furthermore, in this work, we require f to be a *multilinear polynomial*.

The bookkeeping table in sumcheck protocol. In [Tha13], Thaler proposed an algorithm for the sumcheck protocol with linear prover time. The core of this algorithm is letting the prover compute a *bookkeeping table*, consisting of ℓ rows with $2^{\ell-i+1}$ entries in the i-th row. More precisely, the prover begins by initializing the first 2^ℓ entries of the bookkeeping table with f's evaluations on a hypercube $\{0,1\}^\ell$. The protocol consists of ℓ rounds interaction between the prover and the verifier. During the i-th round of the protocol, the prover utilizes the entries in i-th row to deduce a univariate polynomial $f_i(x_i)$ and send it to the verifier. The verifier then checks $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$ holds; if so, the verifier returns a random challenge $r_i \in \mathbb{F}$. Subsequently, the prover computes the $2^{\ell-i}$ entries for the next row by performing linear combinations on the current $2^{\ell-i+1}$ entries as follows:

$$\begin{aligned} f(r_1, ..., r_{i-1}, r_i, \boldsymbol{b^{(i)}}) &= (1 - r_i) \cdot f(r_1, ..., r_{i-1}, 0, \boldsymbol{b^{(i)}}) \\ &\quad + r_i \cdot f(r_1, ..., r_{i-1}, 1, \boldsymbol{b^{(i)}}) \ , \end{aligned} \tag{3}$$

for all $\boldsymbol{b^{(i)}} \in \{0,1\}^{\ell-i}$. This process continues for ℓ rounds and finally, the bookkeeping table is filled. We give an example in part (a) of Figure 3 and a detailed description in Section 5.1.

Distributing the bookkeeping table. We note it is possible to distribute the computational workload required for the bookkeeping table evenly among multiple servers. We observe that running Equation 3 for $2^{\ell-i}$ times can be efficiently executed in a SIMD fashion, with the assistance of PSS. For instance, assuming there are $n = 2^\ell$ entries, denoted $\boldsymbol{x} = (x_1, ..., x_n)$. These entries can be segmented into $\frac{n}{k}$ vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_{\frac{n}{k}}$, where k is the packing factor. Assuming each server are provided with the packed shares $\{[\![\boldsymbol{x_j}]\!]\}_{j \in [\frac{n}{k}]}$, then Equation 3 can be rewritten as:

$$\forall j \in [1, \frac{n}{2k}] : \ [\![\boldsymbol{y_j}]\!] = (1 - r_i) \cdot [\![\boldsymbol{x_j}]\!] + r_i \cdot [\![\boldsymbol{x}_{\boldsymbol{j}+\frac{\boldsymbol{n}}{\boldsymbol{2k}}}]\!] \ ,$$

where $\{[\![\boldsymbol{y_j}]\!]\}_{j \in [\frac{n}{2k}]}$ are the packed shares of entries in the next row. This process, which can be repeated for multiple rounds, enhances the efficiency of computing the bookkeeping table by a factor of $k = O(N)$.

However, a challenge arises when $k > 2$; in such cases, by a certain round (i.e., the $\log_2 \frac{n}{k}$-th round), each server holds only a single share $[\![\boldsymbol{x}]\!]$, which impedes further computation. To overcome this, for each remaining round, we suggest that servers first perform a carefully designed permutation on the single share $[\![\boldsymbol{x}]\!]$. This enables each server to obtain an additional share $[\![\hat{\boldsymbol{x}}]\!]$, where entries in the same position within $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$ are aligned according to the linear combination. Then each server can locally compute $[\![\boldsymbol{y}]\!] = (1 - r_i) \cdot [\![\boldsymbol{x}]\!] + r_i \cdot [\![\hat{\boldsymbol{x}}]\!]$, where $[\![\boldsymbol{y}]\!]$ is the packed shares of entries in the next row. To facilitate the permutation, we introduce a sub-protocol known as the *PSS permutation*, adapted from [GPS21a]. Combining these two phases, the task of computing the bookkeeping table can be distributed among the servers evenly. As a result, we achieve a fully distributed sumcheck protocol, where each server has the same computation and space complexities $O(\frac{n}{N})$, where n is the input size and N is the number of servers.

Fully distributed PC scheme. In the GKR-based zk-SNARKs, the prover uses the PC scheme to commit to the witness, which can be transformed into the form of a multilinear polynomial. In this work, we try to distribute the mKZG PC scheme, which is described in Section 2.2 and is used in a GKR-based zk-SNARK called Libra [XZZ+19].

Distributing the commitment generation. Recall that, in mKZG PC scheme, for an ℓ-variate multilinear polynomial f, the prover will generate a commitment as $\mathsf{com}_f = g^{f(\boldsymbol{s})}$, utilizing the public parameter pp.

In order to distribute the above commitment generation among a group of servers, we leverage the property of multilinear polynomials. Let $\boldsymbol{x}$ represent the evaluations of f on the hypercube $\{0,1\}^\ell$. Assuming that each server is provided with the packed shares $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$, we observe that: $g^{f(\boldsymbol{s})} = \prod_{\boldsymbol{b}\in\{0,1\}^\ell} g^{\prod_{i=1}^{\ell}\beta_{b_i}(s_i)\cdot f(\boldsymbol{b})}$, where $\beta_{b_i}(s_i) = (1-s_i)(1-b_i) + s_i b_i$, and each $f(\boldsymbol{b})$ corresponds to an element in $\boldsymbol{x}$. The above equation can be computed in a distributed manner using a procedure called distributed Multi-Scalar Multiplication (dMSM), as described in [GGJ+23]. More concretely, dMSM enables the servers to compute $\prod_{i=1}^{n} A_i^{b_i}$, given the packed shares of a set of elliptic curve points $A_1, ..., A_n$ and a set of scalars $b_1, ..., b_n$. Each server is required to perform only $O(\frac{n}{N})$ group exponentiations. It is easy to see that, if the public parameter can be generated in an appropriate form of packed shares, then the commitment above can be efficiently computed by the servers collectively.

Distributing the opening proof generation. Given an evaluation point $\boldsymbol{u} \in \mathbb{F}^\ell$ and the evaluation result $y \in \mathbb{F}$, the prover can generate an opening proof showing $y = f(\boldsymbol{u})$ indeed valid. In mKZG scheme, this process typically involves using FFT to perform multiple polynomial divisions by $(x_i - u_i)$, resulting in a series of quotient polynomials $Q_i(\boldsymbol{x})_{i\in[\ell]}$, and generate proofs as $\pi_i = g^{Q_i(s_{i+1},...,s_\ell)}$, for $i \in [\ell]$. However, as previously mentioned, it is not clear how to distribute the workload of FFT evenly among the servers. Therefore, we propose an alternative method to compute the opening proof, which avoids the need for FFT.

Consider the division of a multilinear polynomial $f(x_i, ..., x_\ell)$ by $(x_i - u_i)$ to obtain the quotient polynomial $Q(x_{i+1}, ..., x_\ell)$ and the remainder polynomial $R(x_{i+1}, ..., x_\ell)$, we observe that the evaluations of Q and R on $\boldsymbol{b} \in \{0,1\}^{\ell-i}$ can be expressed as: $Q(\boldsymbol{b}) = f(1,\boldsymbol{b}) - f(0,\boldsymbol{b})$ and $R(\boldsymbol{b}) = (1-u_i)\cdot f(0,\boldsymbol{b}) + u_i \cdot f(1,\boldsymbol{b})$. This relationship, similar to Equation 3, also exhibits a SIMD structure. Assuming the servers possess the packed shares of the evaluations of f, they can also utilize the SIMD structure to locally compute the packed shares for each Q_i, proceeding round-by-round in a manner akin to the sumcheck protocol. Finally, in order to generate the opening proofs, the servers can employ the dMSM protocol again, which is similar to the distributed commitment generation. As a result, we achieve a fully distributed PC scheme for multilinear polynomials, where each server has the same computation and space complexities $O(\frac{n}{N})$, where n is the input size and N is the number of servers.

Fully distributed proof generation for Libra **[XZZ+19].** We show how to transform an existing GKR-based zk-SNARK called Libra [XZZ+19], to a collaborative zk-SNARK. To accomplish this, we integrated the distributed sumcheck protocol and the distributed polynomial commitment scheme. We notice that, for *data-parallel circuits*, we can achieve the fully distributed proof generation for Libra, each server has the same computation and space complexities $O(\frac{|\mathcal{C}|}{N})$, where $|\mathcal{C}|$ is the circuit size and N is the number of servers.

Notice that, up to now, all the protocols mentioned above are secure against a semi-honest adversary. In order to achieve malicious security, our approach is to add some lightweight verification protocols to detect the potential malicious behaviors caused by the adversary. As a result, for data-parallel circuits, the computation complexity of each server in our malicious secure distributed proof generation is $\tilde{O}(\frac{|C|}{N})$ while the space complexity is the same as the semi-honest protocol. The details are put in Section 7.

4 Collaborative zk-SNARK

In this section, we revisit the notion of collaborative zk-SNARK [OB22], which enables multiple servers to collaboratively generate a proof for a given NP relation. At the heart of this process is an MPC protocol Π, allowing several servers to efficiently collaborate on executing the prover algorithm of an existing zk-SNARK. In the following, we present the formal definition of collaborative zk-SNARK, adapted from [OB22].

Definition 2. *Let N represent the number of servers, and $\mathsf{S}_1, \ldots, \mathsf{S}_N$ be the servers. Let* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a zk-SNARK for some NP relation $\mathcal{R}$. Let x be the public input and $\boldsymbol{w}$ be the witness. For each server S_i, where $i \in [N]$, $\boldsymbol{w}_i$*

is the packed shares of $\boldsymbol{w}$ received by S_i. A collaborative zk-SNARK for an NP relation $\mathcal{R}$ consists of a tuple of algorithms $(\mathsf{Setup}, \Pi, \mathsf{Verify})$*, where:*

- $\mathsf{Setup}(1^\lambda, \mathcal{R}) \to \mathsf{pp}$*: This is the same as the setup algorithm* Setup *of the underlying zk-SNARK. It takes the security parameter λ and the NP relation $\mathcal{R}$ as inputs and outputs the public parameter* pp.
- $\Pi(\mathsf{pp}, x, \boldsymbol{w}_1, ..., \boldsymbol{w}_N) \to \pi$*: This is an MPC protocol among N servers, and it computes the prover algorithm* Prove *of the underlying zk-SNARK. Given the public parameters* pp*, the public statement x and the servers' received packed secret shares $\boldsymbol{w}_1, ..., \boldsymbol{w}_N$, the servers engage in Π and collaboratively generate a proof π.*
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0, 1\}$*: This is the same as the verification algorithm* Verify *of the underlying zk-SNARK. It takes the public parameter* pp*, the statement x, and the proof π as inputs and outputs a bit b indicating acceptance $(b = 1)$ or rejection $(b = 0)$.*

This framework is secure if it satisfies the following properties:

- ***Completeness**: For all $(x, \boldsymbol{w}) \in \mathcal{R}$, the following relation holds:*

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, x, \pi) = 1 : \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R}), \\ \pi \leftarrow \Pi(\mathsf{pp}, x, \boldsymbol{w}_1, ..., \boldsymbol{w}_N) \end{array}\right] = 1 \ .$$

- ***Knowledge Soundness**: For all x, and all sets of PPT algorithms $\vec{\mathsf{S}} = \{\mathsf{S}_1^*, ..., \mathsf{S}_N^*\}$, there exists a PPT extractor $\mathcal{E}$ such that,*

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, x, \pi^*) = 1 : \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R}), \\ \boldsymbol{w}^* \leftarrow \mathcal{E}^{\vec{\mathsf{S}}}(\mathsf{pp}, x), \\ \pi^* \leftarrow \vec{\mathsf{S}}(\mathsf{pp}, x), \\ (x, \boldsymbol{w}^*) \notin \mathcal{R} \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

- ***t-zero-knowledge**: For all PPT adversary $\mathcal{A}$ controlling at most t servers denoted as $\mathcal{C}orr$,* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{R})$*, there exists a simulator $\mathcal{S}$ such that for all $x, \boldsymbol{w}$ (where $b \leftarrow \mathcal{R}(x, \boldsymbol{w}) \in \{0, 1\}$), the following relation holds:*

$$\mathsf{View}_\Pi^\mathcal{A}(x, \boldsymbol{w}) \approx \mathcal{S}(\mathsf{pp}, x, b, \{\boldsymbol{w}_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}) \ .$$

  Here $\mathsf{View}_\Pi^\mathcal{A}(x, \boldsymbol{w})$ denotes the view of $\mathcal{A}$ from the real-world execution of Π and $\mathcal{S}(\mathsf{pp}, x, b, \{\boldsymbol{w}_i\}_{i \in \mathcal{C}orr})$ is the view generated by $\mathcal{S}$ given x and inputs from corrupted parties. We use $\approx$ to denote the two distributions are computationally indistinguishable.

- ***Succinctness**: The proof size and verification time are both of* $\mathsf{poly}(\lambda, |x|, \log |\boldsymbol{w}|)$.

Prior work [OB22] has proved that, for any given zk-SNARK scheme $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$, if there exists an MPC protocol Π that can compute the prover algorithm Prove of the underlying zk-SNARK against up to t corruptions, then there exists a collaborative zk-SNARK $(\mathsf{Setup}, \Pi, \mathsf{Verify})$. We kindly refer interested readers to see more details in [OB22]. Due to this result, in this work, our primary focus is to design such an MPC protocol Π.

Fully distributed proof generation. Here we introduce a crucial property called *fully distributed* proof generation to describe the MPC protocol Π. Recall that, we wish to evenly distribute the workload of the zk-SNARK prover algorithm among the servers, and we formalize this by the following definition.

Definition 3. *Let* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a zk-SNARK for some NP relation $\mathcal{R}$, T_P and S_P be the computation and space complexity of* Prove*, respectively. Let* $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ *be the corresponding collaborative zk-SNARK. We say the proof generation of the collaborative zk-SNARK is **fully distributed**, if all servers in Π have the same computation complexity $O(\frac{T_\mathsf{P}}{N})$ and space complexity $O(\frac{S_\mathsf{P}}{N})$, where N is the number of servers.*

When the protocol Π is designed in the preprocessing model[2]*, we only require all servers in the* online phase *of Π have the same computation complexity $O(\frac{T_\mathsf{P}}{N})$ and space complexity $O(\frac{S_\mathsf{P}}{N})$.*

[2]The protocol in the preprocessing model is divided into two phases: (i) preprocessing phase, where the inputs are unknown and some correlated randomness is generated; (ii) online phase, where the inputs are known and the correlated randomness is consumed to complete the computation task.

Not that, we make a reasonable relaxation when the protocol Π is designed in the preprocessing model. However, even if there is such a relaxation, the previous work [GGJ+23], which designed their protocols in the preprocessing model, cannot make their MPC protocol to be fully distributed.

We will also use the term "fully distributed" to describe some components of the proof generation, such as the sumcheck protocol and the PC scheme, if the workload of the proving algorithms of these components can be evenly distributed across the servers.

Distributing the witness. In the collaborative zk-SNARK framework, the starting point is to let each server receive the secret shares of witness $\boldsymbol{w}$. Prior works [OB22, GGJ+23] has explored various approaches for distributing the witness, typically under two scenarios:

- *Multiple-provers scenario:* in [OB22], the authors consider a scenario where multiple provers, each of them holds a different witness, want to collaboratively generate a proof using jointly computed witness (a.k.a., extended witness). To achieve this, the servers first employ an MPC protocol to compute the extended witness, such as calculating the extended witness as the sum of their witnesses; in the end, each server receives a share of the extended witness.
- *Outsourcing scenario:* in [GGJ+23], the authors consider a scenario where a client who holds the witness wishes to outsource the proof generation to a group of servers. In order to do so, the client first performs some local computations to compute the (extended) witness, and then shares it among the servers.

As noted in [OB22, GGJ+23], distributing the witness is not a central concern in collaborative zk-SNARKs. This is primarily because computing and sharing the witness are less resource-intensive compared to the proof generation process. Hence, in this work, we assume the servers already obtained shares of the extended witness and put our main effort into the proof generation phase.

Threat model. Now, let us delve into the threat model of this work. We adopt an *honest-majority* setting. We set the packing factor of PSS as $k := \frac{N}{4}$, where N is the number of servers, and our protocols are secure against $t < \frac{N}{4}$ corrupted servers, which aligns with [GGJ+23]. Both semi-honest adversaries and malicious adversaries are considered in this work. More precisely, in Section 5 and 6, we discuss the collaborative proof generation against a semi-honest adversary. Our malicious secure protocols are put in Section 7.

zk-SNARKs for arithmetic circuits. Contrasting with zk-SNARKs [Gro16, GWC19, CHM+19], which are tailored for Rank-1 Constraint Systems (R1CS) as considered in prior works [OB22, GGJ+23], our focus is the GKR-based zk-SNARKs which are designed for *arithmetic circuits*. Specifically, we will delve more into *data-parallel* arithmetic circuits, where inputs are split into multiple batches, with identical computations conducted on each batch. This structure is valuable in practical applications, such as (i) machine learning, where a series of similar matrix multiplications are performed on various inputs; (ii) cross-chain bridges, where batches of transactions require verification.

5 Fully Distributed Sumcheck Protocol via Packed Secret Sharing

In our endeavor to enable highly efficient collaborative proof generation, we choose a class of doubly-efficient GKR-based zk-SNARKs, as our foundation. The term "doubly-efficient" means that the computational overhead for both the prover and the verifier are optimized. Given the fact that the GKR protocol fundamentally relies on the sumcheck protocol as its core component, our primary focus is to develop a highly efficient, tailor-made Multi-Party Computation (MPC) protocol for the sumcheck protocol [LFKN90].

5.1 Linear-time sumcheck for multilinear functions

In [Tha13], Thaler proposed a linear prover time algorithm for the sumcheck protocol. Specifically, for a polynomial f summed over an ℓ-variable hypercube, the running time of the prover is $O(2^\ell)$. This running time is considered linear because we can conceptualize f as a multilinear extension of a vector $\boldsymbol{x}$ with $n = 2^\ell$ elements, translating to a prover time complexity of $O(n)$. We first provide a quick review of its mechanism.

The protocol contains ℓ rounds. During the i-th round of the sumcheck protocol, the verifier V needs to check whether $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$ holds, where r_{i-1} is the challenge sent by V in the previous round and $f_i(x_i) = \sum_{b_{i+1},...,b_\ell \in \{0,1\}} f(r_1, ..., r_{i-1}, x_i, b_{i+1}, ..., b_\ell)$. Given that $f_i(x_i)$ is a *multilinear polynomial*, it is sufficient for the prover P to simply claim $f_i(0)$ and $f_i(1)$ in each round.

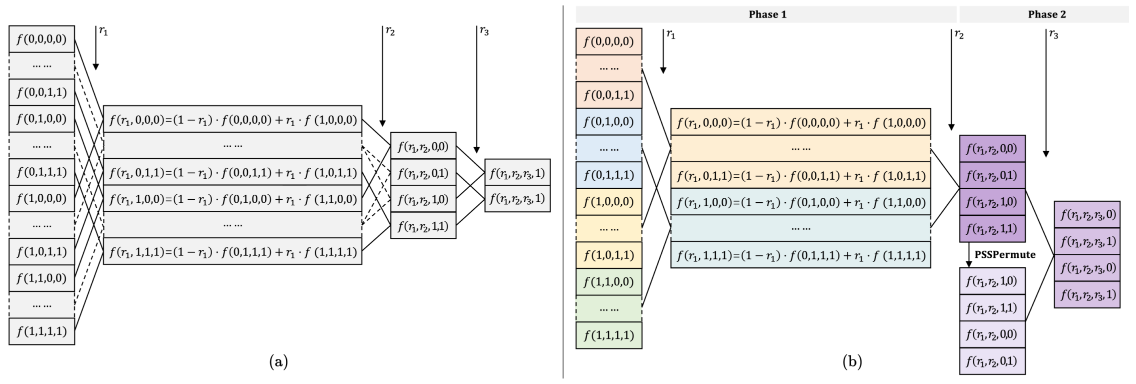


Figure 3: The comparison between the linear-time sumcheck protocol presented in [Tha13] and our proposed scheme, in the case of $\ell = 4$. In (a), a single prover P fills the bookkeeping table round-by-round according to Equation 3. Under our setting, the servers use PSS to accelerate this procedure according to Protocol 4. The computation of each server during the procedure is depicted in (b). In this case we pick $k = 4$, packing $n = 2^\ell = 16$ elements into $\frac{n}{k} = 4$ vectors. Each server initially holds $\frac{n}{k} = 4$ shares, where different colors represent different packed shares in the figure.

To expedite the computation of $f_i(0), f_i(1)$, P maintains a *bookkeeping table*. In the i-th round, there are $2^{\ell-i+1}$ entries stored in this table. In the first round, the table is initialized by 2^ℓ entries according to the evaluations of f on the hypercube. An important observation to note is the existence of a relationship for entries in successive rows of the bookkeeping table, as shown in Equation 3. Note that in this relation, both $f(r_1, ..., r_{i-1}, 0, \boldsymbol{b^{(i)}})$, $f(r_1, ..., r_{i-1}, 1, \boldsymbol{b^{(i)}})$ were computed in the previous round. Then in i-th round ($i \in [\ell]$), utilizing Equation 3, given challenge r_i, P computes $2^{\ell-i}$ entries of the values $f(r_1, ..., r_i, b_{i+1}, ..., b_\ell)$ for all $b_{i+1}, ..., b_\ell \in \{0,1\}^{l-i}$. These computed entries are stored in the bookkeeping table for the next round. To obtain the evaluations of $f_i(0)$ and $f_i(1)$ in each round, P sums up the first and second halves of the entries in the table, respectively. For a more intuitive understanding of this idea, consider the simple example illustrated in part (a) of Figure 3.

Efficiency. It can be concluded that P runs in $\sum_{i=1}^{\ell} 2^{\ell-i} = O(2^\ell) = O(n)$. Maintaining the bookkeeping table, the space complexity is also $O(n)$. The round complexity of this protocol is $\ell = O(\log n)$.

