Scalable Collaborative zk-SNARK and Its Application to Efficient Proof Outsourcing *

Xuanming Liu[‡], Zhelei Zhou[‡,§], Yinghao Wang[‡], Jinye He[¶], Bingsheng Zhang[‡,§], Xiaohu Yang [†‡], and Jiaheng Zhang[¶]

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

June 12, 2024

Abstract

Collaborative zk-SNARK (USENIX'22) allows multiple parties to jointly create a zk-SNARK proof over distributed secrets (also known as the witness). It provides a promising approach to proof outsourcing, where a client wishes to delegate the tedious task of proof generation to many servers from different locations, while ensuring no corrupted server can learn its witness (USENIX'23). Unfortunately, existing work remains a significant efficiency problem, as the protocols rely heavily on a particularly powerful server, and thus face challenges in achieving scalability for complex applications.

In this work, we address this problem by extending the existing zk-SNARKs Libra (Crypto'19) and HyperPlonk (Eurocrypt'23) into scalable collaborative zk-SNARKs. Crucially, our collaborative proof generation does not require a powerful server, and all servers take up roughly the same proportion of the total workload. In this way, we achieve privacy and scalability simultaneously for the first time in proof outsourcing. To achieve this, we develop an efficient MPC toolbox for a number of useful multivariate polynomial primitives, including sumcheck, productcheck, and multilinear polynomial commitment, which can also be applied to other applications as independent interests. For proof outsourcing purposes, when using 128 servers to jointly generate a proof for a circuit size of 2^{24} gates, our benchmarks for these two collaborative proofs show a speedup of $21\times$ and $24\times$ compared to a local prover, respectively. Furthermore, we are able to handle enormously large circuits, making it practical for real-world applications.

*This work is an extensive update of a previous work, which can be found at https://eprint.iacr.org/2024/143. The update includes semi-honest protocols for collaborative HyperPlonk, sub-protocol used by collaborative Libra, additional optimizations, and new experimental results.

†The corresponding author: Xiaohu Yang, email: yangxh@zju.edu.cn.

Contents

1 Introduction

Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) allows a prover to produce a short proof convincing that his secret data (i.e., the witness) satisfies a complex relation. This technique has broad applications, such as blockchain [24, 42], verifiable machine learning [25, 44], and verifiable program execution [2, 40]. A major limitation of existing zk-SNARKs is that proof generation is cumbersome, taking a long time to produce a proof while consuming a large amount of memory. Moreover, experiments show that even with a prover-efficient zk-SNARK called HyperPlonk [7], it still takes several hours and over 300 GB of memory to generate a proof for a circuit consisting of 2^{27} gates. Handling proof generation for such large applications is difficult even with a powerful server, let alone a lightweight client.

A natural solution is to allow the client outsourcing proof generation to a cluster of servers that are responsible for generating the proof in a distributed manner in exchange for payment. Distributed proof generation has been explored in several studies [24, 41, 42]. By distributing the workload across servers, these works achieve impressive speedups, and can handle large circuits that neither a lightweight client nor a single powerful server can handle. Their results are therefore *scalable*. However, these works assume that the servers are "harmless", directly exposing the client's secret witness to the cluster. This assumption may not always hold, especially if the witness is sensitive. For example, a client may wish to outsource the tedious proof for the correct execution of a machine learning program [25], while being unwilling to reveal its input. Therefore, the goal of proof outsourcing is twofold: let servers from different locations (i) efficiently generate the proof desired by the client, even for enormously large circuits, while (ii) protecting the privacy of the client's witness.

Recently, Garg et al. [16] have attempted to achieve this goal by proposing an innovative notion called *zk-SaaS*, which extends the *collaborative zk-SNARK framework* introduced by Ozdemir and Boneh [28] into the proof outsourcing scenario. In [16], a client is allowed to delegate its proof to many servers, each of which holds a secret-shared witness from the client, and work together to generate a proof without knowing the witness. At its core is a secure Multi-Party Computation (MPC) protocol that enables the participants to collaboratively compute the proof of a zk-SNARK. As a result, they implement collaborative zk-SNARKs for Groth16 [22] and Plonk [15]. However, an important efficiency issue remains unresolved, which is listed below.