5.2 Fully distributed sumcheck for multilinear functions

To design a fully distributed sumcheck protocol, the key step is to let the servers collaboratively compute the bookkeeping table. We start by letting each server S_i receive only packed (secret) shares of the witness, which is represented by the evaluations of f on the hypercube $\{0,1\}^\ell$.

Naively performing packed secret sharing of all n elements in a batch is not a good choice. Firstly, it does not help with subsequent computations, since each round of computation in the sumcheck protocol requires operations to be performed between corresponding elements at specific positions. Secondly, directly packing of all elements in a batch means the packing factor should be as large as the input size; however, the packing factor cannot be set arbitrarily since it influences the corruption threshold.

Local computation via PSS. Our idea is to use the PSS technique to facilitate the computations. More precisely, we can divide the $n_i = 2^{\ell-i+1}$ entries in round i, denoted as $x_1^{(i)}, \ldots, x_{n_i}^{(i)}$ (we use superscript to denote the round number), into multiple vectors $\{\boldsymbol{x}_j^{(i)}\}_{j\in[\frac{n_i}{k}]}$. Each vector is represented as packed shares. With shares at hand, Equation 3 can be performed in a SIMD fashion across corresponding share-pairs $\boldsymbol{x}_j^{(i)}$ and $\boldsymbol{x}_{j+\frac{n_i}{2k}}^{(i)}$ ($j \in [\frac{n_i}{2k}]$) locally by each server. The results $\{\boldsymbol{x}_j^{(i+1)}\}_{j\in[\frac{n_{i+1}}{k}]}$ are still in the form of packed shares, allowing the local computation to be repeated round by round. It is easy to see that, these local computations will be stuck in the $(\log_2 \frac{n}{k} + 1)$-th round, since at that round, each server possesses only one share, unfeasible for further computations. For better understanding, we list the first $\log_2 \frac{n}{k}$-round computation among the servers in the following:

- During the initialization phase, for $n_1 = n = 2^\ell$ inputs, each server receives $\frac{n_1}{k}$ packed shares $\{[\![\boldsymbol{x_j^{(1)}}]\!]\}_{j\in[\frac{n_1}{k}]}$, where $\boldsymbol{x_j^{(1)}}$ is a k-sized vector that $\boldsymbol{x_j^{(1)}} = \{x_{k(j-1)+1}^{(1)}, ..., x_{kj}^{(1)}\}$.

- Subsequently, in i-th round ($i \in [\log_2 \frac{n}{k}]$), the servers first compute claims needed in this round. In each round, two shares are summed as follows: $[\![\boldsymbol{a_1^{(i)}}]\!] = \sum_{j=1}^{\frac{n_i}{2k}} [\![\boldsymbol{x_j^{(i)}}]\!]$, $[\![\boldsymbol{a_2^{(i)}}]\!] = \sum_{j=\frac{n_i}{2k}+1}^{\frac{n_i}{k}} [\![\boldsymbol{x_j^{(i)}}]\!]$, where $[\![\boldsymbol{a_1^{(i)}}]\!]$ and $[\![\boldsymbol{a_2^{(i)}}]\!]$ are two shares, each packing k elements. After this, each server sends his two shares to the verifier, who, upon collecting sufficient shares, reconstructs $\boldsymbol{a_1^{(i)}}$ and $\boldsymbol{a_2^{(i)}}$ and sums the elements within each vector to obtain $f_i(0)$ and $f_i(1)$ for completing the verification process. After checking the validity of these two claims, the verifier returns a random challenge r_i to the servers, and each server can locally compute shares of entries needed in the next round. More precisely, for $j \in [\frac{n_{i+1}}{k}]$,

$$[\![\boldsymbol{x_j^{(i+1)}}]\!] = (1 - r_i) \cdot [\![\boldsymbol{x_j^{(i)}}]\!] + r_i \cdot [\![\boldsymbol{x_{j+\frac{n_i}{2k}}^{(i)}}]\!] \ . \tag{4}$$

See Phase 1 in part (b) of Figure 3 for an illustration of this process.

Further computation via PSS permutation. For better expression, without loss of generality, we assume that $k = 2^s$ and $n = 2^\ell$. After the $(\ell - s)$-th round, each server is left with only one element $[\![\boldsymbol{x^{(\ell-s+1)}}]\!]$. To address the above challenge, our approach, particularly in each i-th round ($i \in [\ell - s + 1, \ell)$), involves having the servers initially acquire an additional element $[\![\boldsymbol{\hat{x}^{(i)}}]\!]$. This element represents a packed secret sharing of the vector $\boldsymbol{\hat{x}^{(i)}}$, which in turn is a *permutation* of the original entries.

To ensure that the linear combination in Equation 4 can be effectively carried out between the two shares, careful design of this permutation is necessary. Generally, in i-th round ($i \in [\ell - s + 1, \ell)$), let $p_i(\cdot)$ be a permutation function, which is defined as follows:

$$p_i(j) = \begin{cases} j + \frac{n_i}{2}, & j \in (0, \frac{n_i}{2}] \\ j - \frac{n_i}{2}, & j \in (\frac{n_i}{2}, n_i] \\ j, & j \in (n_i, k] \end{cases} \tag{5}$$

For notion convenience, we use $\boldsymbol{\hat{x}} = p_i(\boldsymbol{x})$ to denote the event that we perform the permutation $p_i(\cdot)$ over the vector $\boldsymbol{x}$ and obtain $\boldsymbol{\hat{x}}$ as a result, where $\boldsymbol{x} = \{x_1, ..., x_{n_i}\}$, $\boldsymbol{\hat{x}} = \{x_{p_i(1)}, ..., x_{p_i(n_i)}\}$. After the servers obtaining $[\![\boldsymbol{x^{(i)}}]\!]$ and $[\![\boldsymbol{\hat{x}^{(i)}}]\!]$, the servers can perform the linear combination locally:

$$[\![\boldsymbol{x^{(i+1)}}]\!] = (1 - r_i) \cdot [\![\boldsymbol{x^{(i)}}]\!] + r_i \cdot [\![\boldsymbol{\hat{x}^{(i)}}]\!] \tag{6}$$

to obtain the share of entries needed in the next round. For an illustration of this process, refer to Phase 2 in part (b) of Figure 3.

Now the only thing left is how to let the servers obtain the permuted shares. To address this, we adopt a technique called *PSS permutation*, which is also used in [DIK10, GPS21a]. More precisely, PSS permutation allows the servers to take the packed share of the original vector as input and output the packed share of the permuted vector. We provide the functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ and the protocol $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$ in Appendix B.4. We stress that the protocol $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$ does not compromise the "fully distributed" nature of our protocol. This is because the king server S_1 in the protocol does not have high computational complexity or substantial space complexity. Furthermore, in each round, S_1 can be selected as any one of the servers, following a round-robin approach.

Remark 1. *Let Π_{dFFT} be the distributed FFT protocol proposed in [GGJ+23]. As discussed in [GGJ+23], the authors also tried to apply the PSS permutation technique to Π_{dFFT}. However, it turns out that it would lead to a new problem: in each of the final $\log k$ rounds of Π_{dFFT}, an interactive PSS multiplication protocol must be executed after each permutation, leading to substantial computational overhead and making their protocol become impractical. In contrast, our approach requires the servers to additionally execute a PSS permutation protocol and some local linear combinations, which only increase some slight cost. This difference stems from the differing natures of the FFT and sumcheck protocols.*

Put everything together. Based on the discussion above, it can be concluded that the proving work of the sumcheck protocol can be executed collaboratively by multiple servers in a fully distributed manner. Notice that, the sumcheck protocol can be transformed into non-interactive using Fiat-Shamir transform [FS87] under the random oracle model. In our setting, this involves letting the servers reconstruct the transcript and use it to query the random oracle, so that each server can receive the same random element. Putting everything together, we present our protocol $\Pi_{\mathsf{dSumcheck}}$ in Figure 4.

Formally, we denote by ($\mathsf{NISumcheck.Prove}$, $\mathsf{NISumcheck.Verify}$) the non-interactive version of the original sumcheck protocol for a given ℓ-variate multilinear polynomial $f(\boldsymbol{x})$. The security of our protocol $\Pi_{\mathsf{dSumcheck}}$ is proven through Theorem 1.

Protocol $\Pi_{\mathsf{dSumcheck}}$

We assume $k = 2^s \geq 2$ for some positive integer s and $n_i = 2^{\ell-i+1}$ for $i \in [\ell]$. Let $f : \mathbb{F}^\ell \to \mathbb{F}$ be a multilinear polynomial, and $\boldsymbol{x} \in \mathbb{F}^n$ is the evaluations of f on the hypercube $\{0,1\}^\ell$. Suppose there are N servers $\mathsf{S}_1, ..., \mathsf{S}_N$. Each server holds packed shares of vectors $\boldsymbol{x_j} = \boldsymbol{x_j^{(1)}} = \{x_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [\frac{n_1}{k}]$. The following procedure runs under the random oracle model, where we use H to denote a random oracle. When we say each server makes a random query to H, it means that the servers reconstruct the current transcript $\mathcal{T}$ and use it to query the random oracle to receive the same random challenge $r = \mathsf{H}(\mathcal{T})$.

1. **Phase 1.** In the i-th round, where $1 \leq i \leq \ell - s$,
   (a) Each server locally computes $[\![\boldsymbol{a_1^{(i)}}]\!] = \sum_{j=1}^{\frac{n_i}{2k}} [\![\boldsymbol{x_j^{(i)}}]\!]$, $[\![\boldsymbol{a_2^{(i)}}]\!] = \sum_{j=\frac{n_i}{2k}+1}^{\frac{n_i}{k}} [\![\boldsymbol{x_j^{(i)}}]\!]$.
   (b) The servers make a random query to H and receive a random $r_i \in \mathbb{F}$, then locally compute $\{[\![\boldsymbol{x_j^{(i+1)}}]\!]\}_{j\in[\frac{n_{i+1}}{k}]}$ by Equation 4.
2. **Phase 2.** In the i-th round, where $\ell - s + 1 \leq i < \ell$,
   (a) The servers take $[\![\boldsymbol{x^{(i)}}]\!]$ as input and invoke $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ to obtain $[\![\hat{\boldsymbol{x}}^{(i)}]\!]$, using the permutation in Equation 5.
   (b) The servers make a random query to H and receive a random $r_i \in \mathbb{F}$, then locally compute $[\![\boldsymbol{x^{(i+1)}}]\!]$ by Equation 6.
3. The servers output $\{[\![\boldsymbol{a_1^{(i)}}]\!], [\![\boldsymbol{a_2^{(i)}}]\!]\}_{i\in[1,\ell-s]}$ and $\{[\![\boldsymbol{x^{(i)}}]\!]\}_{i\in[\ell-s+1,\ell]}$.

Figure 4: The fully distributed sumcheck protocol $\Pi_{\mathsf{dSumcheck}}$ in the $\{\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}, \mathsf{H}\}$-hybrid world.

Theorem 1. *The protocol $\Pi_{\mathsf{dSumcheck}}$ depicted in Figure 4 is a secure MPC protocol that computes* NISumcheck.Prove *in the $\{\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}, \mathsf{H}\}$-hybrid world against a semi-honest adversary who corrupts at most t servers. This protocol $\Pi_{\mathsf{dSumcheck}}$ is fully distributed.*

Proof. In i-th round ($i \in [1, l-s]$), the sums of elements inside $\boldsymbol{a_1^{(i)}}, \boldsymbol{a_2^{(i)}}$ are actually $f_i(0), f_i(1)$, respectively. In i-th round, ($i \in [l-s+1, l]$), $f_i(0) = \sum_1^{\frac{n_i}{2}} x_j^{(i)}$ and $f_i(1) = \sum_{\frac{n_i}{2}+1}^{n_i} x_j^{(i)}$ holds. Therefore, the correctness of the protocol is straightforward.

Security. As the protocol $\Pi_{\mathsf{dSumcheck}}$ only makes oracle access to the ideal functionality of $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ and H, besides performing local operations on the shares of the inputs, the security of this protocol follows the security of the functionality invoked.

Efficiency. Here we only discuss the online phase efficiency, and let $n = 2^\ell$. In Phase 1, each server individually performs $O(\frac{n}{k}) = O(\frac{n}{N})$ field operations. In Phase 2, each server individually does $O(\log k) = O(\log N)$ field operations. The computation complexity of $\log k$ calls to $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ is negligible to the overall cost, as $N \ll n$. Therefore, the total proving work of N servers is $O(n)$, and the computational overhead for each server S_i is $O(\frac{n}{N})$. The space complexity of each server is consistently $O(\frac{n}{N})$. Since the computation complexity and the space complexity of each server are equally $O(\frac{n}{N})$, the protocol is fully distributed. □

Remark 2. *So far we obtain a fully distributed sumcheck protocol where N servers collaborate together with k times improvement of proving time. The total proving work is $O(n)$ while each server's overhead is equally of $O(\frac{n}{N})$. In total, this approach is as efficient as the original protocol described in Section 5.1. When compared with the distributed sumcheck protocol proposed in [XZC+22], our method achieves similar efficiency in terms of asymptotic complexity, with respect to the average overhead on each server. Importantly, our scheme ensure privacy that no single server can gain any information about the inputs; while the protocol proposed in [XZC+22] does not.*

Extending to the product of two multilinear polynomials. The linear-time sumcheck described in Section 5.1 can be extended for a product of two multilinear polynomials f, g:

$$\sum_{b_1,b_2,...,b_\ell \in \{0,1\}} f(b_1, b_2, ..., b_\ell) \cdot g(b_1, b_2, ..., b_\ell) \ . \tag{7}$$

The approach involves initially computing the bookkeeping tables for f, g separately, each within $O(n)$ time. Then, in each round, the computation is:

$$\sum_{b_{i+1},...,b_\ell \in \{0,1\}} f(r_1,...,r_{i-1},x_i,b_{i+1},...,b_\ell) \cdot g(r_1,...,r_{i-1},x_i,b_{i+1},...,b_\ell)$$

This is achieved by multiplying the corresponding entries in the bookkeeping tables of f, g, and performing a summation. The overall time complexity remains $O(n)$.

We note that the distributed sumcheck described in Figure 4 can be generalized to a product of two multilinear polynomials as well. In a manner akin to the extension above, the servers first use $\Pi_{\mathsf{dSumcheck}}$ to collaboratively compute the bookkeeping tables for each polynomial. Subsequently, the entry-wise multiplications of these two bookkeeping tables are replaced by the servers utilizing a procedure called PSS multiplication. PSS multiplication means that, given two packed shares $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$, the goal is to compute $[\![\boldsymbol{c}]\!]$ such that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. We model it as an ideal functionality $\mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}$ and outline it in Appendix B.3. The total computational complexity for each server remains $O(\frac{n}{N})$.

6 Fully Distributed Proof Generation

In this section, we first describe how to distribute the mKZG PC scheme described in Section 2.2. Following this, we demonstrate the method to achieve fully distributed proof generation for Libra.

6.1 Fully distributed KZG for multilinear polynomials

In the last step of the original GKR protocol [GKR08], the prover P claims two value $\tilde{V}_d(\boldsymbol{u}^{(d)})$ and $\tilde{V}_d(\boldsymbol{v}^{(d)})$ to the verifier V, where $\boldsymbol{u}^{(d)}, \boldsymbol{v}^{(d)} \in \mathbb{F}^m$ are two random vectors selected by V and $\tilde{V}_d$ is the multilinear extension of V_d, which is the witness on the input layer of the circuit. To eliminate the necessity for V to access the entire witness, [ZGK+17a] proposed to firstly commit to the witness on the input layer and then verify the claims using a polynomial commitment (PC) scheme. Since then, many GKR-based zk-SNARKs [ZGK+18, XZZ+19, ZXZS20] adopt a PC scheme as their main building block. In this work, we also require a PC scheme and we will show how to distribute it evenly.

Generalizing mKZG for multiple servers. In [GGJ+23], Garg et al. introduced a distributed KZG scheme for univariate polynomials, utilizing packed secret sharing. However, adapting their approach to our context is a non-trivial task. In their configuration, each server holds a share of the polynomial's coefficients. In contrast, in our situation, each server holds a share of the witness, which actually represents the evaluations of the polynomial on a hypercube. Moreover, their scheme incorporates the use of sub-protocols like distributed Multi-Scalar Multiplication (dMSM) and distributed Fast-Fourier Transform (dFFT). The latter, is only "partially" distributed since it relies heavily on a single powerful, larger server for most computations. This reliance results in increased communication costs and a potential memory bottleneck, which is contrary to our goal of establishing a fully distributed collaborative zk-SNARK where computation and space complexity are equally shared among all servers. Additionally, a recent work by Liu et al. [LXZ+23] introduced a distributed bivariate KZG protocol. However, this method is also not ideally suited to our setting, since it allows the servers to obtain the entire witness. In contrast, in our setting, we let the servers obtain only packed shares of the witness.

Recall that, in GKR-based zk-SNARKs, the PC scheme is used to commit the multilinear extension of the witness in the input layer. In our setting, each server only holds packed secret shares of the witness. For ease of presentation, let $\boldsymbol{x}$ denote a vector of n elements, corresponding to witness on the input layer. $\boldsymbol{x}$ can be effectively represented by a function $V : \{0,1\}^\ell \rightarrow \mathbb{F}$, where $\ell = \log_2 n$. Let f be the multilinear extension of V. With PSS, initially, each server holds $\frac{n}{k}$ packed shares, represented as $\{[\![\boldsymbol{x_j}]\!]\}_{j \in [\frac{n}{k}]}$. Here, each $\boldsymbol{x_j}$ is a vector of size k, where $\boldsymbol{x_j} = \{x_{k(j-1)+1}, ..., x_{kj}\}$ and k is the packing factor we choose. The primary challenge is two-fold: (i) generate commitment for f, and (ii) generate proof for evaluations.

Generating the commitment. The main problem for generating commitment for f in a distributed way is that each server possesses only the packed shares of f's evaluations on the hypercube, rather than its direct coefficients. To tackle this, we propose to leverage Equation 1. More precisely, we observe that the commitment can be computed as $\mathsf{com}_f = g^{f(\boldsymbol{s})} = \prod_{\boldsymbol{b} \in \{0,1\}^\ell} g^{\prod_{i=1}^{\ell} \beta_{b_i}(s_i) \cdot V(\boldsymbol{b})}$, where $\beta_{b_i}(s_i) = (1-s_i)(1-b_i) + s_i b_i$, each $V(\boldsymbol{b}) \in \mathbb{F}$ is a scalar in $\boldsymbol{x}$ and $g^{\prod_{i=1}^{\ell} \beta_{b_i}(s_i)} \in \mathbb{G}$ is a group element. This commitment can then be computed running an MSM on inputs $x_1, ..., x_n \in \mathbb{F}$ and the corresponding group elements in $\mathbb{G}$.

To facilitate this process, we initially prepare packed shares of $g^{\prod_{i=1}^{\ell}\beta_{b_i}(s_i)}$ for all possible $\boldsymbol{b} \in \{0,1\}^{\ell}$, for the servers. With these public parameters and the packed shares of $\boldsymbol{x}$, the servers can employ the technique of distributed MSM (dMSM), as introduced in [GGJ+23], to collaboratively compute com_f. In this sub-protocol dMSM, each server only equally bears an overhead of performing $O(\frac{n}{k})$ group exponentiations and a space cost of $O(\frac{n}{k})$. For clarity and completeness, we borrow the ideal functionality $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$ and protocol $\Pi_{\mathsf{dMSM\text{-}Semi}}$ from [GGJ+23] and put them in Appendix B.5 with minor modifications.

Protocol Π_{dMKZG}

Let $\mathbb{F}$ denote a finite field and assume $k = 2^s \geq 2$ for some positive integer s and $n_i = 2^{\ell-i}$ for $i \in [0,\ell]$. We assume $n = n_0 = 2^{\ell}$. Suppose $f : \mathbb{F}^{\ell} \to \mathbb{F}$ is a multilinear polynomial, and $\boldsymbol{x} \in \mathbb{F}^n$ is the evaluations of f on the hypercube $\{0,1\}^{\ell}$. Suppose there are N servers $\mathsf{S}_1, ..., \mathsf{S}_N$. Each server holds packed secret shares of vectors $\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{0})} = \{x_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [\frac{n_0}{k}]$.

Procedure $\mathsf{dMKZG.Setup}$**:** This procedure is executed by a trusted setup, aiming to provide public parameters pp for servers.

1. Sample $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}^{\ell}$ as the trapdoor.
2. Prepare packed shares of group elements needed for $\mathsf{dMKZG.Commit}$ and each round in $\mathsf{dMKZG.Open}$. For $i \in [0,\ell]$, do the following:
   (a) Compute the evaluations of $p_i(b_{i+1},\ldots,b_{\ell}) = g^{\prod_{j=i+1}^{\ell}\beta_{b_j}(s_j)}$ on the hypercube $\{0,1\}^{\ell-i}$, where $\beta_{b_j}(s_j) = (1-s_j)(1-b_j) + s_j b_j$. In i-th round, there will be n_i group elements.
   (b) Pack the n_i group elements from the Step 2a into k-size vectors $\boldsymbol{P}_{\boldsymbol{1}}^{(\boldsymbol{i})}, ..., \boldsymbol{P}_{\frac{\boldsymbol{n_i}}{\boldsymbol{k}}}^{(\boldsymbol{i})}$. If the elements cannot fully populate one vector (i.e., $n_i < k$), it will be supplemented with zero elements of the group until there are k elements.
   (c) Each server receives $\{[\![\boldsymbol{P}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!]\}_{j\in[\frac{n_i}{k}]}$ as pp_i.

Procedure $\mathsf{dMKZG.Commit}$**:** This procedure is executed by the servers to collaboratively compute the commitment of a given multilinear polynomial $f : \mathbb{F}^{\ell} \to \mathbb{F}$. Note that, the servers do not know the entire f, but they hold the PSS of $\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{0})} = \{x_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [\frac{n_0}{k}]$, where $\boldsymbol{x} \in \mathbb{F}^{n_0}$ is the evaluations of f on the hypercube $\{0,1\}^{\ell}$.