Problem: lack of scalability. The collaborative proof generation in [16] is not scalable. More precisely, in their protocols there is still a leader server with $O(T_{\mathsf{P}})$ time and $O(S_{\mathsf{P}})$ space complexity, where T_{P} and S_{P} denote those for a local zk-SNARK prover. This indicates that the leader server requires almost the same time and memory as in the local case to dominate the collaborative proof. Furthermore, our experiments show that the efficiency of zkSaaS mainly depends on the hardware advantages of the leader, such as high computational parallelism and large memory capacity. While these advantages are possible for a powerful server, the efficiency achieved is always limited. In contrast, collaborative proof generation is expected to distribute the total workload evenly across servers, thereby efficiently handling larger circuits as proof outsourcing requires. Thus, in [16], the authors leave this as a research question: *In proof outsourcing, can we eliminate such a powerful leader server and achieve collaborative zk-SNARKs with scalable proof generation?*

We find that [16] fails to achieve this, mainly because of a sub-protocol that attempts to distribute the FFT operation for univariate polynomials, which is a core of the zk-SNARKs they study. Specifically, they find it difficult to distribute the workload of FFT evenly among the servers, so a leader is chosen to do most of the work, and all servers must communicate with the leader, creating a bottleneck for the leader. In contrast, our goal is to eliminate such a prominent leader and achieve scalable collaborative zk-SNARKs where each participant takes on the same workload.

Our techniques. To achieve the efficiency goals, we turn to a category of multivariate polynomial-based zk-SNARKs that avoid the use of FFT, such as the GKR-based ones [39, 43, 45], HyperPlonk [7], and Spartan [33]. The provers of these zk-SNARKs are efficient, making them suitable to deal with proof outsourcing. Nevertheless, they still face time and memory challenges when dealing with large circuits, which are common in real-world applications, as the earlier example shows. Therefore, it still motivates us to improve the efficiency of proof generation for these zk-SNARKs. For example, a previous work [42] also focuses on distributed proof generation for a GKR-based zk-SNARK called Virgo [45], though in a different setting than ours.

We adopt the packed secret sharing [14] technique as the main weapon to improve the computation efficiency. As a result, our protocols have been shown to be secure against a semi-honest adversary controlling at most $\frac{N}{2} - k$ servers, where N is the number of servers and k is the packing factor of packed secret sharing. The packed secret sharing weapon is common in existing MPC protocols, however, the main challenge is to use it throughout the entire proof generation without the help of a single powerful server. It is difficult because there are many different primitives throughout the process, while it remains unclear how to fully distribute many of them with packed

Ref.	Underlying zk-SNARK(s)	Time		Space		Priv.
		S_1	S_i	S_1	S_i	
[42]	GKR-based	$O(\frac{T_\mathsf{P}}{N})$		$O(\frac{S_\mathsf{P}}{N})$		✗
[24]	Plonk	$O(\frac{T_\mathsf{P}}{N})$		$O(\frac{S_\mathsf{P}}{N})$		✗
[16]	Groth16, Plonk	$O(T_\mathsf{P})$	$O(\frac{T_\mathsf{P}}{N})$	$O(S_\mathsf{P})$	$O(\frac{S_\mathsf{P}}{N})$	✓
Ours	GKR-based, HyperPlonk	$O(\frac{T_\mathsf{P}}{N})$		$O(\frac{S_\mathsf{P}}{N})$		✓

Table 1: Comparisons of schemes distributing different zk-SNARKs. $T_\mathsf{P}, S_\mathsf{P}$ denote the time complexity and space complexity of a local prover. $\mathsf{S}_1, \mathsf{S}_i$ denote the leader server and other servers in a scheme if they have different properties. Priv. denotes the privacy of witness.

secret sharing. This is exactly the difficulty that zkSaaS encounters.