1. Each server parse pp_0 as $\{[\![\boldsymbol{P}_{\boldsymbol{j}}^{(\boldsymbol{0})}]\!]\}_{j\in[\frac{n_0}{k}]}$.
2. The servers send $(\textsc{Mult}, \frac{n_0}{k}, \{[\![\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{0})}]\!]\}_{j\in[\frac{n_0}{k}]}, \{[\![\boldsymbol{P}_{\boldsymbol{j}}^{(\boldsymbol{0})}]\!]\}_{j\in[\frac{n_0}{k}]})$ to $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, which returns $\langle \mathsf{com}_f \rangle$ to the servers.

Procedure $\mathsf{dMKZG.Open}$**:** Given the evaluation point $\boldsymbol{u}$ and the evaluation y, this procedure is executed by the servers to collaboratively compute proof π that proves $y = f(\boldsymbol{u})$ indeed holds. Note that, the servers takes the PSS of $\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{0})} = \{x_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [\frac{n_0}{k}]$ as input, as the above procedure. The servers also hold the evaluation point $\boldsymbol{u}$. The procedure proceeds in ℓ rounds.

1. In the i-th round, where $1 \leq i \leq \ell - s$,
   (a) Each server locally computes $[\![\boldsymbol{q}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!] = [\![\boldsymbol{x}_{\boldsymbol{j}+\frac{\boldsymbol{n_i}}{\boldsymbol{k}}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!] - [\![\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!]$ and $[\![\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!] = (1-u_i)\cdot[\![\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!] + u_i \cdot [\![\boldsymbol{x}_{\boldsymbol{j}+\frac{\boldsymbol{n_i}}{\boldsymbol{k}}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!]$, for $j \in [\frac{n_i}{k}]$.
   (b) Each server parse pp_i as $\{[\![\boldsymbol{P}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!]\}_{j\in[\frac{n_i}{k}]}$.
   (c) The servers send $(\textsc{Mult}, \frac{n_i}{k}, \{[\![\boldsymbol{q}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!]\}_{j\in[\frac{n_i}{k}]}, \{[\![\boldsymbol{P}_{\boldsymbol{j}}^{(\boldsymbol{i})}]\!]\}_{j\in[\frac{n_i}{k}]})$ to $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, which returns $\langle \pi_i \rangle$ to the servers.
2. In the i-th round, where $\ell - s < i \leq \ell$,
   (a) The servers invoke $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ to obtain $[\![\hat{\boldsymbol{x}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!]$, following the permutation described in Equation 5.
   (b) Each server locally computes $[\![\boldsymbol{q}^{(\boldsymbol{i})}]\!] = [\![\hat{\boldsymbol{x}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!] - [\![\boldsymbol{x}^{(\boldsymbol{i}-\boldsymbol{1})}]\!]$ and $[\![\boldsymbol{x}^{(\boldsymbol{i})}]\!] = (1-u_i)\cdot[\![\boldsymbol{x}^{(\boldsymbol{i}-\boldsymbol{1})}]\!] + u_i \cdot [\![\hat{\boldsymbol{x}}^{(\boldsymbol{i}-\boldsymbol{1})}]\!]$.
   (c) Each server parse pp_i as $\{[\![\boldsymbol{P}^{(\boldsymbol{i})}]\!]\}$.
   (d) The servers send $(\textsc{Mult}, 1, [\![\boldsymbol{q}^{(\boldsymbol{i})}]\!], [\![\boldsymbol{P}^{(\boldsymbol{i})}]\!])$ to $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, which returns $\langle \pi_i \rangle$ to the servers.
3. The servers output $\langle \pi \rangle = (\langle \pi_1 \rangle, ..., \langle \pi_{\ell} \rangle)$.

Figure 5: The fully distributed PC for multilinear polynomials in the $\{\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}, \mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}\}$-hybrid world.

Generating the opening proof. During the process of mKZG.Open, in order to generate a proof for the claim $z = f(\boldsymbol{u})$, where f is evaluated at a random point $\boldsymbol{u}$, the servers take the following two steps:

- Firstly, the servers undertake a sequence of polynomial divisions. This procedure involves consecutively running ℓ times divisions for multilinear polynomials, resulting in a set of quotient polynomials $\{Q_i(x_{i+1}, ..., x_\ell)\}_{i\in[\ell]}$ and remainder polynomials $\{R_i(x_{i+1}, ..., x_\ell)\}_{i\in[\ell]}$. Let R_0 denote the original polynomial f, then in i-th division, the operation is performed on the remainder polynomials of the last round R_{i-1} with respect to the divisor $(x_i - u_i)$, executed in a recursive manner. Formally, for $i \in [\ell]$,

$$R_{i-1}(x_i, x_{i+1}, ..., x_\ell) = Q_i(x_{i+1}, ..., x_\ell)(x_i - u_i) + R_i(x_{i+1}, ..., x_\ell) \tag{8}$$

- Upon obtaining these polynomials, the servers then compute the proof, represented as $\{g^{Q_i(\boldsymbol{s})}\}_{i\in[\ell]}$. Obviously, if the evaluations of Q_i on the hypercubes are on hand, then this computation can be carried out in a manner akin to the commitment generation process we described earlier.

In [GGJ+23], conducting polynomial divisions involves using a sub-protocol called dFFT. A significant drawback of employing this protocol is its reliance on a powerful server. Additionally, the necessity for servers to execute dFFT multiple times also results in substantial computational and communication overheads.

To tackle the issue, we propose a new approach that leverages the algebraic property of multilinear polynomials. Note that, in Equation 8, both the quotient polynomials Q_i and remainder polynomials R_i are also multilinear. Let $(x_{i+1}, ..., x_\ell)$ takes $\boldsymbol{b} \in \{0, 1\}^{\ell-i}$, and x_i takes 0 and 1 separately, we can derive that $R_{i-1}(0, \boldsymbol{b}) = -u_i \cdot Q_i(\boldsymbol{b}) + R_i(\boldsymbol{b})$ and $R_{i-1}(1, \boldsymbol{b}) = (1 - u_i) \cdot Q_i(\boldsymbol{b}) + R_i(\boldsymbol{b})$. Consequently, the evaluations of R_i and Q_i on $\boldsymbol{b} \in \{0, 1\}^{\ell-i}$ can be derived as per Equation 9:

$$\begin{aligned} Q_i(\boldsymbol{b}) &= R_{i-1}(1, \boldsymbol{b}) - R_{i-1}(0, \boldsymbol{b}) \ , \\ R_i(\boldsymbol{b}) &= (1 - u_i) \cdot R_{i-1}(0, \boldsymbol{b}) + u_i \cdot R_{i-1}(1, \boldsymbol{b}) \end{aligned} \tag{9}$$

Similar to Equation 3, the above formula also has a SIMD structure to leverage. Initially, each server holds $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$. Upon receiving a specific point $\boldsymbol{u} \in \mathbb{F}^\ell$, each server begins to compute the shares of evaluations on the hypercube $\{0, 1\}^{\ell-1}$ for the first quotient polynomial Q_1. This step involves the subtraction of the first half of the shares $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$ from the second half. The shares of evaluations for the first remainder polynomial R_1 are determined through linear combinations of the packed shares within $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$, analogous to Equation 4. This computation is then recursively applied for each of the quotient polynomials, up to Q_ℓ, in a manner akin to the process of computing the bookkeeping table in Section 5.

In the second step, after acquiring packed shares of the evaluations of Q_i on the hypercube $\{0, 1\}^{\ell-i}$, the servers collaborate to compute $g^{Q_i(s_{i+1},...,s_\ell)}$. This computation is conducted by servers invoking the $\Pi_{\mathsf{dMSM\text{-}Semi}}$ protocol, similar to the previously described procedure of computing $g^{f(\boldsymbol{s})}$.

Put everything together. Combining the above discussions, we develop a fully distributed PC protocol Π_{dMKZG}, specifically designed for multilinear polynomials. Notice that, to facilitate this protocol, we make some minor adjustments to mKZG.Setup and design dMKZG.Setup as the setup procedure. Formally, the Π_{dMKZG} protocol is detailed in Figure 5. The security of this protocol is proven in Theorem 2.

Theorem 2. *The protocol* Π_{dMKZG} *depicted in Figure 5 is an MPC protocol whose procedures* dMKZG.Commit *and* dMKZG.Open *can compute* mKZG.Commit *and* mKZG.Open *respectively, in the* $\{\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}, \mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}\}$*-hybrid world against a semi-honest adversary who corrupts at most t servers. The protocol* Π_{dMKZG} *is fully distributed.*

Proof. The correctness is straightforward.

Security. As the protocol Π_{dMKZG} only makes oracle access to $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ and $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, besides performing local operations to the shares of the inputs, the proof is straightforward by applying the security of $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ and $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$.

Efficiency. Here we only discuss the online phase efficiency of the protocol, and let $n = 2^\ell$. During dMKZG.Commit, the servers engage in $\Pi_{\mathsf{dMSM\text{-}Semi}}$ to collaboratively compute the commitment. Each server performs $O(\frac{n}{N})$ group exponentiations, has a space complexity of $O(\frac{n}{N})$, and communicates $O(N)$ group elements in total. In dMKZG.Open, the total proving work of servers is $O(n)$ and each server S_i incurs a computational overhead of $O(\frac{n}{N})$. The space complexity for each server remains $O(\frac{n}{N})$. The overall communication cost among servers is $O(N \log n)$ group elements. Since the computation complexity and the space complexity of each server are equally $O(\frac{n}{N})$, the protocol is fully distributed. □

6.2 Fully distributed, collaborative proof generation

Combining the two fully distributed primitives above, we can achieve a collaborative zk-SNARK for data-parallel circuits. A data-parallel circuit is a circuit composed of several identical sub-circuits, each operating independently without interconnections.

The critical step involves organizing the values at the corresponding positions across different sub-copy of the circuit into the same vectors. These vectors are then distributed among the servers using PSS. Utilizing the SIMD structure of the data-parallel circuit, after this distribution, the distributed sumcheck protocol can be extended for GKR relations, as detailed in Appendix C.1. The final step involves employing the distributed PC scheme to generate proofs for claims at the input layer. By synthesizing these components, the collaborative proof generation is achieved by replacing the sumcheck in Libra with $\Pi_{\mathsf{dSumcheck}}$, and the PC scheme with Π_{dMKZG}. Notice that, we also require a procedure called PSS multiplication, which is modeled as a functionality $\mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}$ (cf. Appendix B.3). This procedure is required when our fully distributed sumcheck protocol is extended for a product of two multilinear polynomials. Formally, we have the following theorem.

Theorem 3. *Let $\mathcal{C}$ be a data-parallel circuit. Let $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ be a zk-SNARK described in [XZZ+19] for $\mathcal{C}$. There exists a collaborative zk-SNARK $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ for $\mathcal{C}$, where Π is a secure MPC protocol that computes Prove in the $\{\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}, \mathcal{F}_{\mathsf{dMSM\text{-}Semi}}, \mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}, \mathsf{H}\}$-hybrid world, against a semi-honest adversary who corrupts at most t servers. Here H denotes a random oracle. This collaborative zk-SNARK is fully distributed.*

Proof sketch. We give a detailed construction of Π in Appendix C.2. The correctness follows the construction directly.

Security. The protocol Π comprises sub-protocols $\Pi_{\mathsf{dSumcheck}}$ and Π_{dMKZG}. Computing these sub-protocols only makes oracle access to the random oracle H, the ideal functionalities $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$, $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, and $\mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}$, besides each server performing local operations on the shares of inputs. The security of this protocol follows the security of the functionalities invoked.

Efficiency. Given a data-parallel arithmetic circuit $\mathcal{C}$, the total proving work of N servers is $O(|\mathcal{C}|)$, and the computational overhead for each server S_i is $O(\frac{|\mathcal{C}|}{N})$. The space complexity of each server is consistently $O(\frac{|\mathcal{C}|}{N})$. Thus, the protocol is fully distributed. $\square$

Extending to general circuits. When $\mathcal{C}$ signifies a general circuit with an arbitrary structure, as opposed to a data-parallel circuit consisting of identical sub-copies, the complexity escalates significantly. While the distributed PC can be employed as before, the distributed sumcheck protocol for GKR relations encounters obstacles due to the circuit's irregularity. Under such circumstances, the most effective currently known approach is sub-optimal: both the computation and space complexities for each server are $O(|\mathcal{C}|)$. Addressing this inefficiency and achieving a significant efficiency improvement by a factor of $O(N)$ for circuits with arbitrary form remains a challenging and open question for future research.

7 Achieving Malicious Security

Recall that, when we consider semi-honest security, our collaborative zk-SNARK is built in the $\{\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}, \mathcal{F}_{\mathsf{dMSM\text{-}Semi}}, \mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}, \mathsf{H}\}$-hybrid world, where H is a random oracle. In order to achieve malicious security, in this section, we first try to design new protocols to make our two main building blocks (i) PSS permutation and (ii) dMSM to be malicious-secure. Then we talk about how to handle PSS multiplication against a malicious adversary. Finally, we put things together and show how to achieve malicious-secure collaborative zk-SNARK.

7.1 Malicious-secure packed secret sharing permutation

In Section 5.2, we have described a protocol for PSS permutation (i.e., $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$ depicted in Protocol 31) against a semi-honest adversary. It is easy to see that when malicious security is taken into consideration, there is no guarantee that the honest parties can receive the correct packed secret shares.

To achieve the malicious security of the whole protocol, we employ the standard approaches that are used in the MPC area (e.g., [GPS21a]): we first let the servers invoke the semi-honest protocol, where the malicious servers may behave dishonestly; at the end of the whole protocol, we let the servers invoke a verification protocol that used to detect the previous malicious behaviors.

In the following, we will first present a malicious-secure version of functionality for PSS permutation, which is denoted by $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$. In $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$, the adversary is allowed to add errors into the honest parties' received shares. Then we will provide a new protocol $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$ that securely-realizes $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$. Next, we will present a functionality for verifying the PSS permutation, which is denoted by $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$, and then we will present a protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$ that securely realizes $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$.

PSS permutation against a malicious adversary. We put the malicious-secure version of functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$ for PSS permutation in Figure 6, which is adapted from [GPS21a]. Then we make some minor modifications to our semi-honest protocol $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$ and turn it into a malicious protocol $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$, which is depicted in Figure 7.

Notice that, in the preprocessing phase of $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$, we let the servers invoke a functionality $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$ to generate a pair of random shares $[\![\boldsymbol{r}]\!]_d$ and $[\![\hat{\boldsymbol{r}}]\!]_d$, where $\hat{\boldsymbol{r}} = p(\boldsymbol{r})$ if all the servers behave honestly; we put the detailed description of $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$ in Appendix B.2. We also note that, in the preprocessing phase, we let the servers invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate a degree-$2d$ PSS $[\![\mathbf{0}]\!]_{2d}$ of 0; the full descriptions about this procedure can be found in [GPS21b, Appendix B.3]. The security of the protocol $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$ is proven through Theorem 4.

Functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$

The functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of corrupted servers. Let $\mathcal{H}$ be the set of honest servers.

Upon receiving $(\textsc{Permute}, [\![\boldsymbol{x}]\!], p(\cdot))$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where $[\![\boldsymbol{x}]\!]$ is a degree-d packed secret sharing of $\boldsymbol{x}$ and $p(\cdot)$ is a permutation, it does:

1. Reconstruct the correct secret $\boldsymbol{x}$ using the shares from the honest servers and compute the permuted secret $\hat{\boldsymbol{x}} := p(\boldsymbol{x})$.
2. Compute a new PSS $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$ of the secret $\boldsymbol{x}$, such that for each honest server, its share of $[\![\boldsymbol{x}]\!]$ is equal to its share of $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$. Send the corrupted servers' shares of $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{x}} := [\![\boldsymbol{x}]\!] - [\![\boldsymbol{x}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{x}}$ describes the inconsistency of $[\![\boldsymbol{x}]\!]$.
3. Receive a vector $\boldsymbol{d} \in \mathbb{F}^k$ from the adversary $\mathcal{S}$ and set $\hat{\boldsymbol{x}} := \hat{\boldsymbol{x}} + \boldsymbol{d}$.
4. Receive a set of shares $\{s_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$. Sample a random degree-d packed secret sharing $[\![\hat{\boldsymbol{x}}]\!]_{\mathcal{H}}$ of $\hat{\boldsymbol{x}}$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $[\![\hat{\boldsymbol{x}}]\!]_{\mathcal{H}}$ is s_i.
5. Receive a vector $\boldsymbol{\Delta}_{\hat{\boldsymbol{x}}} \in \mathbb{F}^N$ from the adversary $\mathcal{S}$ and compute $[\![\hat{\boldsymbol{x}}]\!] := [\![\hat{\boldsymbol{x}}]\!]_{\mathcal{H}} + \boldsymbol{\Delta}_{\hat{\boldsymbol{x}}}$.
6. For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
   - If it is $(\textsc{Continue}, \mathsf{h})$, send its corresponding share of $[\![\hat{\boldsymbol{x}}]\!]$ to h.
   - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 6: The functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$

Protocol $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$

Let $[\![\boldsymbol{x}]\!]_d$ be the input degree-d PSS of a vector $\boldsymbol{x}$ that is to be permuted and let $p(\cdot)$ be the permutation function that is to be used.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$ to prepare a pair random shares $[\![\boldsymbol{r}]\!]_d$ and $[\![\hat{\boldsymbol{r}}]\!]_d$, where $\hat{\boldsymbol{r}} = p(\boldsymbol{r})$ if all the servers behave honestly.
2. The servers invoke $\mathcal{F}_{\mathsf{Rand}}$ to prepare a degree-$2d$ PSS $[\![\mathbf{0}]\!]_{2d}$ of 0.

Online phase:

1. Each server S_i computes $[\![\boldsymbol{m}]\!]_{2d} := [\![\boldsymbol{x}]\!]_d + [\![\boldsymbol{r}]\!]_d + [\![\mathbf{0}]\!]_{2d}$ and send $[\![\boldsymbol{m}]\!]_{2d}$ to S_1.
2. S_1 reconstructs to get the vector $\boldsymbol{m}$ and performs the permutation on $\boldsymbol{m}$ to get $\hat{\boldsymbol{m}} := p(\boldsymbol{m})$. Then S_1 computes $[\![\hat{\boldsymbol{m}}]\!]_d$ and distributes the shares to other servers.
3. Each server S_i locally computes $[\![\hat{\boldsymbol{x}}]\!]_d := [\![\hat{\boldsymbol{m}}]\!]_d - [\![\hat{\boldsymbol{r}}]\!]_d$.

Figure 7: The $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$ protocol

Theorem 4. *The protocol $\Pi_{\mathsf{PSSPermute\text{-}Mal}}$ depicted in Figure 31 securely realizes $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$ depicted in Figure 6 in the $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$-hybrid world against a malicious adversary corrupting up to t servers.*

Proof. We refer interested readers to see the proof in [GPS21a]. □

Verifying PSS permutation. In the work by Goyal et al. [GPS21a], they propose a technique called *network routing*, which is used to check whether a tuple $(\llbracket \boldsymbol{a} \rrbracket, \llbracket \boldsymbol{b} \rrbracket, i, j)$ is formed correctly, i.e., whether $a_i = b_j$ holds. In this work, we adapt the network routing technique into our setting and make some modifications; more precisely, our goal is to check whether a tuple $(\llbracket \boldsymbol{a} \rrbracket, \llbracket \boldsymbol{b} \rrbracket, p)$ is formed correctly, i.e., whether $\boldsymbol{b} = p(\boldsymbol{a})$.

Formally, we put the functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$ for verifying the PSS permutation in Figure 8 and present our protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$ for verifying the PSS permutation in Figure 9. The security of our protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$ through Theorem 5. Notice that, our protocol requires two functionalities $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Rand}}$, which can be found in Appendix B.1 and Appendix B.2, respectively. We also note that, in the preprocessing phase of our protocol, the servers need to invoke $\mathcal{F}_{\mathsf{Rand}}$ prepare k tuples $\{(\llbracket \boldsymbol{s}^{(i)} \rrbracket, \llbracket \boldsymbol{t}^{(i)} \rrbracket, i, p(i))\}_{i\in[k]}$ such that for $i \in [k]$, $\boldsymbol{s}^{(i)}$ and $\boldsymbol{t}^{(i)}$ are random and unknown to any server, and the i-th component of $\boldsymbol{s}^{(i)}$ is equal to the $p(i)$-th component of $\boldsymbol{t}^{(i)}$; we refer interested readers to see the details of this procedure in [Appendix B.5] [GPS21b].

Functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$

The functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{VerPermute}, m, \{\llbracket \boldsymbol{x}^{(i)} \rrbracket, \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket\}_{i\in[m]}, p(\cdot))$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where m is the number of PSS pairs that all servers want to verify, $\{\llbracket \boldsymbol{x}^{(i)} \rrbracket, \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket\}_{i\in[m]}$ are the corresponding PSS pairs and $p(\cdot)$ is a permutation, it does:

1. For $i \in [m]$:
   - Reconstruct $\boldsymbol{x}^{(i)}$ and $\hat{\boldsymbol{x}}^{(i)}$ using the shares from the honest servers.
   - Compute the new PSS $\llbracket \boldsymbol{x}^{(i)} \rrbracket_{\mathcal{H}}$ and $\llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket_{\mathcal{H}}$ of the secrets $\boldsymbol{x}^{(i)}$ and $\hat{\boldsymbol{x}}^{(i)}$, such that for each honest server, its share of $\llbracket \boldsymbol{x}^{(i)} \rrbracket$ (resp. $\llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket$) is equal to its share of $\llbracket \boldsymbol{x}^{(i)} \rrbracket_{\mathcal{H}}$ (resp. $\llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket_{\mathcal{H}}$). Send the corrupted servers' shares of $\llbracket \boldsymbol{x}^{(i)} \rrbracket_{\mathcal{H}}, \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{x}^{(i)}} := \llbracket \boldsymbol{x}^{(i)} \rrbracket - \llbracket \boldsymbol{x}^{(i)} \rrbracket_{\mathcal{H}} \in \mathbb{F}^N, \boldsymbol{\Delta}_{\hat{\boldsymbol{x}}^{(i)}} := \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket - \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{x}^{(i)}}$ (resp. $\boldsymbol{\Delta}_{\hat{\boldsymbol{x}}^{(i)}}$) describes the inconsistency of $\llbracket \boldsymbol{x}^{(i)} \rrbracket$ (resp. $\llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket$).
   - Compute $\boldsymbol{d}^{(i)} := p(\boldsymbol{x}^{(i)}) - \hat{\boldsymbol{x}}^{(i)}$ and send $\boldsymbol{d}^{(i)}$ to the adversary $\mathcal{S}$.
2. If there exists $i \in [m]$ such that $\boldsymbol{d}^{(i)} \neq \boldsymbol{0}^N$, set $b := 0$; otherwise, set $b := 1$.
3. Send b to the adversary $\mathcal{S}$. For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
   - If it is $(\textsc{Continue}, \mathsf{h})$, send b to h.
   - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 8: The functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$.