Our main technical contribution is a novel toolbox for computing multivariate polynomial-related primitives in a privacy-preserving yet efficient manner. These primitives, including sumcheck, productcheck, and multilinear polynomial commitment, are widely used by zk-SNARKs we study. It was unknown how to get servers to compute these primitives together using only secret-shared polynomials. Building on the secret-sharing technique and many new ideas in this work, we design novel and efficient protocols for each of the primitives. Impressively, all protocols ensure that the total workload of the corresponding primitive is evenly distributed. Furthermore, we perform several optimizations to ensure that the communication complexity is also evenly distributed, and that all protocols maintain a constant round complexity. This comes at the cost of assuming a *peer-to-peer* network between servers. Considering the proof outsourcing scenario where the servers are separate and far apart, we believe such a network topology is reasonable.

Finally, these tools can be combined, allowing us to replace the primitives in the original zk-SNARKs with the collaborative tools to obtain collaborative zk-SNARKs with scalable proof generation for the first time.

Our results. We answer the previous research question by instantiating two scalable collaborative zk-SNARKs from Libra [43] and HyperPlonk [7]. Both protocols remain the privacy of witness. Crucially, each server in the protocols consumes similar amount of time and memory, proportional to the total workload. The total communication costs are also shared among the servers, allowing our protocols to achieve significant efficiency improvements even on a limited network. Our protocols are applicable to different applications: collaborative Libra is designed for *data-parallel circuits* and is concretely fast even when handling circuits with billions of gates, while collaborative HyperPlonk is able to handle more expressive *general circuits*. Furthermore, due to the similar recipe, other aforementioned zk-SNARKs such as Spartan [33] also have the potential to be extended to scalable collaborative zk-SNARKs following the same path.

The collaborative zk-SNARKs and the collaborative primitive protocols are implemented in a modular fashion and provided for further research. Evaluation results show that our protocols are efficient, scalable, and cost-effective:

- When 128 servers are working in a good network, our collaborative proofs for Libra and HyperPlonk take 8 seconds and 4 minutes respectively to handle a proof outsourcing for a circuit of size 2^{24}, while a local prover takes more than 2 minutes and 1.5 hours, indicating a $21\times$ and $24\times$ speedup for our proposals, respectively. This efficiency improvement is even more significant when dealing with larger circuits. Moreover, with 128 servers, we can scale to circuits $32\times$ larger than those a local prover can handle.
- Compared to zkSaaS [16] in the same setting, we observe significant memory and time savings for each server with our proposals. However, the leader server in zkSaaS still exhibits high memory consumption and achieves limited efficiency gains even with more servers involved. Moreover, when switching to a network with limited capacity, the zkSaaS scheme does not achieve any efficiency improvement, while our collaborative HyperPlonk still saves significant time. A complementary financial calculation also demonstrates our low financial cost.

1.1 Related works

There is a nice line of work in the literature that focuses on enabling a group of N servers to jointly generate a zk-SNARK proof. A comparison of some representative works is presented in Table 1. We observe that these

works can be divided into two types, based on whether the servers are allowed to learn the client's input (i.e. the witness).

Witness is exposed. Many works [24,41,42] show how to propose multiple servers working in tandem to generate a proof. For example, the recent works deVirgo [42] and Pianist [24] design distributed proofs for a GKR-based zk-SNARK called Virgo [45] and Plonk [15] respectively. We stress that their work is orthogonal to ours: these approaches distribute a prover's workload evenly across the cluster, allowing zk-SNARKs to scale efficiently to larger circuits. However, since they assume that the servers are honest and allow direct witness disclosure to the servers, these protocols are not suitable for many proof outsourcing applications where the client's inputs are sensitive.

Witness is secret-shared. Recently, works such as [8, 12, 16, 28, 32] have discussed how to delegate proofs to servers *without* revealing the witness. A representative approach relies on the notion of collaborative zk-SNARK introduced by Ozdemir and Boneh [28]. This approach has two phases: First, each server receives a secret-shared testimony, rather than the entire testimony. Then, the servers N execute an MPC protocol to complete the proof generation and finally obtain the proof. [28] implements collaborative proof through generic MPC protocols in both the honest majority setting [21] and the dishonest majority setting [10]. However, the protocols do not bring any efficiency gains to proof generation. On the other hand, Garg et al. [16] extend this approach for the first time to the proof outsourcing scenario, formalizing a framework called zkSaaS. Using packed Shamir's secret sharing scheme [14] to improve efficiency, they design specific MPC protocols for primitives such as FFT and MSM, and combine the primitives to design collaborative proofs for Groth16 [22] and Plonk [15]. However, their protocol relies heavily on a particularly powerful leader server, which limits scalability. In contrast, we aim to remove such assumptions and build scalable collaborative proofs where each normally equipped server shares the same workload of $O(\frac{T_\mathsf{P}}{N})$ and $O(\frac{S_\mathsf{P}}{N})$.

There are other works in the literature [8, 12] that try to distribute proof generation in a different setting: they assume that there is a special party (in [8], it is called a delegator; while in [12], it is called an aggregator) that is always honest and online, while the rest of the servers can be corrupted by an adversary. Note that this special party is required to stay online and participate in the protocol throughout proof generation. In contrast, we do not assume such a special server during proof generation.

2 Preliminaries

Notations. 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 $\{a, \ldots, b\}$. Let $\mathbb{F}$ be a large finite field with a prime order such that $|\mathbb{F}|^{-1} = \mathsf{negl}(\lambda)$. Let $(\mathbb{G}, \mathbb{G}_T)$ be cyclic groups of prime order q with generator $g \in \mathbb{G}$.

Multilinear extension. We say a polynomial f is multilinear if it is a multivariate polynomial whose degree in each variable is at most one. A multilinear extension of a function $V : \{0,1\}^\ell \to \mathbb{F}$ can be defined 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, the multilinear polynomial $\tilde{V}$ can be expressed as:

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

Here $\beta_{b_i}(x_i) = (1 - x_i)(1 - b_i) + x_i b_i$ and b_i is i-th bit of $\boldsymbol{b}$. For any $\boldsymbol{r} \in \mathbb{F}^\ell$, $\tilde{V}(\boldsymbol{r})$ can be computed in $O(2^\ell)$ field operations [38].

2.1 zk-SNARK

Definition 1. *A tuple of three algorithms* $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ *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* $\mathcal{R}(x, w) = 1$*, we have*

$$\Pr\left[\langle \mathsf{P}(w), \mathsf{V} \rangle(x, \mathsf{pp}) = 1\right] = 1$$

- ***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 probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\langle \mathsf{P}^*, \mathsf{V}\rangle(x, \mathsf{pp}) = 1 \wedge \mathcal{R}(x, w^*) \neq 1\right]$$

- ***Zero-knowledge***. *There exists a PPT simulator* $\mathcal{S}$ *that for any PPT* V^**,* $\mathcal{R}(x, w) = 1$*,* pp *output by* $\mathsf{G}(1^\lambda)$*, we have*

$$\mathsf{View}^{\mathsf{V}^*}(\langle \mathsf{P}(w), \mathsf{V}^*\rangle(x, \mathsf{pp})) \approx \mathcal{S}^{\mathsf{V}^*}(x)$$

  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 can be made non-interactive by applying the Fiat-Shamir transformation [13]. We say the argument is *succinct* if the running time of the verifier and the total 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*, constituting a tuple of algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$:

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

2.2 Secure Multi-Party Computation

Let $C : (\{0,1\}^\lambda)^N \to (\{0,1\}^\lambda)^N$ be a circuit and let $\mathsf{P}_1, \ldots, \mathsf{P}_N$ be the parties that will participate in a secure Multi-Party Computation (MPC) protocol Π for 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 $(y_1, \ldots, y_N) := C(x_1, \ldots, x_N)$, without revealing its private input.