Theorem 5. *The protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$ depicted in Figure 9 securely realizes the functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$ depicted in Figure 8 in the $\{\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Coin}}\}$-hybrid world against a malicious adversary corrupting up to t servers.*

Proof. We denote by $\mathcal{A}$ the adversary. Here we will first provide the workflow of an ideal adversary (a.k.a., the simulator) $\mathcal{S}$ that simulates the behaviors of the honest parties. Then we will prove that the adversary $\mathcal{A}$ cannot distinguish whether it is interacting with the simulator $\mathcal{S}$ or the real servers.

Simulation strategy. We describe the simulation strategy of the simulator $\mathcal{S}$ as follows:

1. For $i \in [m]$, the simulator $\mathcal{S}$ receives the corrupted servers' shares of $\llbracket \boldsymbol{x}^{(i)} \rrbracket_{\mathcal{H}}, \llbracket \hat{\boldsymbol{x}}^{(i)} \rrbracket_{\mathcal{H}}, \boldsymbol{\Delta}_{\hat{\boldsymbol{x}}^{(i)}}, \boldsymbol{\Delta}_{\boldsymbol{x}^{(i)}}$ and $\boldsymbol{d}^{(i)}$ from $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$.
2. In the preprocessing phase, the simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Rand}}$; therefore, $\mathcal{S}$ knows the entire secrets $\{\boldsymbol{s}^{(i)}, \boldsymbol{t}^{(i)}\}_{i\in[k]}$ and all servers' corresponding shares.
3. In the online phase, for $i \in [k]$:
   (a) The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ and faithfully samples $r \in \mathbb{F}$.
   (b) The simulator $\mathcal{S}$ computes $\boldsymbol{\Delta}_{\boldsymbol{a}} := \sum_{j=1}^{m} r^j \cdot \boldsymbol{\Delta}_{\boldsymbol{x}^{(i)}}$, $\boldsymbol{\Delta}_{\boldsymbol{b}} := \sum_{j=1}^{m} r^j \cdot \boldsymbol{\Delta}_{\hat{\boldsymbol{x}}^{(i)}}$ and $\boldsymbol{d} := \sum_{j=1}^{m} r^j \cdot \boldsymbol{d}^{(j)}$. The simulator $\mathcal{S}$ computes the corrupted servers' shares of $\llbracket \boldsymbol{a} \rrbracket, \llbracket \boldsymbol{b} \rrbracket$ using the received corrupted servers' shares of $\{\llbracket \boldsymbol{x}^{(j)} \rrbracket_{\mathcal{H}}, \llbracket \hat{\boldsymbol{x}}^{(j)} \rrbracket_{\mathcal{H}}\}_{j\in[m]}$ and $\{\llbracket \boldsymbol{s}^{(j)} \rrbracket, \llbracket \boldsymbol{t}^{(j)} \rrbracket\}_{j\in[k]}$.

Protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$

Let $\{[\![\boldsymbol{x}^{(i)}]\!], [\![\hat{\boldsymbol{x}}^{(i)}]\!]\}_{i\in[m]}$ be the sets of PSS that the servers want to verify. Let $p(\cdot)$ be the permutation function.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\mathsf{Rand}}$ to prepare k tuples $\{([\![\boldsymbol{s}^{(i)}]\!], [\![\boldsymbol{t}^{(i)}]\!], i, p(i))\}_{i\in[k]}$ such that for $i \in [k]$, $\boldsymbol{s}^{(i)}$ and $\boldsymbol{t}^{(i)}$ are random and unknown to any server, and the i-th component of $\boldsymbol{s}^{(i)}$ is equal to the $p(i)$-th component of $\boldsymbol{t}^{(i)}$.

Online phase:

1. For $i \in [k]$, the servers perform the following checks:
   - All servers send $(\textsc{Flip}, 1)$ to $\mathcal{F}_{\mathsf{Coin}}$, which returns a uniformly random $r \in \mathbb{F}$ to them.
   - All servers locally compute $[\![\boldsymbol{a}]\!] := [\![\boldsymbol{s}^{(i)}]\!] + \sum_{j=1}^{m} r^j \cdot [\![\boldsymbol{x}^{(j)}]\!]$ and $[\![\boldsymbol{b}]\!] := [\![\boldsymbol{t}^{(i)}]\!] + \sum_{j=1}^{m} r^j \cdot [\![\hat{\boldsymbol{x}}^{(j)}]\!]$.
   - All servers reconstruct $\boldsymbol{a}$ and $\boldsymbol{b}$. If the reconstruction procedure fails, the servers simply abort; otherwise, the servers check if $a_i = b_{p(i)}$ holds.
2. If all the checks above pass, the servers accept the fact that $\hat{\boldsymbol{x}}^{(i)} = p(\boldsymbol{x}^{(i)})$ holds for all $i \in [m]$; otherwise, the servers reject.

Figure 9: The protocol $\Pi_{\mathsf{Verify\text{-}PSSPermute}}$.

(c) The simulator $\mathcal{S}$ samples a random $\boldsymbol{a} \in \mathbb{F}^k$ as the secret. Basing on $\boldsymbol{a}$, the corrupted servers' shares of $[\![\boldsymbol{a}]\!]$, and the inconsistency error $\boldsymbol{\Delta_a}$, the simulator $\mathcal{S}$ can reconstruct all servers' shares of $[\![\boldsymbol{a}]\!]$.

(d) As for $[\![\boldsymbol{b}]\!]$, the simulator $\mathcal{S}$ samples a random $\boldsymbol{b} \in \mathbb{F}^k$ such that $b_{p(i)} = a_i + d_i$ while the other $k-1$ positions of $\boldsymbol{b}$ are all random. Basing on $\boldsymbol{b}$, the corrupted servers' shares of $[\![\boldsymbol{b}]\!]$, and the inconsistency error $\boldsymbol{\Delta_b}$, the simulator $\mathcal{S}$ can reconstruct all servers' shares of $[\![\boldsymbol{b}]\!]$.

4. $\mathcal{S}$ completes the checks acting as the honest servers. $\mathcal{S}$ sets $b' := 1$ if all the checks pass; otherwise, $\mathcal{S}$ sets $b' := 0$.

5. $\mathcal{S}$ receives b from $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$. If $b \neq b'$, $\mathcal{S}$ aborts.

6. If an honest server aborts, $\mathcal{S}$ sends $\textsc{Abort}$ to $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$; otherwise, $\mathcal{S}$ sends $\textsc{Continue}$ to $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$.

Indistinguishability proof. We prove the indistinguishability through the following hybrids.

- Hybrid Hybr_0: This is the real-world execution.
- Hybrid Hybr_1: Same as hybrid Hybr_0, except that $\mathcal{S}$ executes Step 1-3 of the above simulation strategy.

  Lemma 1. *Hybrid* Hybr_1 *is perfectly indistinguishable from* Hybr_0.

  Proof. Here we show that the simulated $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$ are perfectly indistinguishable from the real $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$ for each $i \in [k]$ in Step 3. Since these k repetitions are the same, we only consider the case in the i-th repetition. Due to Step 1-2, $\mathcal{S}$ can compute the corrupted servers' shares of $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$. Furthermore, $\mathcal{S}$ can compute $d_i = b_{p(i)} - a_i$, which is the additive error. To determine the whole sharings of $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$, $\mathcal{S}$ only needs to know the distribution of $\boldsymbol{a}, \boldsymbol{b}$. In Hybr_0 (the real-world), $\boldsymbol{a}, \boldsymbol{b}$ are masked by $\boldsymbol{s}^{(i)}, \boldsymbol{t}^{(i)}$ which are random subject to $s_i^{(i)} = t_{p(i)}^{(i)}$. Therefore, we conclude that $\boldsymbol{a}$ is a uniformly random vector while $\boldsymbol{b}$ is a vector whose $p(i)$-th component is $b_{p(i)} = a_i + d_i$ and all other components are uniformly random. On the other hand, in hybrid Hybr_1, $\mathcal{S}$ randomly samples $\boldsymbol{a}$ and $\{b_j\}_{j\neq p(i)}$ and sets $b_{p(i)} := a_i + d_i$, which has the same distribution as that in Hybr_0. In conclusion, Hybr_1 is perfectly indistinguishable from Hybr_0. □

- Hybrid Hybr_2: Same as hybrid Hybr_1, except that $\mathcal{S}$ executes Step 4-6 of the above simulation strategy.

  Lemma 2. *Let* $\mathbb{F}$ *be a large prime field such that* $|\mathbb{F}|^{-1} = \mathsf{negl}(\lambda)$. *Hybrid* Hybr_2 *is statistically indistinguishable from* Hybr_1.

Proof. We observe that the adversary $\mathcal{A}$ can distinguish these two hybrids if and only if the simulator $\mathcal{S}$ aborts, which would happen when there exists a $i \in [m]$ such that $p(\boldsymbol{x^{(i)}}) \neq \hat{\boldsymbol{x}}^{(i)}$ but $a_j = b_{p(j)}$ holds for all $j \in [k]$. It is easy to see that it is sufficient to show that if there exists a $i \in [m]$ such that $\boldsymbol{d^{(i)}} \neq \mathbf{0}^N$, then with overwhelming probability we have $\boldsymbol{d} \neq \mathbf{0}^N$.

Since $\boldsymbol{d^{(i)}} \neq \mathbf{0}^N$, there must exist a $j \in [N]$ such that $d_j^{(i)} \neq 0$. Then let us consider a polynomial $h_j(r) = \sum_{t=1}^{m} r^t \cdot d_j^{(t)}$. Since $d_j^{(i)} \neq 0$, $h_j(r)$ is a non-zero polynomial. By Schwartz-Zippel lemma [Sch80, Zip79], we conclude $\Pr[h_j(r) = 0] \leq \frac{m}{|\mathbb{F}|}$ which is negligible. It is easy to see that, $h_j(r)$ corresponds to the j-th component of $\boldsymbol{d}$; thus, we show that $\boldsymbol{d} \neq \mathbf{0}^N$ with overwhelming probability. In conclusion, Hybr_2 is statistically indistinguishable from Hybr_1. □

Hybrid Hybr_2 is the ideal-world execution. Therefore, we prove that the real-world execution is statistically indistinguishable from the ideal-world execution, which completes the proof. □

7.2 Malicious-secure distributed multi-scalar multiplication

Similar to Section 7.1, in this subsection, we will first present a malicious-secure version of functionality for distributed Multi-Scalar Multiplication (dMSM), which is denoted by $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$. In $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$, the adversary is allowed to add errors into the honest parties' received shares. Then we will provide a malicious-secure protocol $\Pi_{\mathsf{dMSM\text{-}Mal}}$ that securely-realizes $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$. To detect the corrupted servers' behaviors, we will present functionality for verifying the dMSM, which is denoted by $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$, and we will present a protocol $\Pi_{\mathsf{Verify\text{-}dMSM}}$ that securely realizes $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$.

Distributed multi-scalar multiplication against a malicious adversary. We put the malicious-secure version of functionality $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$ for dMSM in Figure 10. Then we make some minor modifications to our semi-honest protocol $\Pi_{\mathsf{dMSM\text{-}Semi}}$ and turn it into a malicious protocol $\Pi_{\mathsf{dMSM\text{-}Mal}}$, which is depicted in Figure 11. Notice that, in the preprocessing phase of the protocol, we let the servers invoke a functionality $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ to generate a pair of random shares $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$ and a functionality $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$ to transform a packed secret sharing $[\![\boldsymbol{r}]\!]_d$ to a normal Shamir secret sharing $\langle r_1 \rangle, ..., \langle r_k \rangle$. The security of our protocol $\Pi_{\mathsf{dMSM\text{-}Mal}}$ is proven secure through Theorem 6.

Functionality $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$

The functionality $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of corrupted servers. Let $\mathcal{H}$ be the set of honest servers.

Upon receiving $(\textsc{Mult}, m, [\![\boldsymbol{A_1}]\!], \ldots, [\![\boldsymbol{A_m}]\!], [\![\boldsymbol{b_1}]\!], \ldots, [\![\boldsymbol{b_m}]\!])$ from the servers, where m is the number of pairs, do:

1. For each $i \in [m]$:
   - Reconstruct the correct $(A_{(i-1)k+1}, ..., A_{ik})$ and $(b_{(i-1)k+1}, ..., b_{ik})$ from the honest servers' shares of $[\![\boldsymbol{A_i}]\!]$ and $[\![\boldsymbol{b_i}]\!]$, respectively.
   - Compute the new PSS $[\![\boldsymbol{A_i}]\!]_{\mathcal{H}}$ and $[\![\boldsymbol{b_i}]\!]_{\mathcal{H}}$ of the secrets $(A_{(i-1)k+1}, ..., A_{ik})$ and $(b_{(i-1)k+1}, ..., b_{ik})$, such that for each honest server, its share of $[\![\boldsymbol{A_i}]\!]$ (resp. $[\![\boldsymbol{b_i}]\!]$) is equal to its share of $[\![\boldsymbol{A_i}]\!]_{\mathcal{H}}$ (resp. $[\![\boldsymbol{b_i}]\!]_{\mathcal{H}}$). Send the corrupted servers' shares of $[\![\boldsymbol{A_i}]\!]_{\mathcal{H}}, [\![\boldsymbol{b_i}]\!]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{A_i}} := [\![\boldsymbol{A_i}]\!] - [\![\boldsymbol{A_i}]\!]_{\mathcal{H}} \in \mathbb{F}^N, \boldsymbol{\Delta}_{\boldsymbol{b_i}} := [\![\boldsymbol{b_i}]\!] - [\![\boldsymbol{b_i}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{A_i}}$ (resp. $\boldsymbol{\Delta}_{\boldsymbol{b_i}}$) describes the inconsistency of $[\![\boldsymbol{A_i}]\!]$ (resp. $[\![\boldsymbol{b_i}]\!]$).
2. Receive the additive error $h \in \mathbb{F}$ from the adversary $\mathcal{S}$, and compute $\mathsf{out} := \prod_{i \in [m]} A_i^{b_i} + h$.
3. Receive a set of shares $\{u_i\}_{i \in \mathcal{C}orr}$ from the adversary. Sample a random sharing $\langle \mathsf{out} \rangle_{\mathcal{H}}$ of out, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_i\}_{i \in \mathcal{C}orr}$.
4. Receive a vector $\boldsymbol{\Delta}_{\mathsf{out}} \in \mathbb{F}^N$ from the adversary $\mathcal{S}$ and compute $\langle \mathsf{out} \rangle := \langle \mathsf{out} \rangle_{\mathcal{H}} + \boldsymbol{\Delta}_{\mathsf{out}}$.
5. For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
   - If it is $(\textsc{Continue}, \mathsf{h})$, send its corresponding share of $\langle \mathsf{out} \rangle$ to h.
   - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 10: The functionality $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

Protocol $\Pi_{\mathsf{dMSM\text{-}Mal}}$

Let $A_1, \dots, A_n$ be n group elements in $\mathbb{G}$ and $b_1, \dots, b_n$ be n field elements in $\mathbb{F}$. The protocol allows N servers to collaboratively compute $\prod_{i=1}^{n} A_i^{b_i}$ in a distributed manner. Without loss of generality, we assume $k|n$, where k is the packing factor, and we set $m := n/k$. Each server holds packed secret shares $[\![\boldsymbol{A_j}]\!]$ of vectors $\boldsymbol{A}_j = \{A_{(j-1)k+i}\}_{i\in[k]}$ and $[\![\boldsymbol{b_j}]\!]$ of vectors $\boldsymbol{b}_j = \{b_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [m]$, respectively.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ to prepare a pair of random shares $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$, where $\boldsymbol{r} \in \mathbb{F}^k$ is a random vector unknown to any server S_i.
2. The servers send $[\![\boldsymbol{r}]\!]_d$ to $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$, which returns $\langle r_1 \rangle, \dots, \langle r_k \rangle$ to the servers.

Online phase:

1. Each server S_i computes $[\![\boldsymbol{C}]\!]_{2d} = \prod_{j\in[m]} [\![\boldsymbol{A_j}]\!]_d^{[\![\boldsymbol{b_j}]\!]_d}$.
2. Each server S_i computes $[\![\boldsymbol{D}]\!]_{2d} = [\![\boldsymbol{C}]\!]_{2d} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$ and send it to S_1.
3. S_1 receives enough shares and reconstructs $\boldsymbol{D} = (D_1, \dots, D_k)$.
4. S_1 computes $E = \prod_{j\in[k]} D_j$ and send it to each server.
5. Each server computes $\langle \mathsf{out} \rangle = \frac{E}{\prod_{j\in[k]} g^{\langle r_j \rangle}}$ as the output.

Figure 11: The $\Pi_{\mathsf{dMSM\text{-}Mal}}$ Protocol .

Theorem 6. *The protocol $\Pi_{\mathsf{dMSM\text{-}Mal}}$ depicted in Figure 11 securely realizes $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$ depicted in Figure 10 in the $\{\mathcal{F}_{\mathsf{Double\text{-}Rand}}, \mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}\}$-hybrid world against a malicious adversary corrupting up to t servers.*

Proof. We denote by $\mathcal{A}$ the adversary. Here we will first provide the workflow of an ideal adversary (a.k.a., the simulator) $\mathcal{S}$ that simulates the behaviors of the honest parties. Then we will prove that the adversary $\mathcal{A}$ cannot distinguish whether it is interacting with the simulator $\mathcal{S}$ or the real servers.

Simulation strategy. We describe the simulation strategy of the simulator $\mathcal{S}$ as follows:

1. The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ and $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$ honestly for the servers; therefore, the servers knows the whole sharings of $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}, \langle r_1 \rangle, \dots, \langle r_k \rangle$. In addition, during the emulation of $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$, $\mathcal{S}$ receives the additive error $\boldsymbol{d}$ to $\boldsymbol{r}$ and the additive error $\boldsymbol{\Delta}_j$ to $\langle x_j \rangle_{\mathcal{H}}$ for each $j \in [k]$.

2. For each $i \in [m]$: The simulator $\mathcal{S}$ receives the corrupted servers' shares of $[\![\boldsymbol{A_i}]\!]$ and $[\![\boldsymbol{b_i}]\!]$ and $\boldsymbol{\Delta}_{\boldsymbol{A_i}} := [\![\boldsymbol{A_i}]\!] - [\![\boldsymbol{A_i}]\!]_{\mathcal{H}} \in \mathbb{F}^N$, $\boldsymbol{\Delta}_{\boldsymbol{b_i}} := [\![\boldsymbol{b_i}]\!] - [\![\boldsymbol{b_i}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ from $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

3. For each honest server, the simulator $\mathcal{S}$ samples a random element in $\mathbb{G}$ as its share of $[\![\boldsymbol{D}]\!]_{2d}$.

4. The simulator $\mathcal{S}$ computes the shares of $[\![\boldsymbol{D}]\!]_{2d}$ that the corrupted servers should hold by $[\![\boldsymbol{D}]\!]_{2d} = \prod_{j\in[\frac{n}{k}]} [\![\boldsymbol{A_j}]\!]_d^{[\![\boldsymbol{b_j}]\!]_d} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$. Let $[\![\hat{\boldsymbol{D}}]\!]_{2d} = \prod_{j\in[\frac{n}{k}]} [\![\boldsymbol{A_j}]\!]_{\mathcal{H}}^{[\![\boldsymbol{b_j}]\!]_{\mathcal{H}}} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$. Then $[\![\hat{\boldsymbol{D}}]\!]_{2d} = \prod_{j\in[\frac{n}{k}]} ([\![\boldsymbol{A_j}]\!] - \boldsymbol{\Delta}_{\boldsymbol{A_j}})^{([\![\boldsymbol{b_j}]\!] - \boldsymbol{\Delta}_{\boldsymbol{b_j}})} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$. Therefore, $\mathcal{S}$ can determine the whole sharings of $[\![\hat{\boldsymbol{D}}]\!]_{2d}$ and reconstruct $\hat{\boldsymbol{D}}$.

5. The simulator $\mathcal{S}$ honestly follows the protocol and learns E. Then $\mathcal{S}$ computes $\hat{E} := \prod_{j\in[k]} \hat{D}_j$ and $h := \frac{E - \hat{E}}{g^r}$. The simulator $\mathcal{S}$ sends h to $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

6. The simulator $\mathcal{S}$ computes the whole sharings of $\langle \mathsf{out} \rangle$ and sends the corrupted servers' share of $\langle \mathsf{out} \rangle$ to $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

7. The simulator $\mathcal{S}$ computes $\boldsymbol{\Delta}_{\mathsf{out}}$ using the whole sharings of $\langle \mathsf{out} \rangle$ and the additive errors received from the adversary when emulating $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$, then sends $\boldsymbol{\Delta}_{\mathsf{out}}$ to $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

8. If an honest server aborts, $\mathcal{S}$ sends ABORT to $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$; otherwise, $\mathcal{S}$ sends CONTINUE to $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$.

Indistinguishability proof. We prove the indistinguishability through the following hybrids.

- Hybrid Hybr_0: This is the real-world execution.

- Hybrid Hybr_1: Same as hybrid Hybr_0, except that $\mathcal{S}$ executes Step 1-3 of the above simulation strategy.

  Lemma 3. *Hybrid* Hybr_1 *is perfectly indistinguishable from hybrid* Hybr_0.