We analyze the security of the MPC protocol Π in the real-world/ideal-world paradigm [5]. Here we provide a high-level description for this paradigm, and more details and be found in [5]. In real-world execution, the real parties $\mathsf{P}_1, \ldots, \mathsf{P}_N$ communicate with each other to execute Π, and there is an adversary $\mathcal{A}$ who can choose a set of parties to corrupt. The set of the corrupted parties is denoted by $\mathcal{C}orr$. In this work, we consider a *semi-honest adversary* as in [16], i.e., the corrupted parties will honestly follow the protocol instructions but are curious about others' private input. In ideal-world execution, there are dummy parties $\tilde{\mathsf{P}}_1, \ldots, \tilde{\mathsf{P}}_N$, 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 $\mathcal{S}$, then computes 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, 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 $\mathcal{G}$.

Recall that, 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 that computes Prove, we mean that Prove can be represented as a circuit, and Π securely realizes an ideal functionality which computes this circuit. Similar treatments can be found in [16, 28].

2.3 Packed secret sharing

In this work, we utilize packed secret sharing (PSS) scheme introduced by Franklin and Yung [14], which is a generalization of the well-known Shamir's secret sharing scheme [36]. Suppose $\boldsymbol{x} = \{x_1, ..., x_k\}$ is a vector of k secrets, where k is called the packing factor. The dealer picks 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 S_i 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. Accordingly, we use $\langle x \rangle$ to denote a regular Shamir's secret sharing. We recall two properties of PSS in the following. For any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$ and $d \geq k-1$:

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

- Multiplication: For all $d_1, d_2 \geq k-1$ subject to $d_1 + d_2 < N$, $[\![\boldsymbol{x} * \boldsymbol{y}]\!]_{d_1+d_2} = [\![\boldsymbol{x}]\!]_{d_1} \cdot [\![\boldsymbol{y}]\!]_{d_2}$, where $*$ represents a coordinate-wise multiplication.

The first property implies that linear combination can be performed locally by parties. Recall that, if we denote by t the number of corrupted parties, the PSS scheme is secure against $t \leq d - k + 1$ corrupted parties. Jumping ahead, we require $2d + 1 \leq N$ to ensure that we can multiply two PSS in our protocols. When $2d = N - 1$, it is easy to see that $t \leq \frac{N}{2} - k$ holds.

3 Scalable Collaborative zk-SNARKs

A client wants to outsource a proof to a group of dedicated participants from different locations without revealing the witness. Besides ensuring privacy, the main task is to make collaborative proof generation *scalable*, which implies two main goals: (i) speeding up proof generation, and (ii) reducing the memory consumption of each server, so that we can handle complex applications.

3.1 Collaborative zk-SNARK

First, we talk about the privacy of the witness. Similar to previous work [16], we also work within the collaborative zk-SNARK framework [28], where N servers collaborate together to generate a proof for a given statement. At its core is an MPC protocol Π that allows servers to efficiently collaborate on executing the prover algorithm of an existing zk-SNARK, without leaking the witness to any corrupted server. For reader's convenience, we present the formal definition of collaborative zk-SNARK, adapted from [28].

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 secret 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 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:

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

- ***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,

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

- ***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 $\mathsf{poly}(\lambda, |x|, \log|\boldsymbol{w}|)$.

Previous work [28] proved that 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 corresponding collaborative zk-SNARK $(\mathsf{Setup}, \Pi, \mathsf{Verify})$. Due to this result, our main focus in this work is the design of such MPC protocols Π.

Security model. Our security model is similar to existing framework in [16]. We assume an honest-majority setting that is realistic in practice, i.e., the adversary can only corrupt a minority of the servers. Concretely, let k be the packing factor of the packed secret sharing scheme we adopt, our framework can be proven to be secure against at most $t = \frac{N}{2} - k$ corrupted servers. Aligning with [16], we mainly consider a semi-honest adversary in this work. For malicious security, we put our discussion in Section 7.