  Proof. Here, we prove that the simulated honest servers' shares of $[\![\boldsymbol{D}]\!]_{2d}$ are perfectly indistinguishable from the real honest servers' shares. In hybrid Hybr_1, for each honest server, the simulator $\mathcal{S}$ samples a random group element in $\mathbb{G}$ as its simulated share of $[\![\boldsymbol{D}]\!]_{2d}$. Note that, $[\![\boldsymbol{D}]\!]_{2d} = [\![\boldsymbol{C}]\!]_{2d} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$. As proven in [GPS21b, Lemma 3], given the corrupted parties' shares of $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$, the honest parties' shares of $[\![\boldsymbol{r}]\!]_{2d}$ are uniformly random. Therefore, it is easy to conclude that the honest servers' shares of $[\![\boldsymbol{D}]\!]_{2d}$ are also uniformly random. Thus, in hybrid Hybr_0 (real-world), the real honest servers' shares of $[\![\boldsymbol{D}]\!]_{2d}$ are uniformly random. In conclusion, the simulated honest servers' shares of $[\![\boldsymbol{D}]\!]_{2d}$ are perfectly indistinguishable from the real honest servers' shares; thus, hybrid Hybr_1 is perfectly indistinguishable from hybrid Hybr_0. □

- Hybrid Hybr_2: Same as hybrid Hybr_1, except that $\mathcal{S}$ executes Step 4-8 of the above simulation strategy. Perfect indistinguishability is trivial since $\mathcal{S}$ can determine the corrupted servers' shares.

Hybrid Hybr_2 is the ideal-world execution. Therefore, we prove that the real-world execution is perfectly indistinguishable from the ideal-world execution, which completes the proof. □

Verifying distributed multi-scalar multiplication. Here we adapt the previous technique for verifying the PSS permutation into verifying the dMSM. Formally, we present the ideal functionality $\mathcal{F}_{\textsf{Verify-dMSM}}$ for verifying dMSM in Figure 12, and we put our protocol $\Pi_{\textsf{Verify-dMSM}}$ in Figure 13. Notice that, in the preprocessing phase of our protocol, we let the servers invoke a functionality $\mathcal{F}_{\textsf{Rand-dMSM}}$ to prepare a random tuple $([\![\boldsymbol{X}]\!]_d, [\![\boldsymbol{y}]\!]_d, \langle z\rangle_d)$ such that $z = \prod_{i\in[k]} X_i^{y_i}$; the full descriptions of $\mathcal{F}_{\textsf{Rand-dMSM}}$ can be found in Appendix B.2. The security of our protocol $\Pi_{\textsf{Verify-dMSM}}$ is proven through Theorem 7.

Functionality $\mathcal{F}_{\textsf{Verify-dMSM}}$

The functionality $\mathcal{F}_{\textsf{Verify-dMSM}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{VerMSM}, m_1, m_2, \{\{[\![\boldsymbol{A}_i^{(j)}]\!], [\![\boldsymbol{b}_i^{(j)}]\!]\}_{i\in[m_1]}, \langle \mathsf{out}^{(j)}\rangle\}_{j\in[m_2]})$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where m_1, m_2 denote the number of inputs, it does:

1. For $j \in [m_2]$:
   - For $i \in [m_1]$: Reconstruct $\boldsymbol{A}_i^{(j)}$ and $\boldsymbol{b}_i^{(j)}$ using the shares from the honest servers, where $\boldsymbol{A}_i^{(j)} = (A_{(i-1)k+1}^{(j)}, \ldots, A_{ik}^{(j)})$ and $\boldsymbol{b}_i^{(j)} = (b_{(i-1)k+1}^{(j)}, \ldots, b_{ik}^{(j)})$.
   - For $i \in [m_1]$: Compute the new PSS $[\![\boldsymbol{A}_i^{(j)}]\!]_{\mathcal{H}}$ and $[\![\boldsymbol{b}_i^{(j)}]\!]_{\mathcal{H}}$ of the secrets $\boldsymbol{A}_i^{(j)}$ and $\boldsymbol{b}_i^{(j)}$, such that for each honest server, its share of $[\![\boldsymbol{A}_i^{(j)}]\!]$ (resp. $[\![\boldsymbol{b}_i^{(j)}]\!]$) is equal to its share of $[\![\boldsymbol{A}_i^{(j)}]\!]_{\mathcal{H}}$ (resp. $[\![\boldsymbol{b}_i^{(j)}]\!]_{\mathcal{H}}$). Send the corrupted servers' shares of $[\![\boldsymbol{A}_i^{(j)}]\!]_{\mathcal{H}}, [\![\boldsymbol{b}_i^{(j)}]\!]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{A}_i^{(j)}} := [\![\boldsymbol{A}_i^{(j)}]\!] - [\![\boldsymbol{A}_i^{(j)}]\!]_{\mathcal{H}} \in \mathbb{F}^N$, $\boldsymbol{\Delta}_{\boldsymbol{b}_i^{(j)}} := [\![\boldsymbol{b}_i^{(j)}]\!] - [\![\boldsymbol{b}_i^{(j)}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{A}_i^{(j)}}$ (resp. $\boldsymbol{\Delta}_{\boldsymbol{b}_i^{(j)}}$) describes the inconsistency of $[\![\boldsymbol{A}_i^{(j)}]\!]$ (resp. $[\![\boldsymbol{b}_i^{(j)}]\!]$).
   - Reconstruct $\mathsf{out}^{(j)}$ using the shares from the honest servers. Compute the new share $\langle \mathsf{out}^{(j)}\rangle_{\mathcal{H}}$ of the secret $\mathsf{out}^{(j)}$ such that for each honest server, its share of $\langle \mathsf{out}^{(j)}\rangle$ is equal to its share of $\langle \mathsf{out}^{(j)}\rangle$. Send the corrupted servers' shares of $\langle \mathsf{out}^{(j)}\rangle_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\mathsf{out}^{(j)}} := \langle \mathsf{out}^{(j)}\rangle - \langle \mathsf{out}^{(j)}\rangle_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$.
   - Send $d^{(j)} := \mathsf{out}^{(j)} - \prod_{i\in[m_1\cdot k]} (A_i^{(j)})^{(b_i^{(j)})}$ to the adversary $\mathcal{S}$.
2. If there exists $j \in [m_2]$ such that $d^{(j)} \neq 0$, set $b := 0$; otherwise, set $b := 1$.
3. Send b to the adversary $\mathcal{S}$. For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
   - If it is $(\textsc{Continue}, \mathsf{h})$, send b to h.
   - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 12: The functionality $\mathcal{F}_{\textsf{Verify-dMSM}}$.

Theorem 7. *The protocol* $\Pi_{\textsf{Verify-dMSM}}$ *depicted in Figure 13 securely realizes* $\mathcal{F}_{\textsf{Verify-dMSM}}$ *in the* $\{\mathcal{F}_{\textsf{Rand-dMSM}}, \mathcal{F}_{\textsf{Coin}}\}$*-hybrid world against a malicious adversary corrupting up to t servers.*

Protocol $\Pi_{\mathsf{Verify\text{-}dMSM}}$

For $j \in [m_2]$, let $A_1^{(j)}, ..., A_n^{(j)}, \mathsf{out}^{(j)}$ be $n+1$ group elements in $\mathbb{G}$ and $b_1^{(j)}, ..., b_n^{(j)}$ be n field elements in $\mathbb{F}$. Without loss of generality, we assume $k|n$, where k is the packing factor, and we set $m_1 := n/k$. The servers hold $\{\{[\![\boldsymbol{A}_i^{(j)}]\!]_d, [\![\boldsymbol{b}_i^{(j)}]\!]_d\}_{i\in[m_1]}, \langle \mathsf{out}^{(j)}\rangle_d\}_{j\in[m_2]}$, where $\boldsymbol{A}_i^{(j)} = (A_{(i-1)k+1}^{(j)}, \ldots, A_{ik}^{(j)})$ and $\boldsymbol{b}_i^{(j)} = (b_{(i-1)k+1}^{(j)}, \ldots, b_{ik}^{(j)})$. The servers want to verify if $\mathsf{out}^{(j)} = \prod_{i\in[n]} (A_i^{(j)})^{b_i^{(j)}}$ holds for all $j \in [m_2]$.

Preprocessing phase:

1. The servers send RANDMULT to $\mathcal{F}_{\mathsf{Rand\text{-}dMSM}}$, which returns random $[\![\boldsymbol{X}]\!]_d, [\![\boldsymbol{y}]\!]_d, \langle z\rangle_d$ to the servers, where $\boldsymbol{X} = (X_1, \ldots, X_k)$ and $\boldsymbol{y} = (y_1, \ldots, y_k)$ for $j \in [m_1]$ and $z = \prod_{i\in[k]} X_i^{y_i}$ holds.

Online phase:

1. For $j \in [m_2]$: All servers locally compute $[\![\boldsymbol{C}^{(j)}]\!]_{2d} = \prod_{i\in[m_1]} [\![\boldsymbol{A}_i^{(j)}]\!]_d^{[\![\boldsymbol{b}_i^{(j)}]\!]_d}$.
2. All servers send $(\text{FLIP}, 1)$ to $\mathcal{F}_{\mathsf{Coin}}$, which returns a uniformly random $r \in \mathbb{F}$ to them.
3. All servers locally compute $[\![\hat{\boldsymbol{C}}]\!]_{2d} := [\![\boldsymbol{X}]\!]_d^{[\![\boldsymbol{y}]\!]_d} \cdot \prod_{j\in[m_2]} ([\![\boldsymbol{C}^{(j)}]\!]_{2d})^{r^j}$ and $\langle \hat{\mathsf{out}}\rangle_d := \langle z\rangle_d \cdot \prod_{j\in[m_2]} (\langle \mathsf{out}^{(j)}\rangle_d)^{r^j}$.
4. All servers reconstruct $\hat{\boldsymbol{C}}$ and $\hat{\mathsf{out}}$. If the reconstruction procedure fails, the servers simply abort; otherwise, the servers check if $\hat{\mathsf{out}} = \prod_{i\in[k]} \hat{C}_i$ holds. If the check passes, the servers accept the fact that $\mathsf{out}^{(j)} = \prod_{i\in[n]} (A_i^{(j)})^{b_i^{(j)}}$ holds for all $j \in [m_2]$; otherwise, the servers reject.

Figure 13: The protocol $\Pi_{\mathsf{Verify\text{-}dMSM}}$.

Proof. The proof is similar to the proof of Theorem 5. We denote by $\mathcal{A}$ the adversary. Here we will first provide the workflow of an ideal adversary (a.k.a., the simulator) $\mathcal{S}$ that simulates the behaviors of the honest parties. Then we will prove that the adversary $\mathcal{A}$ cannot distinguish whether it is interacting with the simulator $\mathcal{S}$ or the real servers.

Simulation strategy. We describe the simulation strategy of the simulator $\mathcal{S}$ as follows:

1. In the beginning, the simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Rand\text{-}dMSM}}$ for the adversary $\mathcal{A}$. Therefore, $\mathcal{S}$ knows the whole sharings of $[\![\boldsymbol{X}]\!]_d, [\![\boldsymbol{y}]\!]_d, \langle z\rangle_d$.
2. For $i \in [m_1], j \in [m_2]$: $\mathcal{S}$ receives the corrupted servers' shares of $[\![\boldsymbol{A}_i^{(j)}]\!]_{\mathcal{H}}$, $[\![\boldsymbol{b}_i^{(j)}]\!]_{\mathcal{H}}$ and $\langle \mathsf{out}^{(j)}\rangle_{\mathcal{H}}$, the inconsistency error $\boldsymbol{\Delta}_{\boldsymbol{A}_i^{(j)}}, \boldsymbol{\Delta}_{\boldsymbol{b}_i^{(j)}}$ and $\boldsymbol{\Delta}_{\mathsf{out}^{(j)}}$, and the additive error $d^{(j)}$ from $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$.
3. The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ and faithfully samples a random $r \in \mathbb{F}$.
4. For $j \in [m_2]$:
   (a) The simulator $\mathcal{S}$ computes the corrupted servers' shares of $[\![\boldsymbol{C}^{(j)}]\!]_{2d}$ and $\langle \mathsf{out}^{(j)}\rangle_d$.
   (b) The simulator $\mathcal{S}$ samples a random group element vector $\boldsymbol{C}^{(j)}$ as the secret. Basing on $\boldsymbol{C}^{(j)}$, the corrupted servers' shares of $[\![\boldsymbol{C}^{(j)}]\!]_{2d}$, the inconsistency errors $\{\boldsymbol{\Delta}_{\boldsymbol{A}_i^{(j)}}, \boldsymbol{\Delta}_{\boldsymbol{b}_i^{(j)}}\}_{i\in[m_1]}$, the simulator $\mathcal{S}$ can determine the whole sharings of $[\![\boldsymbol{C}^{(j)}]\!]_{2d}$.
   (c) The simulator $\mathcal{S}$ computes $\mathsf{out}^{(j)} := d^{(j)} + \prod_{i\in[k]} C_i^{(j)}$ and sets $\mathsf{out}^{(j)}$ as the secret. Basing on $\mathsf{out}^{(j)}$, the corrupted servers' shares of $\langle \mathsf{out}^{(j)}\rangle_d$, the inconsistency error $\boldsymbol{\Delta}_{\mathsf{out}^{(j)}}$, the simulator $\mathcal{S}$ can determine the whole sharings of $\langle \mathsf{out}^{(j)}\rangle_d$.
5. The simulator $\mathcal{S}$ computes the whole sharings of $[\![\hat{\boldsymbol{C}}]\!]_{2d}$ and $\langle \hat{\mathsf{out}}\rangle_d$.
6. The simulator $\mathcal{S}$ honestly complete the protocol. If the check passes, $\mathcal{S}$ sets $b' := 1$; otherwise, $\mathcal{S}$ sets $b' := 0$.
7. The simulator $\mathcal{S}$ receives b from $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$. If $b \neq b'$, $\mathcal{S}$ aborts.
8. If an honest server aborts, $\mathcal{S}$ sends ABORT to $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$; otherwise, $\mathcal{S}$ sends CONTINUE to $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$.

Indistinguishability proof. We prove the indistinguishability through the following hybrids.

- Hybrid Hybr_0: This is the real-world execution.
- Hybrid Hybr_1: Same as hybrid Hybr_0, except that $\mathcal{S}$ executes Step 1-5 of the above simulation strategy.

  Lemma 4. *Hybrid* Hybr_1 *is perfectly indistinguishable from* Hybr_0.

  Proof. Here we show that the simulated $[\![\hat{\boldsymbol{C}}]\!]_{2d}, \langle \hat{\mathsf{out}} \rangle_d$ are perfectly indistinguishable from the real $[\![\hat{\boldsymbol{C}}]\!]_{2d}, \langle \hat{\mathsf{out}} \rangle_d$. Due to Step 1-2, $\mathcal{S}$ can compute the corrupted servers' shares of $[\![\hat{\boldsymbol{C}}]\!]_{2d}, \langle \hat{\mathsf{out}} \rangle_d$. To determine the whole sharings of $[\![\hat{\boldsymbol{C}}]\!]_{2d}, \langle \hat{\mathsf{out}} \rangle_d$, $\mathcal{S}$ only needs to know the distribution of $\hat{\boldsymbol{C}}, \hat{\mathsf{out}}$. Notice that, $\mathcal{S}$ receives $d^{(j)} = \mathsf{out}^{(j)} - \prod_{i \in [m_1 \cdot k]} (A_i^{(j)})^{(b_i^{(j)})}$ for $j \in [m_2]$. In Hybr_0 (the real-world), $\hat{\boldsymbol{C}}, \hat{\mathsf{out}}$ are masked by $\prod_{i \in [k]} X_i^{y_i}$ and z respectively, which are random subject to $z = \prod_{i \in [k]} X_i^{y_i}$. Therefore, we conclude that $\hat{\boldsymbol{C}}$ is a uniformly random vector while $\hat{\mathsf{out}}$ is a group element such that $\hat{\mathsf{out}} = z \cdot \prod_{j \in [m_2]} (d^{(j)} + \prod_{i \in [k]} C_i^{(j)})^{r^j}$. On the other hand, in hybrid Hybr_1, $\mathcal{S}$ randomly samples $\boldsymbol{C}^{(j)}$ for $j \in [m_2]$; thus, it is easy to see that $\hat{\boldsymbol{C}}$ is also uniformly random, which has the same distribution as that in Hybr_0. As for $\hat{\mathsf{out}}$, $\mathcal{S}$ computes $\hat{\mathsf{out}} = z \cdot \prod_{j \in [m_2]} (d^{(j)} + \prod_{i \in [k]} C_i^{(j)})^{r^j}$, which also has the same distribution as that in Hybr_0. In conclusion, Hybr_1 is perfectly indistinguishable from Hybr_0. □

- Hybrid Hybr_2: Same as hybrid Hybr_1, except that $\mathcal{S}$ executes Step 4-6 of the above simulation strategy.

  Lemma 5. *Hybrid* Hybr_2 *is perfectly indistinguishable from* Hybr_1.

  Proof. We observe that the adversary $\mathcal{A}$ can distinguish these two hybrids if and only if the simulator $\mathcal{S}$ aborts, which would happen when there exists a $j \in [m_2]$ such that $\mathsf{out}^{(j)} \neq \prod_{i \in [m_1 \cdot k]} (A_i^{(j)})^{(b_i^{(j)})}$ but $\hat{\mathsf{out}} = \prod_{i \in [k]} \hat{C}_i$ holds. It is easy to see that it is sufficient to show that if there exists a $j \in [m_2]$ such that $d^{(j)} \neq 0$, then we have $z \cdot \prod_{j \in [m_2]} (d^{(j)} + \prod_{i \in [k]} C_i^{(j)})^{r^j} \neq \prod_{i \in [k]} \hat{C}_i$.

  Since $\prod_{i \in [k]} \hat{C}_i = \prod_{i \in [k]} X_i^{y_i} \cdot \prod_{i \in [k], j \in [m_2]} (C_i^{(j)})^{r^j}$ and $\prod_{i \in [k]} X_i^{y_i} = z$, our goal becomes to show that $\prod_{i \in [k], j \in [m_2]} (C_i^{(j)})^{r^j} \neq \prod_{j \in [m_2]} (d^{(j)} + \prod_{i \in [k]} C_i^{(j)})^{r^j}$, if there exists a $j \in [m_2]$ such that $d^{(j)} \neq 0$, which is trivial. In conclusion, Hybr_2 is perfectly indistinguishable from Hybr_1. □

Hybrid Hybr_2 is the ideal-world execution. Therefore, we prove that the real-world execution is indistinguishable from the ideal-world execution, which completes the proof. □

7.3 Malicous-secure packed secret sharing multiplication

In this subsection, we discuss how to handle PSS multiplication against a malicious adversary. The malicious secure version of $\mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}$, which is denoted as $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$, is depicted in Appendix B.3. In [GPS21a], Goyal et al. propose a protocol that can securely realize $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$. For completeness, we present this protocol Π_{PSSMult} in Figure 14.

Theorem 8. *The protocol* Π_{PSSMult} *depicted in Figure 14 securely realizes* $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$ *depicted in Figure 29 in the* $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$*-hybrid world against a malicious adversary corrupting up to t servers.*

Proof. we refer interested readers to see the proof in [GPS21a]. □

Verifying PSS multiplication. In [GPS21a], Goyal et al. showed how to detect malicious behaviors during the process of Π_{PSSMult}. More precisely, they propose an ideal functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ and show how to realize this functionality. Formally, we present the ideal functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ in Figure 15. We refer interested readers to see the protocol that securely realizes $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ in [GPS21a] and present a formal theorem in Theorem 9.

Theorem 9. *There exists a protocol that can securely realize* $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ *depicted in Figure 15 against a malicious adversary corrupting up to t servers.*

Proof. we refer interested readers to see the proof in [GPS21a]. □

Protocol Π_{PSSMult}

Let $[\![\boldsymbol{x}]\!]_d, [\![\boldsymbol{y}]\!]_d$ be the input degree-d packed secret shares of vectors $\boldsymbol{a}, \boldsymbol{b}$ that are to be element-wise multiplied.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ to prepare a pair of random shares $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$, where $\boldsymbol{r} \in \mathbb{F}^k$ is a random vector unknown to any server S_i.

Online phase:

1. Each server S_i computes $[\![\boldsymbol{e}]\!]_{2d} = [\![\boldsymbol{x}]\!]_d \cdot [\![\boldsymbol{y}]\!]_d + [\![\boldsymbol{r}]\!]_{2d}$ and send it to S_1.
2. S_1 receives enough shares and reconstructs $\boldsymbol{e}$.
3. S_1 sample a random degree-d sharing $[\![\boldsymbol{e}]\!]_d$ and distribute it to others.
4. Each server computes $[\![\boldsymbol{z}]\!]_d := [\![\boldsymbol{e}]\!]_d - [\![\boldsymbol{r}]\!]_d$.

Figure 14: The Π_{PSSMult} Protocol

Functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$

The functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{VerifyMult}, m, \{[\![\boldsymbol{x}^{(i)}]\!], [\![\boldsymbol{y}^{(i)}]\!], [\![\boldsymbol{z}^{(i)}]\!]\}_{i\in[m]})$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where m is the number of PSS tuples that all servers want to verify, it does:

1. For $i \in [m]$:
   - Reconstruct $\boldsymbol{x}^{(i)}$, $\boldsymbol{y}^{(i)}$ and $\boldsymbol{z}^{(i)}$ using the shares from the honest servers.
   - For $\boldsymbol{a} \in \{\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}\}$: compute the new PSS $[\![\boldsymbol{a}^{(i)}]\!]_{\mathcal{H}}$ of the secret $\boldsymbol{a}^{(i)}$, such that for each honest server, its share of $[\![\boldsymbol{a}^{(i)}]\!]$ is equal to its share of $[\![\boldsymbol{a}^{(i)}]\!]_{\mathcal{H}}$. Send the corrupted servers' shares of $[\![\boldsymbol{a}^{(i)}]\!]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{a}^{(i)}} := [\![\boldsymbol{a}^{(i)}]\!] - [\![\boldsymbol{a}^{(i)}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{a}^{(i)}}$ describes the inconsistency of $[\![\boldsymbol{a}^{(i)}]\!]$.
   - Compute $\boldsymbol{d}^{(i)} := \boldsymbol{z}^{(i)} - \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$ and send $\boldsymbol{d}^{(i)}$ to the adversary $\mathcal{S}$.
2. If there exists $i \in [m]$ such that $\boldsymbol{d}^{(i)} \neq \boldsymbol{0}^N$, set $b := 0$; otherwise, set $b := 1$.
3. Send b to the adversary $\mathcal{S}$. For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
   - If it is $(\textsc{Continue}, \mathsf{h})$, send b to h.
   - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 15: The functionality $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$

7.4 Put everything together

We have already shown how to achieve malicious security for PSS permutation, dMSM, and PSS multiplication. Replacing the semi-honest secure components with these malicious secure ones, we can obtain a malicious secure collaborative zk-SNARK. Formally, we have the following theorem.

Theorem 10. *Let $\mathcal{C}$ be a data-parallel circuit. Let* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a zk-SNARK described in [XZZ+19] for $\mathcal{C}$. There exists a collaborative zk-SNARK* $(\mathsf{Setup}, \Pi_{\mathsf{Mal}}, \mathsf{Verify})$ *for $\mathcal{C}$, where Π_{Mal} is a MPC protocol that computes* Prove *in* $\{\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}, \mathcal{F}_{\mathsf{dMSM\text{-}Mal}}, \mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}, \mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}, \mathcal{F}_{\mathsf{Verify\text{-}dMSM}}, \mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}, \mathsf{H}\}$*-hybrid world, against a malicious adversary who corrupts at most t servers. Here* H *denotes a random oracle.*

Proof. **Protocol description.** We first provide the protocol description of the underlying protocol Π_{Mal} in the following.

1. The servers invoke the semi-honest protocol Π described in Appendix C.2 with the following modifications: (i) when the servers need to invoke $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$, $\mathcal{F}_{\mathsf{dMSM\text{-}Semi}}$, $\mathcal{F}_{\mathsf{PSSMult\text{-}Semi}}$, they invoke $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}, \mathcal{F}_{\mathsf{dMSM\text{-}Mal}}, \mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$ instead; (ii) the servers do not output the final result for the time being.
2. The servers try to detect the malicious behaviors through the following checks:
   (a) For a permutation function p, let m_p denote the number of tuples computed by $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$. The tuples are denoted by
   $$([\![\boldsymbol{x}^{(1)}]\!], [\![\hat{\boldsymbol{x}}^{(1)}]\!]), ([\![\boldsymbol{x}^{(2)}]\!], [\![\hat{\boldsymbol{x}}^{(2)}]\!]), \ldots, ([\![\boldsymbol{x}^{(m_p)}]\!], [\![\hat{\boldsymbol{x}}^{(m_p)}]\!]).$$
   The servers send $(\textsc{VerPermute}, m_p, \{[\![\boldsymbol{x}^{(i)}, [\![\hat{\boldsymbol{x}}^{(i)}]\!]\!]\}_{i\in[m_1]}, p(\cdot))$ to $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ to check the correctness of these tuples.
   (b) Let m_1, m_2 denote the numbers that are used to describe the tuples computed by $\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$. More precisely, the tuples are denoted by
   $$\begin{aligned}&(\{[\![\boldsymbol{A}_i^{(1)}]\!], [\![\boldsymbol{b}_i^{(1)}]\!]\}_{i\in[m_1]}, \langle \mathsf{out}^{(1)}\rangle),\\ &(\{[\![\boldsymbol{A}_i^{(2)}]\!], [\![\boldsymbol{b}_i^{(2)}]\!]\}_{i\in[m_1]}, \langle \mathsf{out}^{(2)}\rangle),\\ &\vdots\\ &(\{[\![\boldsymbol{A}_i^{(m_2)}]\!], [\![\boldsymbol{b}_i^{(m_2)}]\!]\}_{i\in[m_1]}, \langle \mathsf{out}^{(m_2)}\rangle).\end{aligned}$$
   The servers send $(\textsc{VerMSM}, m_1, m_2, \{\{[\![\boldsymbol{A}_i^{(j)}]\!], [\![\boldsymbol{b}_i^{(j)}]\!]\}_{i\in[m_1]}, \langle \mathsf{out}^{(j)}\rangle\}_{j\in[m_2]})$ to $\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$ to check the correctness of these tuples.
   (c) Let m_3 denote the number of tuples computed by $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$. The tuples are denoted by
   $$\begin{aligned}&([\![\boldsymbol{x}^{(1)}]\!], [\![\boldsymbol{y}^{(1)}]\!], [\![\boldsymbol{z}^{(1)}]\!]), ([\![\boldsymbol{x}^{(2)}]\!], [\![\boldsymbol{y}^{(2)}]\!], [\![\boldsymbol{z}^{(2)}]\!]), \ldots,\\ &([\![\boldsymbol{x}^{(m_3)}]\!], [\![\boldsymbol{y}^{(m_3)}]\!], [\![\boldsymbol{z}^{(m_3)}]\!]).\end{aligned}$$
   The servers send $(\textsc{VerifyMult}, m_3, \{[\![\boldsymbol{x}^{(i)}]\!], [\![\boldsymbol{y}^{(i)}]\!], [\![\boldsymbol{z}^{(i)}]\!]\}_{i\in[m_3]})$ to $\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$ to check the correctness of these tuples.
3. If all the above checks pass, the servers output the final result of Π; otherwise, the servers simply abort.

Security. Here we provide a stretch of the security analysis. The security of our semi-honest protocol Π is proven through Theorem 3. However, a malicious adversary may instruct the corrupted servers to deviate from the protocol, resulting in incorrect outputs received by the honest servers, which is captured by $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}$,$\mathcal{F}_{\mathsf{dMSM\text{-}Mal}}$, and $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$. By letting the servers invoke $\mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}$,$\mathcal{F}_{\mathsf{Verify\text{-}dMSM}}$,$\mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$, the malicious behaviors caused by the adversary will be detected. Since our protocol Π_{Mal} only makes oracle to the random oracle H, the ideal functionalities $\mathcal{F}_{\mathsf{PSSPermute\text{-}Mal}}, \mathcal{F}_{\mathsf{dMSM\text{-}Mal}}, \mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}, \mathcal{F}_{\mathsf{Verify\text{-}PSSPermute}}, \mathcal{F}_{\mathsf{Verify\text{-}dMSM}}, \mathcal{F}_{\mathsf{Verify\text{-}PSSMult}}$, besides each server performing local operations on the shares of inputs. The security of this protocol follows the security of the functionalities invoked straightforwardly. □

Efficiency. Given a data-parallel d-depth arithmetic circuit $\mathcal{C}$ with B sub-copies and a total of n gates in the input layer, the total work of each server can be divided into two parts: proving and verification. The proving cost is the same as the semi-honest case, namely $O(\frac{|\mathcal{C}|}{N})$ computation and space. The verification involves $O(\frac{|\mathcal{C}|}{N}\log\frac{|\mathcal{C}|}{N}) = \tilde{O}(\frac{|\mathcal{C}|}{N})$ computation and $O(\frac{|\mathcal{C}|}{N})$ space. Therefore, the total cost of each server is $\tilde{O}(\frac{|\mathcal{C}|}{N})$ computation and $O(\frac{|\mathcal{C}|}{N})$ space.

8 Implementation and Evaluations

We provide implementation and performance evaluation of our semi-honest protocols.

Implementation details. We first implemented two primary components, $\Pi_{\mathsf{dSumcheck}}$ and Π_{dMKZG}. This was followed by developing a proof-of-concept implementation for collaborative proof generation, specifically designed for data-parallel circuits. We also created a simplified implementation of Libra for comparison purposes. We utilized the mpc-net crate [Ozd22] for network communication and the arkworks library [ac22] for finite field and elliptic curve operations. Overall, our project involved more than 2800 lines of Rust code. We provide our codebase at https://github.com/LBruyne/Collaborative-GKR.

Experiment setup. To evaluate the performance of our protocols, we conducted a comparative analysis with zkSaaS [GGJ+23] and a local prover of Libra [XZZ+19]. We chose the distributed Plonk implementation (denoted as Plonk-zkSaaS) in [GGJ+23] as a benchmark baseline since it focuses on arithmetic circuits, similar to our approach. The benchmark was performed on their open-source implementation [Pol23]. Recall that Plonk-zkSaaS operates under a *partially* distributed setting with one powerful server and several regular ones, while our work is fully distributed with equal server capabilities. Therefore, we consider the following two settings:

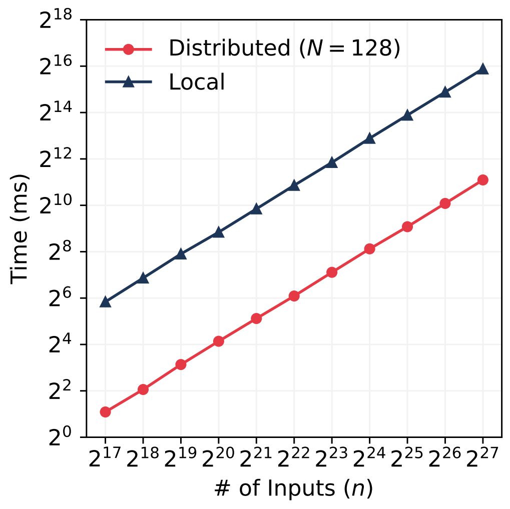
- Partially distributed setting (for Plonk-zkSaaS): This includes a powerful server (g7.6xlarge instance with 24 vCPUs, 96 GB RAM), and the other normal servers (c7.large instances with 2 vCPUs, 4 GB RAM).
- Fully distributed setting (for our proposal): All servers are uniform c7.large instances with 2 vCPUs and 4 GB of RAM.

We also assessed the local prover of Libra and other protocols on a single 2 vCPUs, 4 GB RAM instance. Our evaluation excluded the preprocessing stage, aligning with previous benchmarks [OB22, GGJ+23].

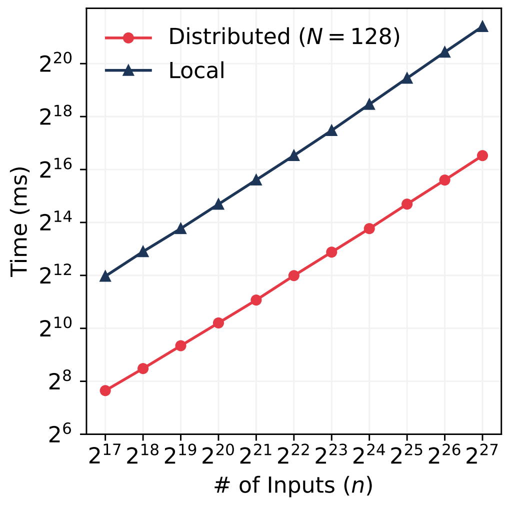
8.1 Performance of distributed primitives

We commence by evaluating the performance of two key protocols in proof generation, the sumcheck protocol and the PC scheme. For both protocols, we varied the number of inputs and compared the execution time between the fully distributed setup (with 128 servers) and a local prover. The servers in the distributed setup were linked through a high-speed 4Gbps network.

The results of these experiments are summarized in Figure 16. For both the sumcheck protocol and the PC scheme, we note that the performance in a fully distributed setting is approximately $30\times$ superior to that of a local prover, particularly when the number of inputs is large. This finding aligns with our theoretical analysis and demonstrates the scalability of our construction.



(a) Performance for $\Pi_{\mathsf{dSumcheck}}$.



(b) Performance for Π_{dMKZG}.

Figure 16: Performance of two distributed primitives, comparing proving time with a local prover. In the distributed setting, 128 servers are linked with a 4Gbps network.

8.2 Performance of collaborative proof generation

Next, we concentrated on assessing the performance of our collaborative proof generation process. We compared our fully distributed protocol against two different setups: (i) Plonk-zkSaaS, and (ii) a local Libra prover. Given that our protocol is tailored for data-parallel circuits, our experiments were based on a circuit comprising 64 copies, each with a depth of 16. We denoted the total number of gates in the circuit as $|\mathcal{C}|$. To evaluate performance across various circuit sizes, we altered the number of gates in each layer.

The evidence of fully distributed proof generation. In our experiments, all servers in the fully distributed setting incur the same running time and communication costs. A key observation is the even distribution of memory requirements among servers, enhancing our protocol's ability to manage larger circuits compared to local provers or partially distributed settings.

Comparison with a local Libra prover. We first compare with a local Libra prover. We put a detailed breakdown of running times for each related component in Figure 17. The results show that the performance of our protocol is $4\times$ to $19\times$ better than a local prover, varying with the number of gates. The reduced memory requirement per server in a fully distributed setting allows the handling of larger circuits than possible by a local prover. With 128 servers, our performance does not match the theoretical $32\times$ improvement, due to the PSS multiplications in the distributed sumcheck protocol for GKR relations, it adds computational and communication overheads which is not required in a local prover case.

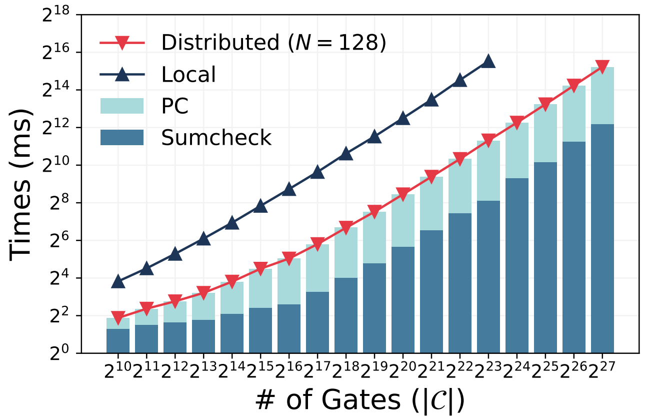
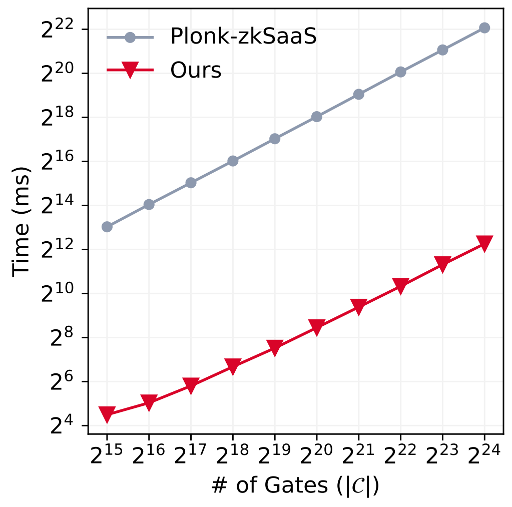


Figure 17: Comparison of proving time between a local Libra prover and a fully distributed cluster of 128 servers, interconnected with a network bandwidth of 4Gbps. The bar graph indicates the breakdown for each component involved in the process. Missing data points of the local prover setting are due to the memory limitation of the machine.

Comparison with Plonk-zkSaaS. We also conduct a comparison between our protocol and Plonk-zkSaaS from [GGJ+23]. This excludes multi-threading optimizations for both schemes. As depicted in Figure 18, our protocol shows significantly lower computation and communication costs per server. This efficiency is attributed to two factors: (i) the underlying zk-SNARK Libra is more efficient than Plonk, and (ii) our protocol avoids the use of the computationally demanding dFFT primitive. According to [GGJ+23], the dFFT primitive places a heavy computational load on the powerful server and requires extensive communication with other servers. Conversely, our protocol distributes overheads evenly among all servers. Moreover, as Table 1 indicates, our protocol is less memory-intensive. While Plonk-zkSaaS maxes out its memory capacity on relatively powerful machines for circuits larger than 2^{24} gates, our protocol which is executed by consumer machines can handle circuits up to 2^{27} gates, demonstrating greater scalability.

Varying server count and bandwidth. We then maintain a constant circuit size while varying the number of servers and network bandwidth. The results are summarized in Figure 19. We observe a linear improvement in our protocol's efficiency with an increasing number of servers. A reduction in bandwidth leads to longer communication times, resulting in a smaller efficiency gain. Despite this, the loss in efficiency due to communication overhead can be mitigated by adding more servers. As shown in Figure 19, our protocol still achieves considerable efficiency gains (about 4 times) with 256 servers, even under a limited bandwidth (64Mbps) network.

Financial cost comparison. Finally, we compare the financial costs of our system with [GGJ+23], referencing



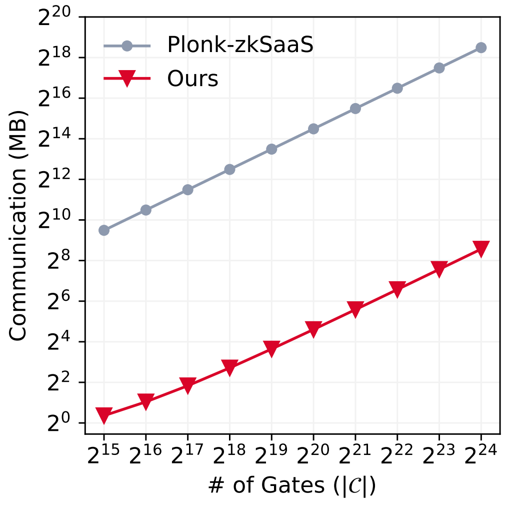


Figure 18: Comparison of proving time and communication cost between Plonk-zkSaaS and this work, both in a cluster where 128 servers are linked with a network bandwidth of 4Gbps. The performance of Plonk-zkSaaS is assessed by measuring the proving time and communication cost specifically for the designated powerful server. The evaluation does not include multi-threading optimizations.

$\lvert\mathcal{C}\rvert$	Ours		Plonk-zkSaaS	
	Time	Comm.	Time	Comm.
2^{23}	2.5 s	190 MB	2193 s	184 GB
2^{24}	4.9 s	379 MB	4399 s	368 GB
2^{25}	9.6 s	757 MB	—	—
2^{26}	19.2 s	1512 MB	—	—

Table 1: Extra data for circuit sizes from 2^{23} to 2^{26}. Here Comm. denotes the communication cost, and — indicates the corresponding circuit is not available due to the memory limitation. The data of Plonk-zkSaaS is obtained from the powerful server. When $|\mathcal{C}| = 2^{25}$, the powerful server of Plonk-zkSaaS consumes 96 GB memory before crash, whereas in our protocol each server utilizes only 2.15 GB. Neither of the performances considers multi-threading optimizations.

Google Cloud's pricing for spot instances and network services[3] [GCP23]. With 128 low-level instances and a 4Gbps network bandwidth, our protocol generates a proof for a 2^{19} gate circuit in 0.2s, at a cost of around 6.2 cents, mainly due to network charges. In comparison, the estimated cost of [GGJ+23] is 89.8 cents. Thus, our cost is just 7% of theirs. This substantial saving is due to two factors: (i) our protocols only require lower-spec machines, avoiding the need for a high-capacity server; (ii) the reduced communication cost leads to lower network fees.

Acknowledgement

We thank Alex Ozdemir, Jiaheng Zhang for their helpful discussion and their valuable comments on the draft of this work. We thank Guru-Vamsi Policharla for his helpful advice regarding this work.

References

[ac22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David

[3]Pricing at \$0.00632 per vCPU hour and \$0.000847 per GB hour for custom N1 machines, and network costs at \$0.08 per GB.

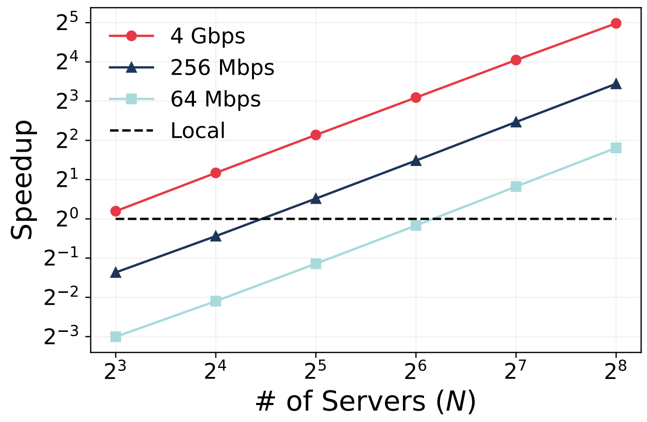


Figure 19: The performance of this work, for a circuit with 2^{23} gates, varying server number and network bandwidth.

Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14*, volume 8642 of *LNCS*, pages 175–196. Springer, Heidelberg, September 2014.

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 663–693. Springer, Heidelberg, October 2021.

[But21] Vitalik Buterin. An incomplete guide to rollups. https://vitalik.eth.limo/general/2021/01/05/rollup.html, Jan 2021.

[But22] Vitalik Buterin. The different types of zk-evms. https://vitalik.eth.limo/general/2022/08/04/zkevm.html, Aug 2022.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[CFS17] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. A zero knowledge sumcheck and its applications. Cryptology ePrint Archive, Report 2017/305, 2017. https://eprint.iacr.org/2017/305.

[CHM$^+$19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Report 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[CLMZ23] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zkSNARK provers. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6453–6469, Anaheim, CA, August 2023. USENIX Association.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *ITCS 2012*, pages 90–112. ACM, January 2012.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg, May / June 2010.

[DPP$^+$22] Pankaj Dayama, Arpita Patra, Protik Paul, Nitin Singh, and Dhinakaran Vinayagamurthy. How to prove any NP statement jointly? Efficient distributed-prover zero-knowledge protocols. *PoPETs*, 2022(2):517–556, April 2022.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[EGP$^+$23] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. SuperPack: Dishonest majority MPC with constant online communication. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2023.

[EGPS22] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. TurboPack: Honest majority MPC with constant online communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 951–964. ACM Press, November 2022.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.

[GCP23] Vm instance pricing. https://cloud.google.com/compute/vm-instance-pricing, 2023.

[GGJ$^+$23] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: Zero-Knowledge SNARKs as a service. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4427–4444, Anaheim, CA, August 2023. USENIX Association.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[GPS21a] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event, August 2021. Springer, Heidelberg.

[GPS21b] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. Cryptology ePrint Archive, Report 2021/834, 2021. https://eprint.iacr.org/2021/834.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2022.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 618–646. Springer, Heidelberg, August 2020.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*, pages 2–10. IEEE Computer Society Press, October 1990.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985. ACM Press, November 2021.

[LXZ$^+$23] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. Cryptology ePrint Archive, Paper 2023/1271, 2023. https://eprint.iacr.org/2023/1271.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[OB22] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-knowledge proofs for distributed secrets. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4291–4308. USENIX Association, August 2022.

[Ozd22] Alex Ozdemir. collaborative-zkSNARK implementation, 2022.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.

[Pol23] Guru-Vamsi Policharla. zkSaaS implementation, 2023.

[PST13] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 222–242. Springer, Heidelberg, March 2013.

[Sch80] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SVdV16] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 346–366. Springer, Heidelberg, June 2016.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, Heidelberg, August 2013.

[WJB$^+$17] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2071–2086. ACM Press, October / November 2017.

[WTs$^+$18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.

[WYX$^+$21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[WZC$^+$18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 675–692. USENIX Association, August 2018.

[XZC$^+$22] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkBridge: Trustless cross-chain bridges made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3003–3017. ACM Press, November 2022.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 299–328. Springer, Heidelberg, August 2022.

[XZZ$^+$19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019.

[ZCL$^+$21] Zhelei Zhou, Xinle Cao, Jian Liu, Bingsheng Zhang, and Kui Ren. Zero knowledge contingent payments for trained neural networks. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part II*, volume 12973 of *LNCS*, pages 628–648. Springer, Heidelberg, October 2021.

[ZFZS20] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2039–2053. ACM Press, November 2020.

[ZGK$^+$17a] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy*, pages 863–880. IEEE Computer Society Press, May 2017.

[ZGK$^+$17b] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. A zero-knowledge version of vSQL. Cryptology ePrint Archive, Report 2017/1146, 2017. https://eprint.iacr.org/2017/1146.

[ZGK$^+$18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925. IEEE Computer Society Press, May 2018.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.

[ZLW$^+$21] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. Doubly efficient interactive proofs for general arithmetic circuits with linear prover time. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 159–177. ACM Press, November 2021.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy*, pages 859–876. IEEE Computer Society Press, May 2020.

A Additional Preliminaries

A.1 Zero-knowledge arguments

An interactive argument system for an NP relationship $\mathcal{R}$ is a protocol between a prover P and a verifier V. The protocol runs in several rounds, allowing V to ask questions in each round based on P's previous answers. At the end of the protocol, V is convinced by P that there exists a *witness* w such that $\mathcal{R}(x, w) = 1$ for some public input x. Further, we focus on the *zero-knowledge argument of knowledge* that P convinces V that he knows such a witness w that satisfies the relationship without leaking information about w. Formally, we give the definition of zero-knowledge interactive argument of knowledge as follows:

Definition 4. *A tuple of three algorithms* (G, P, V) *is a zero-knowledge interactive argument of knowledge for* $\mathcal{R}$ *if it satisfies the following properties:*

- ***Completeness***. *For every* pp *output by* $\mathsf{G}(1^\lambda)$, *a statement-witness pair* (x, w) *such that* $\mathcal{R}(x, w) = 1$, *the following relation holds:*
$$\Pr\left[\langle \mathsf{P}(w), \mathsf{V}\rangle(x, \mathsf{pp}) = 1\right] = 1$$

- ***Knowledge soundness***. *For any PPT prover* P^*, *there exists a PPT extractor* $\mathcal{E}$ *such that for every* pp *output by* $\mathsf{G}(1^\lambda)$, *any input* x, *and the extractor's output* $w^* \leftarrow \mathcal{E}^{\mathsf{P}^*}(\mathsf{pp}, x)$, *the following relation is* $\mathsf{negl}(\lambda)$*:*
$$\Pr\left[\langle \mathsf{P}^*, \mathsf{V}\rangle(x, \mathsf{pp}) = 1 \wedge \mathcal{R}(x, w^*) = 1\right]$$

- ***Zero-knowledge***. *There exists a PPT simulator* $\mathcal{S}$ *that for any PPT algorithm* V^*, $\mathcal{R}(x, w) = 1$, pp *output by* $\mathsf{G}(1^\lambda)$, *it holds that:*
$$\mathsf{View}^{\mathsf{V}^*}(\langle \mathsf{P}(w), \mathsf{V}^*\rangle(x, \mathsf{pp})) \approx \mathcal{S}^{\mathsf{V}^*}(x)$$
where $\mathsf{View}^{\mathsf{V}^*}(\langle \mathsf{P}(w), \mathsf{V}^*\rangle(x, \mathsf{pp}))$ *is the view of* V^* *in the real protocol, and* $\mathcal{S}^{\mathsf{V}^*}(x)$ *is the view generated by* $\mathcal{S}$ *given* x *and the transcript of* V^*. $\approx$ *denotes the two distributions are computationally indistinguishable.*

A public-coin interactive argument system can be made non-interactive under the random-oracle model using the Fiat-Shamir heuristic transformation [FS87]. We say the argument system is *succinct* if the running time of the verifier and the total communication cost (proof size) are both of $\mathsf{poly}(\lambda, |x|, \log|w|)$. A *zero-knowledge Succinct Non-interactive ARgument of Knowledge* is called a *zk-SNARK*. Formally, A zk-SNARK constitutes a tuple of algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$:

- $\mathsf{Setup}(1^\lambda, \mathcal{R}) \to \mathsf{pp}$: It takes the security parameter λ and the NP relation $\mathcal{R}$ as inputs, and outputs the public parameter pp.
- $\mathsf{Prove}(\mathsf{pp}, x, w) \to \pi$: It takes the public parameter pp, the public input x, and the witness w as inputs, and outputs a proof π.
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0, 1\}$: It takes the public parameter pp, the statement x, and the proof π as inputs, and outputs a bit b indicating acceptance $(b = 1)$ or rejection $(b = 0)$.

A zk-SNARK has the following security properties: completeness, knowledge soundness, zero-knowledge property and succinctness. We refer interested readers to see formal definitions of these properties in [GGJ+23].

A.2 Secure multi-party computation

Let $\mathcal{C} : (\{0,1\}^\lambda)^N \to (\{0,1\}^\lambda)^N$ be a circuit and let $\mathsf{P}_1, \dots, \mathsf{P}_N$ be the parties that will participate in a secure Multi-Party Computation (MPC) protocol Π for $\mathcal{C}$. During the execution of Π, we assume that each party P_i has a private input $x_i \in \{0,1\}^\lambda$, and P_i wants to receive $y_i \in \{0,1\}^\lambda$ as output, where $(y_1, \dots, y_N) := \mathcal{C}(x_1, \dots, x_N)$, without revealing its private input.

We analyze the security of the MPC protocol Π in the real-world/ideal-world paradigm [Can00]. Here we provide an informal and high-level description for this paradigm, and we refer readers to see more details in [Can00]. In real-world execution, the real parties $\mathsf{P}_1, \dots, \mathsf{P}_N$ communicate with each other to execute Π, and there is an adversary $\mathcal{A}$ who can choose a set of parties before the beginning of the execution, and we call it static corruption. The set of the corrupted parties is denoted by $\mathcal{C}orr$. In ideal-world execution, there are dummy parties $\tilde{\mathsf{P}}_1, \dots, \tilde{\mathsf{P}}_N$, an ideal-world adversary (a.k.a, the simulator) $\mathcal{S}$ who can corrupt the same set $\mathcal{C}orr$, and a trusted entity called ideal functionality $\mathcal{F}$. The ideal functionality $\mathcal{F}$ receives inputs from the dummy parties and the simulator, then computes $\mathcal{C}$, and delivers the corresponding output to the parties. We say the protocol Π securely realizes $\mathcal{F}$, if the outputs of parties in real-world execution is computationally indistinguishable from those in ideal-world execution. Notice that, in this work, we also use the term "hybrid world". More concretely, when we say a protocol is in the $\mathcal{G}$-hybrid world, it means that the parties can have an oracle access to an ideal functionality $\mathcal{G}$. In this work, we consider *security with abort* as in [OB22], i.e., the adversary can cause some honest parties to abort during the protocol execution.

Recall that, in this work, we aim to design an efficient MPC protocol for the prover algorithm Prove of a zk-SNARK scheme $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$. For the ease of presentation, when we say that Π is an MPC protocol Π that computes Prove, we mean that the prover algorithm Prove can be represented as a circuit, and the protocol Π securely realizes an ideal functionality who computes this circuit. Similar treatments can be found in prior works [OB22, GGJ+23].

B Helper Functionalities

In this section, we put some helper functionalities that are used in the main body of this paper.

B.1 Functionality for coin-flipping

Here we introduce the functionality for coin-flipping. It allows all parties to receive the same uniformly random string. Throughout the paper, we only consider the security with abort; therefore, here we let the functionality capture the security with abort. Formally, we present the functionality for coin-flipping in Figure 20.

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

The functionality $\mathcal{F}_{\mathsf{Coin}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{Flip}, \ell)$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do:

- Sample $\boldsymbol{s} \leftarrow \mathbb{F}^{\ell}$ and send $(\textsc{Flip}, \boldsymbol{s})$ to all corrupted parties.
- Send $\textsc{Continue}$ to the adversary $\mathcal{S}$. For each honest party $\mathsf{h} \in \mathcal{H}$, upon receiving an input from $\mathcal{S}$,
  - If it is $(\textsc{Continue}, \mathsf{h})$, send $(\textsc{Flip}, \boldsymbol{s})$ to h.
  - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 20: The functionality $\mathcal{F}_{\mathsf{Coin}}$.

B.2 Functionalities for preprocessing

Here we introduce some ideal functionalities that will be used in the preprocessing phase, and these functionalities aim to provide some correlated randomness that will be consumed in the online phase protocol. We note that there are two approaches to realize these functionalities: (i) let the servers invoke a corresponding MPC protocol; and (ii) let a trusted third party deliver these correlated randomness. In this work, we employ the second approach; therefore, we omit the protocol descriptions that are used to realize these functionalities. We note that in [GPS21a, GGJ+23], the authors showed how to realize some of these functionalities; we refer interested readers to see them in [GPS21a, GGJ+23].

Generating shares of random values. Here we provide the ideal functionality for generating shares of a random value, which is denoted as $\mathcal{F}_{\mathsf{Rand}}$. Formally, we put the detailed description of $\mathcal{F}_{\mathsf{Rand}}$ in Figure 21.

Generating double packed shares of random vectors. Here we provide the ideal functionality for generating double-packed shares of a batch of random vectors, which is denoted as $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$. Formally, we put the detailed description of $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ in Figure 22.

Generating random shares for PSS permutation. Here we provide the ideal functionalities for generating random shares for PSS permutation for both semi-honest security (which is denoted as $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$) and malicious security (which is denoted as $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$). Formally, we put the detailed descriptions of $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$ and $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Mal}}$ in Figure 23 and Figure 24 respectively.

Converting packed shares to regular shares. Here we provide the ideal functionality for converting packed shares to regular shares for both semi-honest security and malicious security, which are denoted as $\mathcal{F}_{\mathsf{PSSToss\text{-}Semi}}$ and $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$, respectively. Formally, we put the detailed description of $\mathcal{F}_{\mathsf{PSSToss\text{-}Semi}}$ and $\mathcal{F}_{\mathsf{PSSToss\text{-}Mal}}$ in Figure 25 and Figure 26, respectively.

Generating random shares for dMSM. Here we provide the ideal functionality for generating random shares for dMSM, which is denoted as $\mathcal{F}_{\mathsf{Rand\text{-}dMSM}}$. Formally, we put the detailed description of $\mathcal{F}_{\mathsf{Rand\text{-}dMSM}}$ in Figure 27.

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

The functionality $\mathcal{F}_{\mathsf{Rand}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving RAND from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do:

- Receive a set of shares $\{s_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- Sample a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and sample a random degree-d packed secret sharing $[\![\boldsymbol{r}]\!]$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $[\![\boldsymbol{r}]\!]$ is s_i.
- Send CONTINUE to the adversary $\mathcal{S}$. For each honest party $\mathsf{h} \in \mathcal{H}$, upon receiving an input from $\mathcal{S}$,
  - If it is $(\text{CONTINUE}, \mathsf{h})$, send its corresponding share of $[\![\boldsymbol{r}]\!]$ to h.
  - If it is $(\text{ABORT}, \mathsf{h})$, send ABORT to h.

Figure 21: The functionality $\mathcal{F}_{\mathsf{Rand}}$.

Functionality $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$

The functionality $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$.

Upon receiving DOUBLERAND from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do::

- Receive shares $\{u_i, v_i\}_{i \in \mathcal{C}orr}$ from $\mathcal{S}$.
- Choose a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and sample random degree-d and $2d$ packed secret sharing $[\![\boldsymbol{r}]\!]_d$ and $[\![\boldsymbol{r}]\!]_{2d}$ such that the shares of the corrupted parties are identical to those received from the $\mathcal{S}$, i.e., $\{u_i, v_i\}_{i \in \mathcal{C}orr}$.
- Send the shares $[\![\boldsymbol{r}]\!]_d$ and $[\![\boldsymbol{r}]\!]_{2d}$ to all parties.

Figure 22: The functionality $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$.

Functionality $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$

The functionality $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\text{RANDPERMUTE}, p(\cdot))$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where $p(\cdot)$ is a permutation function, do:

- Receive a set of shares $\{u_i, v_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- Sample a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and compute $\hat{\boldsymbol{r}} := p(\boldsymbol{x})$.
- Sample two random degree-t packed secret sharing $[\![\boldsymbol{r}]\!]$ and $[\![\hat{\boldsymbol{r}}]\!]$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $[\![\boldsymbol{r}]\!]$ (resp. $[\![\hat{\boldsymbol{r}}]\!]$) is u_i (resp. v_i).
- Distribute $[\![\boldsymbol{r}]\!]$ and $[\![\hat{\boldsymbol{r}}]\!]$ to all servers.

Figure 23: The functionality $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$.

Functionality $\mathcal{F}_{\textsf{Rand-PSSPermute-Mal}}$

The functionality $\mathcal{F}_{\textsf{Rand-PSSPermute-Mal}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{RandPermute}, p(\cdot))$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where $p(\cdot)$ is a permutation function, do:

- Receive a set of shares $\{u_i, v_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- Receive a vector $\boldsymbol{d} \in \mathbb{F}^k$ from the adversary $\mathcal{S}$.
- Sample a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and compute $\hat{\boldsymbol{r}} := p(\boldsymbol{r}) + \boldsymbol{d}$.
- Sample two random degree-t packed secret sharing $[\![\boldsymbol{r}]\!]_{\mathcal{H}}$ and $[\![\hat{\boldsymbol{r}}]\!]_{\mathcal{H}}$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $[\![\boldsymbol{r}]\!]_{\mathcal{H}}$ (resp. $[\![\hat{\boldsymbol{r}}]\!]_{\mathcal{H}}$) is u_i (resp. v_i).
- Receive two vectors $\boldsymbol{\Delta}_{\boldsymbol{r}}, \boldsymbol{\Delta}_{\hat{\boldsymbol{r}}} \in \mathbb{F}^N$ from the adversary $\mathcal{S}$ and compute $[\![\boldsymbol{r}]\!] := [\![\boldsymbol{r}]\!]_{\mathcal{H}} + \boldsymbol{\Delta}_{\boldsymbol{r}}$ and $[\![\hat{\boldsymbol{r}}]\!] := [\![\hat{\boldsymbol{r}}]\!]_{\mathcal{H}} + \boldsymbol{\Delta}_{\hat{\boldsymbol{r}}}$.
- Send $\textsc{Continue}$ to the adversary $\mathcal{S}$. For each honest party $\mathsf{h} \in \mathcal{H}$, upon receiving an input from $\mathcal{S}$,
  - If it is $(\textsc{Continue}, \mathsf{h})$, send its corresponding shares of $[\![\boldsymbol{r}]\!]$ and $[\![\hat{\boldsymbol{r}}]\!]$ to h.
  - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 24: The functionality $\mathcal{F}_{\textsf{Rand-PSSPermute-Mal}}$.

Functionality $\mathcal{F}_{\textsf{PSSToss-Semi}}$

The functionality $\mathcal{F}_{\textsf{PSSToss-Semi}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$.

Upon receiving $(\textsc{Toss}, [\![\boldsymbol{x}]\!])$ from all servers, do:

- For each $j \in [k]$, receive from the adversary a set of shares $\{u_{j,i}\}_{i \in \mathcal{C}orr}$ from $\mathcal{S}$.
- Reconstruct $\boldsymbol{x} = (x_1, ..., x_k)$ from $[\![\boldsymbol{x}]\!]$.
- For each $j \in [k]$, computes a random sharing of x_j such that the shares of the corrupted servers are identical to those received from $\mathcal{S}$, i.e., $\{u_{j,i}\}_{i \in \mathcal{C}orr}$.
- For each $j \in [k]$, distribute $\langle x_j \rangle$ to all servers.

Figure 25: The functionality $\mathcal{F}_{\textsf{PSSToss-Semi}}$.

Functionality $\mathcal{F}_{\textsf{PSSToss-Mal}}$

The functionality $\mathcal{F}_{\textsf{PSSToss-Mal}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$.

Upon receiving $(\textsc{Toss}, [\![\boldsymbol{x}]\!])$ from all servers, do:

- Reconstruct the correct $\boldsymbol{x} = (x_1, \ldots, x_k)$ using the shares from the honest servers.
- Compute the new PSS $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$ of the secrets $\boldsymbol{x}$, such that for each honest server, its share of $[\![\boldsymbol{x}]\!]$ is equal to its share of $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$. Send the corrupted servers' shares of $[\![\boldsymbol{x}]\!]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_{\boldsymbol{x}} := [\![\boldsymbol{x}]\!] - [\![\boldsymbol{x}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta}_{\boldsymbol{x}}$ describes the inconsistency of $[\![\boldsymbol{x}]\!]$.
- Receive a vector $\boldsymbol{d} \in \mathbb{F}^k$ from the adversary $\mathcal{S}$, and set $\boldsymbol{x} := \boldsymbol{x} + \boldsymbol{d}$.
- For each $j \in [k]$, receive a set of shares $\{u_{j,i}\}_{i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- For each $j \in [k]$, computes a random sharing $\langle x_j \rangle_{\mathcal{H}}$ such that the shares of the corrupted servers are identical to those received from $\mathcal{S}$, i.e., $\{u_{j,i}\}_{i, \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$.
- For each $j \in [k]$, receive $\boldsymbol{\Delta}_j$ from the adversary $\mathcal{S}$ and compute $\langle x_j \rangle := \langle x_j \rangle_{\mathcal{H}} + \boldsymbol{\Delta}_j$.
- For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
  - If it is $(\textsc{Continue}, \mathsf{h})$, send its corresponding share of $\{\langle x_j \rangle\}_{j \in [k]}$ to h.
  - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 26: The functionality $\mathcal{F}_{\textsf{PSSToss-Mal}}$.

Functionality $\mathcal{F}_{\text{Rand-dMSM}}$

The functionality $\mathcal{F}_{\text{Rand-dMSM}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving RANDMULT from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do:

- Receive a set of shares $\{u_i, v_i, s_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- Sample random vectors $\boldsymbol{A} \in \mathbb{G}^k, \boldsymbol{b} \in \mathbb{F}^k$ and sample random degree-d packed secret sharing $[\![\boldsymbol{A}]\!]$ and $[\![\boldsymbol{b}]\!]$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th shares of $[\![\boldsymbol{A}]\!]$ and $[\![\boldsymbol{b}]\!]$ are u_i and v_i respectively. Note that, we let $\boldsymbol{A} = (A_1, \ldots, A_k)$ and $\boldsymbol{b} = (b_1, \ldots, b_k)$.
- Compute $z := \prod_{i \in [k]} A_i \cdot g^{b_i}$. Sample a random degree-d Shamir's sharing $\langle z \rangle$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $\langle z \rangle$ is s_i.
- Send CONTINUE to the adversary $\mathcal{S}$. For each honest party $\mathsf{h} \in \mathcal{H}$, upon receiving an input from $\mathcal{S}$,
  - If it is (CONTINUE, h), send its corresponding shares of $[\![\boldsymbol{A}]\!], [\![\boldsymbol{b}]\!], \langle z \rangle$ to h.
  - If it is (ABORT, h), send ABORT to h.

Figure 27: The functionality $\mathcal{F}_{\text{Rand-dMSM}}$.

B.3 Functionality for packed secret sharing multiplication

Here we provide the ideal functionality for PSS multiplication (given two PSS $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$, the goal is to compute $[\![\boldsymbol{c}]\!]$ such that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$) for both semi-honest and malicious security, which are denoted as $\mathcal{F}_{\text{PSSMult-Semi}}$ and $\mathcal{F}_{\text{PSSMult-Mal}}$, respectively. Formally, we put the detailed description of $\mathcal{F}_{\text{PSSMult-Semi}}$ and $\mathcal{F}_{\text{PSSMult-Mal}}$ in Figure 28 and Figure 29, respectively.

Functionality $\mathcal{F}_{\text{PSSMult-Semi}}$

The functionality $\mathcal{F}_{\text{PSSMult-Semi}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving (PSSMULT, $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$) from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do:

- Receive a set of shares $\{u_i\}_{i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$.
- Reconstruct $\boldsymbol{a} = (a_1, ..., a_k)$ and $\boldsymbol{b} = (b_1, ..., b_k)$ from $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$.
- Compute $\boldsymbol{c} := \boldsymbol{a} * \boldsymbol{b}$, where $*$ means element wise multiplication, i.e., $c_i = a_i \cdot b_i$ for $i \in [k]$.
- Sample random sharing $[\![\boldsymbol{c}]\!]$ of $\boldsymbol{c}$, such that the shares of the corrupted servers are identical to those received from $\mathcal{S}$, i.e., $\{u_i\}_{i, \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$.
- Distribute $[\![\boldsymbol{c}]\!]$ to all servers.

Figure 28: The functionality $\mathcal{F}_{\text{PSSMult-Semi}}$.

In [GPS21a], Goyal et al. presented a protocol that can securely realize $\mathcal{F}_{\text{PSSMult-Semi}}$ against a semi-honest adversary. They also proved that the same protocol can securely realize $\mathcal{F}_{\text{PSSMult-Mal}}$ against a malicious adversary. For completeness, we present their protocol in Section 7.3. Notice that, for multiplying two packed shares, this procedure incurs a communication cost of $O(k)$ elements.

B.4 Functionality for packed secret sharing permutation

Formally, we present the ideal functionality for PSS permutation $\mathcal{F}_{\text{PSSPermute-Semi}}$ in Figure 30 and put the protocol $\Pi_{\text{PSSPermute-Semi}}$ that securely realizes it in Figure 31. Notice that, in the preprocessing phase of our protocol $\Pi_{\text{PSSPermute-Semi}}$, we let the servers invoke a functionality $\mathcal{F}_{\text{Rand-PSSPermute-Semi}}$ to generate a pair of random shares $[\![\boldsymbol{r}]\!]$ and $[\![\hat{\boldsymbol{r}}]\!]$, where $\hat{\boldsymbol{r}} = p(\boldsymbol{r})$ and $\boldsymbol{r}$; we put the detailed description of $\mathcal{F}_{\text{Rand-PSSPermute-Semi}}$ in Appendix B.2. The security of the protocol $\Pi_{\text{PSSPermute-Semi}}$ is proven through Theorem 11.

Theorem 11. *The protocol $\Pi_{\text{PSSPermute-Semi}}$ depicted in Figure 31 securely realizes $\mathcal{F}_{\text{PSSPermute-Semi}}$ depicted in Figure 30 in the*
$\mathcal{F}_{\text{Rand-PSSPermute-Semi}}$-hybrid world against a semi-honest adversary corrupting up to t servers.

Functionality $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$

The functionality $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$ interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. It is parameterized with a finite field $\mathbb{F}$. Let $\mathcal{C}orr$ be the set of the corrupted servers. Let $\mathcal{H}$ be the set of the honest servers.

Upon receiving $(\textsc{PssMult}, [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, do:

- Reconstruct the correct $\boldsymbol{a} = (a_1, \ldots, a_k)$ and $\boldsymbol{b} = (b_1, \ldots, b_k)$ using the shares from the honest servers.
- Compute the new PSS $[\![\boldsymbol{a}]\!]_{\mathcal{H}}, [\![\boldsymbol{b}]\!]_{\mathcal{H}}$ of the secrets $\boldsymbol{x}$, such that for each honest server, its share of $[\![\boldsymbol{a}]\!]$ (resp. $[\![\boldsymbol{b}]\!]$) is equal to its share of $[\![\boldsymbol{a}]\!]_{\mathcal{H}}$ (resp. $[\![\boldsymbol{b}]\!]_{\mathcal{H}}$). Send the corrupted servers' shares of $[\![\boldsymbol{a}]\!]_{\mathcal{H}}, [\![\boldsymbol{b}]\!]_{\mathcal{H}}$, $\boldsymbol{\Delta_a} := [\![\boldsymbol{a}]\!] - [\![\boldsymbol{a}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ and $\boldsymbol{\Delta_b} := [\![\boldsymbol{b}]\!] - [\![\boldsymbol{b}]\!]_{\mathcal{H}} \in \mathbb{F}^N$ to the adversary $\mathcal{S}$, where $\boldsymbol{\Delta_a}$ (resp. $\boldsymbol{\Delta_b}$) describes the inconsistency of $[\![\boldsymbol{a}]\!]$ (resp. $[\![\boldsymbol{b}]\!]$).
- Receive a vector $\boldsymbol{d} \in \mathbb{F}^k$ from the adversary $\mathcal{S}$. Compute $\boldsymbol{c} := \boldsymbol{a} * \boldsymbol{b} + \boldsymbol{d}$.
- Receive a set of shares $\{u_i\}_{i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$. Sample a random sharing $[\![\boldsymbol{c}]\!]_{\mathcal{H}}$ of $\boldsymbol{c}$ such that the shares of the corrupted servers are identical to those received from $\mathcal{S}$, i.e., $\{u_i\}_{i, \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$.
- Receive $\boldsymbol{\Delta} \in \mathbb{F}^N$ from the adversary $\mathcal{S}$ and compute $[\![\boldsymbol{c}]\!] := [\![\boldsymbol{c}]\!]_{\mathcal{H}} + \boldsymbol{\Delta}$.
- For each honest server $\mathsf{h} \in \mathcal{H}$, upon receiving an input from the adversary $\mathcal{S}$,
  - If it is $(\textsc{Continue}, \mathsf{h})$, send its corresponding share of $[\![\boldsymbol{c}]\!]$ to h.
  - If it is $(\textsc{Abort}, \mathsf{h})$, send $\textsc{Abort}$ to h.

Figure 29: The functionality $\mathcal{F}_{\mathsf{PSSMult\text{-}Mal}}$.

Proof. We refer interested readers to see the proof in [GPS21a]. □

Functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$

The functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of corrupted servers.

Upon receiving $(\textsc{Permute}, [\![\boldsymbol{x}]\!], p(\cdot))$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where $[\![\boldsymbol{x}]\!]$ is a degree-d packed secret sharing of $\boldsymbol{x}$ and $p(\cdot)$ is a permutation, it does:

1. Reconstruct the secret $\boldsymbol{x} \leftarrow \mathsf{Open}([\![\boldsymbol{x}]\!])$ and compute the permuted secret $\hat{\boldsymbol{x}} := p(\boldsymbol{x})$.
2. Receive a set of shares $\{s_i\}_{i \text{ s.t. } \mathsf{S}_i \in \mathcal{C}orr}$ from the adversary $\mathcal{S}$. Sample a random degree-d packed secret sharing $[\![\hat{\boldsymbol{x}}]\!]$ of $\hat{\boldsymbol{x}}$ such that for all $\mathsf{S}_i \in \mathcal{C}orr$, the i-th share of $[\![\hat{\boldsymbol{x}}]\!]$ is s_i.
3. Distribute the shares $[\![\hat{\boldsymbol{x}}]\!]$ to all servers.

Figure 30: The functionality $\mathcal{F}_{\mathsf{PSSPermute\text{-}Semi}}$

Protocol $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$

Let $[\![\boldsymbol{x}]\!]$ be the input degree-d PSS of a vector $\boldsymbol{x}$ that is to be permuted and let $p(\cdot)$ be the permutation function that is to be used.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\mathsf{Rand\text{-}PSSPermute\text{-}Semi}}$ to prepare a pair of random shares $[\![\boldsymbol{r}]\!]$ and $[\![\hat{\boldsymbol{r}}]\!]$, where $\hat{\boldsymbol{r}} = p(\boldsymbol{r})$ and $\boldsymbol{r}$ and $\hat{\boldsymbol{r}}$ are unknown to any server S_i.

Online phase:

1. Each server S_i computes $[\![\boldsymbol{m}]\!] := [\![\boldsymbol{x}]\!] + [\![\boldsymbol{r}]\!]$ and send $[\![\boldsymbol{m}]\!]$ to S_1.
2. S_1 reconstructs to get the vector $\boldsymbol{m}$ and performs the permutation on $\boldsymbol{m}$ to get $\hat{\boldsymbol{m}} := p(\boldsymbol{m})$. Then S_1 computes $[\![\hat{\boldsymbol{m}}]\!]$ and distributes the shares to other servers.
3. Each server S_i locally computes $[\![\hat{\boldsymbol{x}}]\!] := [\![\hat{\boldsymbol{m}}]\!] - [\![\hat{\boldsymbol{r}}]\!]$.

Figure 31: The $\Pi_{\mathsf{PSSPermute\text{-}Semi}}$ protocol

B.5 Functionality for distributed multi-scalar multiplication

Notice that, in the preprocessing phase of the protocol, we let the servers invoke a functionality $\mathcal{F}_{\textsf{Double-Rand}}$ to generate a pair of random shares $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$ and a functionality $\mathcal{F}_{\textsf{PSSToss-Semi}}$ to transform a packed secret sharing $[\![\boldsymbol{r}]\!]_d$ to a normal Shamir secret sharing $\langle r_1\rangle, ..., \langle r_k\rangle$. The detailed description of these two functionalities is provided in Appendix B.2. The security of the protocol $\Pi_{\textsf{dMSM-Semi}}$ is proven secure through Theorem 12.

Theorem 12. *The protocol* $\Pi_{\textsf{dMSM-Semi}}$ *depicted in Figure 33 securely realizes* $\mathcal{F}_{\textsf{dMSM-Semi}}$ *depicted in Figure 32 in the* $\{\mathcal{F}_{\textsf{Double-Rand}}, \mathcal{F}_{\textsf{PSSToss-Semi}}\}$*-hybrid world against a semi-honest adversary corrupting up to* t *servers.*

Proof. We refer interested readers to see the proof in [GGJ+23]. □

Functionality $\mathcal{F}_{\textsf{dMSM-Semi}}$

The functionality $\mathcal{F}_{\textsf{dMSM-Semi}}$ interacts with a set of servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$. Let $\mathcal{C}orr$ be the set of corrupted servers. It does:

Upon receiving $(\textsc{Mult}, m, [\![\boldsymbol{A_1}]\!], \ldots, [\![\boldsymbol{A_m}]\!], [\![\boldsymbol{b_1}]\!], \ldots, [\![\boldsymbol{b_m}]\!])$ from the servers, where m is the number of pairs, do:

1. For each $i \in [m]$, reconstruct $(A_{(i-1)k+1}, ..., A_{ik})$ from $[\![\boldsymbol{A_i}]\!]$ and reconstruct $(b_{(i-1)k+1}, ..., b_{ik})$ from $[\![\boldsymbol{b_i}]\!]$.
2. Receive a set of shares $\{u_i\}_{i\in\mathcal{C}orr}$ from the adversary.
3. Compute $\mathsf{out} = \prod_{i\in[m]} A_i^{b_i}$.
4. Sample a random sharing $\langle\mathsf{out}\rangle$ of out, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_i\}_{i\in\mathcal{C}orr}$.
5. Distribute the shares $\langle\mathsf{out}\rangle$ to all servers.

Figure 32: The functionality $\mathcal{F}_{\textsf{dMSM-Semi}}$

Protocol $\Pi_{\textsf{dMSM-Semi}}$

Let $A_1, ..., A_n$ be n group elements in $\mathbb{G}$ and $b_1, ..., b_n$ be n field elements in $\mathbb{F}$. The protocol allows N servers to collaboratively compute $\prod_{i=1}^{n} A_i^{b_i}$ in a distributed manner. Without loss of generality, we assume $k|n$, where k is the packing factor, and we set $m := n/k$. Each server holds packed secret shares $[\![\boldsymbol{A_j}]\!]$ of vectors $\boldsymbol{A}_j = \{A_{(j-1)k+i}\}_{i\in[k]}$ and $[\![\boldsymbol{b_j}]\!]$ of vectors $\boldsymbol{b}_j = \{b_{(j-1)k+i}\}_{i\in[k]}$, for each $j \in [m]$, respectively.

Preprocessing phase:

1. The servers invoke $\mathcal{F}_{\textsf{Double-Rand}}$ to prepare a pair of random shares $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$, where $\boldsymbol{r} \in \mathbb{F}^k$ is a random vector unknown to any server S_i.
2. The servers send $[\![\boldsymbol{r}]\!]_d$ to $\mathcal{F}_{\textsf{PSSToss-Semi}}$, which returns $\langle r_1\rangle, ..., \langle r_k\rangle$ to the servers.

Online phase:

1. Each server S_i computes $[\![\boldsymbol{C}]\!]_{2d} = \prod_{j\in[\frac{n}{k}]} [\![\boldsymbol{A_j}]\!]_d^{[\![\boldsymbol{b_j}]\!]_d}$.
2. Each server S_i computes $[\![\boldsymbol{D}]\!]_{2d} = [\![\boldsymbol{C}]\!]_{2d} \cdot g^{[\![\boldsymbol{r}]\!]_{2d}}$ and send it to S_1.
3. S_1 receives enough shares and reconstructs $\boldsymbol{D} = (D_1, ..., D_k)$.
4. S_1 computes $E = \prod_{j\in[k]} D_j$ and send it to each server.
5. Each server computes $\langle\mathsf{out}\rangle = \frac{E}{\prod_{j\in[k]} g^{\langle r_j\rangle}}$ as the output.

Figure 33: The $\Pi_{\textsf{dMSM-Semi}}$ Protocol .

C Collaborative proof generation for Libra

In [XZZ+19], Xie et al. introduced Libra, a zk-SNARK with $O(|\mathcal{C}|)$ prover time for arbitrary arithmetic circuit $\mathcal{C}$ of size $|\mathcal{C}|$. The high-level idea is to combine a linear-time sumcheck protocol with a PC scheme, as detailed in Section

2.3. In this section, we demonstrate how to utilize distributed primitives introduced in this work to implement a collaborative proof generation for data-parallel circuits, by securely computing the prover algorithm of Libra.

C.1 Distributed sumcheck protocol for GKR relations

We first detail how to adapt the distributed sumcheck protocol ($\Pi_{\mathsf{dSumcheck}}$) for GKR relations (Equation 2). First consider a specific term $\tilde{mult}_i(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})\tilde{V}_i(\boldsymbol{x})\tilde{V}_i(\boldsymbol{y})$ in this relation, referred as the GKR function in [XZZ+19]. In their study, the authors propose an efficient two-phase algorithm for this function. Here we give a quick review of this algorithm. Assuming $n = 2^\ell$, without loss of generality, the objective is to run sumcheck within $O(n)$ time on

$$\sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^\ell} f_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})f_2(\boldsymbol{x})f_3(\boldsymbol{y}) \tag{10}$$

for a specified challenge vector $\boldsymbol{g} \in \mathbb{F}^\ell$, where $f_2, f_3 : \mathbb{F}^\ell \to \mathbb{F}$ are multilinear extension of known n-sized arrays $\boldsymbol{A}_{f_2}, \boldsymbol{A}_{f_3}$, and $f_1 : \mathbb{F}^{3l} \to \mathbb{F}$ is a multilinear polynomial of a sparse array containing at most $n = 2^\ell$ non-zero elements (representing the gate label in the circuit). The linear-time algorithm involves two phases, each running a sumcheck for the product of two multilinear functions:

- In the first phase, run sumcheck on:

$$\sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^\ell} f_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})f_2(\boldsymbol{x})f_3(\boldsymbol{y}) = \sum_{\boldsymbol{x}\in\{0,1\}^\ell} f_2(\boldsymbol{x})h_g(\boldsymbol{x})$$

  where $h_g(\boldsymbol{x}) = \sum_{\boldsymbol{y}\in\{0,1\}^\ell} f_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})f_3(\boldsymbol{y})$. After this, the variables of $\boldsymbol{x}$ are bound to a random vector $\boldsymbol{u}$.

- In the second phase, run sumcheck on:

$$\sum_{\boldsymbol{y}\in\{0,1\}^\ell} f_1(\boldsymbol{g},\boldsymbol{u},\boldsymbol{y})f_2(\boldsymbol{u})f_3(\boldsymbol{y})$$

  after which the variables of $\boldsymbol{y}$ are bound to a random vector $\boldsymbol{v}$.

Since both of the two phases are similar, we focus on the first phase for illustration. We refer readers to Section 3 of [XZZ+19] for details. To run sumcheck on the products of $f_2(\boldsymbol{x})$ and $h_g(\boldsymbol{x})$, one need to firstly compute $\boldsymbol{A}_{h_g}$, the evaluations of $h_g(\boldsymbol{x})$ on $x \in \{0,1\}^\ell$, which presents challenge. To complete this, the authors suggest the single prover compute:

$$h_g(\boldsymbol{x}) = \sum G(\boldsymbol{z}) \cdot f_3(\boldsymbol{y}), \quad \text{for all } (\boldsymbol{z},\boldsymbol{x},\boldsymbol{y}) \text{ such that } f_1(\boldsymbol{z},\boldsymbol{x},\boldsymbol{y}) = 1 \tag{11}$$

where $G(\boldsymbol{z})$ is the multilinear extension of another n-sized array $\boldsymbol{A}_{\boldsymbol{g}}$, which can be easily computed. On the other hand, it takes $O(n)$ time for the prover to traverse all non-zero elements in the evaluations of $f_1(\boldsymbol{z},\boldsymbol{x},\boldsymbol{y})$ to figure out $\boldsymbol{A}_{h_g}$. After obtaining $\boldsymbol{A}_{\boldsymbol{h_g}}$, sumcheck on the product of f_2 and h_g can be performed in $O(n)$ time smoothly.

All the steps outlined above can be expedited by invoking Π_{sumcheck} and the PSS multiplications Π_{PSSMult}, except for Equation 11, due to the sparsity of f_1. Assuming the servers possess packed shares of the vectors $\boldsymbol{A}_{f_2}, \boldsymbol{A}_{f_3}$, and $\boldsymbol{A}_g$ (each share representing a small k-size vector), handling Equation 11 remains a challenge because the entries to compute are not well-aligned within the shares. To see this, one can consider a case that the two entries corresponding to $G(\boldsymbol{z})$ and $f_3(\boldsymbol{y})$ might not be at identical indices within the same shares. A realignment of these shares is required before performing PSS multiplication. This realignment involves not just internal permutations within the shares, but also value swapping across different shares. Such operations result in both $O(n)$ computational complexity and equivalent communication costs. Therefore, in the context of general circuits, it is not immediately clear how to leverage PSS to accelerate this process.

However, if $\mathcal{C}$ is data-parallel, this issue can be resolved. For data-parallel circuits, we can arrange the values at the identical positions of each sub-copy into a vector (assuming $B > k$, for simplicity). These values are then packed into a single element using the PSS scheme. Since these values are at identical positions in each copy, the computations performed on them will be consistent. Therefore, they are naturally aligned within the shares. This alignment allows Equation 11 to be processed in a SIMD fashion, utilizing PSS multiplications between the packed shares of $\boldsymbol{A}_{\boldsymbol{g}}$ and $\boldsymbol{A}_{\boldsymbol{f_3}}$. As a result, the computational complexity is reduced to $O(\frac{n}{k}) = O(\frac{n}{N})$, and the need for additional communications due to realignment is avoided.

Considering that every term in the GKR relation takes the form of Equation 10, it is feasible to execute the sumcheck protocol on GKR relations for data-parallel circuits. Under this approach, each server faces both computational and space complexities of $O(\frac{n}{N})$.

C.2 Detailed construction

In this subsection, we outline the construction of a collaborative zk-SNARK for data-parallel circuits, based on a zk-SNARK called Libra [XZZ$^+$19]. We will describe a version *without* the property t-zero-knowledge. This property can be achieved by adding random masking polynomials, following the methodology described in prior works [ZGK$^+$17b, XZZ$^+$19, ZXZS20, XZC$^+$22].

Consider $\mathcal{C}$ as a data-parallel circuit comprising B identical copies, where each copy is a d-depth layered arithmetic circuit. Suppose $\mathcal{C}$ contains n gates in each layer. Without loss of generality, we assume n, N, k, B are all powers of 2, with the provision for padding them if necessary. The servers collaboratively generate proof as follows.

Preprocessing phase. The servers invoke $\mathsf{dMKZG.Setup}$ in Π_{dMKZG} (Figure 5) to prepare public parameters, which will be used in the dMSM computation of $\mathsf{dMKZG.Commit}$ and $\mathsf{dMKZG.Open}$ procedure.

Online phase. On the input x and (extended) witness $\boldsymbol{w}$, the servers collaboratively generate proof for the relation $\mathcal{C}(x, \boldsymbol{w}) = 1$. Here, $(x, \boldsymbol{w})$ represents all wires in the circuit, namely $\{V_i\}_{i\in[0,d]}$, which is the evaluations of $\{\tilde{V}_i\}_{i\in[0,d]}$ on the hypercube $\{0,1\}^{\log n}$, respectively. The servers receive packed secret shares of $\{V_i\}_{i\in[0,d]}$ as inputs, denoted as $\{[\![\boldsymbol{V_i}]\!]\}_{i\in[0,d]}$. Specifically, variables in the same position of each sub-copy are packed together.

The following procedure runs under the random oracle model, where we use H to denote a random oracle. When we say each instance of the servers makes a random query to H, it means that the servers reconstruct shares of the transcript $\mathcal{T}$ and each of them locally makes an oracle query to receive the same random challenge $r = \mathsf{H}(\mathcal{T})$.

1. The servers invoke $\mathsf{dMKZG.Commit}$ in Π_{dMKZG} (Figure 5) to compute the commitment $\mathsf{com}_{\tilde{V}_d}$.

2. The servers make a random query to H and receive a challenge $\boldsymbol{g} \in \mathbb{F}^{\log n}$.

3. The servers generate proof for $\tilde{V}_0(\boldsymbol{g}) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^{\log n}} \tilde{add}_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})(\tilde{V}_1(\boldsymbol{x})+\tilde{V}_1(\boldsymbol{y}))+\tilde{mult}_1(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})\tilde{V}_1(\boldsymbol{x})\tilde{V}_1(\boldsymbol{y})$. This involves running 3 sumcheck for GKR functions, which can be facilitated by invoking $\Pi_{\mathsf{dSumcheck}}$ (Figure 4) and Π_{PSSMult} (Figure 14). After this process, the servers obtain two random vectors $\boldsymbol{u}^{(1)}, \boldsymbol{v}^{(1)}$ from the sumcheck protocols.

4. In each layer $i = 1, ..., d-1$, the servers make two random queries to H and receive two random challenges $\alpha_i, \beta_i \in \mathbb{F}$. The servers collaboratively generate proof for $\alpha_i\tilde{V}_i(\boldsymbol{u}^{(i)})+\beta_i\tilde{V}_i(\boldsymbol{v}^{(i)}) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^{\log n}} f_i(\tilde{V}_{i+1}(\boldsymbol{x}), \tilde{V}_{i+1}(\boldsymbol{y}))$, where f_i is the GKR relation, as defined by Equation 2. This involves running sumcheck for each GKR function, which can be facilitated by our protocols as well. After this process, the servers obtain two random vectors $\boldsymbol{u}^{(i+1)}, \boldsymbol{v}^{(i+1)}$ from the sumcheck protocols.

5. At the input layer, the servers invoke $\mathsf{dMKZG.Open}$ in Π_{dMKZG} to generate proofs for evaluations $f(\boldsymbol{u}^{(d)})$ and $f(\boldsymbol{v}^{(d)})$.

Since there are $O(d)$ calls to $\Pi_{\mathsf{dSumcheck}}$ and Π_{PSSMult}, and one call to $\mathsf{dMKZG.Commit}$ and $\mathsf{dMKZG.Open}$, the total proving work is of $O(d \cdot n + n) = O(|\mathcal{C}|)$, and the computational and space overhead for each server are $O(\frac{|\mathcal{C}|}{N})$.