3.2 Scalable proof outsourcing

Proof outsourcing is a direct application of the collaborative zk-SNARK framework. To outsource a proof, a client first performs very cheap local computations to obtain the (extended) witness, and then distributes the witness among the servers using packed secret sharing. As noted in [16, 28], this step is neither a central concern nor an efficiency bottleneck, because computing and sharing the witness is less resource-intensive for the client itself compared to proof generation. Therefore, we mainly focus on how to make proof generation scalable. We first introduce two properties to represent the aforementioned scalability goals:

Efficiency goals. For a given NP relation $\mathcal{R}$, let T_{P} and S_{P} be the time and space complexity of Prove for a local zk-SNARK prover, respectively. We say that the proof generation of the collaborative zk-SNARK is *time-efficient* if all servers in Π have the same time complexity $O(\frac{T_{\mathsf{P}}}{N})$. Similarly, collaborative proof generation is *space-efficient* if all servers in Π have the same space complexity $O(\frac{S_{\mathsf{P}}}{N})$. These properties are important for proof outsourcing: In terms of time complexity, proof generation can be made more efficient by increasing the number of servers. In addition, memory usage is also averaged, so scalability is not limited by the capacity of a single server, allowing larger circuits to be handled if more servers are available.

Here we discuss the complexity comparison between the above definition and previous work. The time and space complexity of the leader server in [16] are $O(T_{\mathsf{P}})$ and $O(S_{\mathsf{P}})$, respectively. This has two limitations: (i) efficiency improvement cannot be achieved without assuming the hardware advantages of the leader server, since the time complexity is the same as the local prover; (ii) there will still be a memory bottleneck for the leader server when dealing with large circuits, since it does not improve memory usage. In contrast, time and space efficient proof generation can theoretically scale to larger circuits with better efficiency if more servers with standard equipment are available. This is especially useful for proof outsourcing, where large circuits are to be handled by participants with normal machines.

Another bottleneck introduced by the "partially distributed" protocol from [16] is the large communication overhead the leader has to take. Instead, we try to minimize the communication complexity that each server has to bear. Assuming that the total communication complexity of proof generation is $O(C)$, we expect that it can also be distributed among the servers, namely that each server undertakes only $O(\frac{C}{N})$ communication. We note that this leads to a peer-to-peer network among the servers, which is feasible in our proof outsourcing service. Finally, the round complexity during proof generation should also be minimized.

Pre-processing. For protocols that require pre-processing to generate randomness and other aids needed by servers during execution, we adopt the same model as in [16, 28]. Specifically, the required randomness can be generated either by the client or by an MPC protocol between the servers in offline phase when the inputs are unknown to the parties. This is acceptable in practice: (i) the randomness is unrelated to the outsourced relation, so it can be prepared when the servers are idle, and (ii) the time cost of generating such randomness is small compared to proof generation. Therefore, when designing the protocol Π in the pre-processing model, we only evaluate its online complexity.

4 Collaborative Multivariate Primitives

This difficulty of [16] stems primarily from the challenges of distributing the FFT operation, which is a core of their primitives for univariate polynomials (e.g., polynomial division). To achieve scalable collaborative zk-SNARKs and address the remaining research question, we turn to another class of zk-SNARKs based on multivariate polynomials. As discussed earlier, even though these zk-SNARKs are already prover-efficient, it is still hard for a single machine to handle complex applications that are often found in proof outsourcing. Therefore, it is meaningful to design scalable collaborative zk-SNARKs for them. Our approach is to first develop collaborative protocols for primitives used in these zk-SNARKs, and then combine them to instantiate scalable collaborative zk-SNARKs. In the following, we provide an overview of the primitives that are commonly used in multivariate polynomial-based zk-SNARKs.

Sumcheck. The zk-SNARKs studied in this work are built upon multivariate-based interactive proofs, where the sumcheck protocol [26] is a fundamental building block. Given an ℓ-variate polynomial f and a claim $H \in \mathbb{F}$, a sumcheck enables a prover to convince a verifier that $H = \sum_{\boldsymbol{x}\in\{0,1\}^\ell} f(\boldsymbol{x})$. The protocol consists of ℓ rounds: during the i-th round of the protocol, the prover sends a univariate polynomial $f_i(x) = \sum_{b_{i+1},\ldots,b_\ell\in\{0,1\}} f(r_1, ..., r_{i-1}, x, b_{i+1}, ..., b_\ell)$ to the verifier. The verifier checks whether $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$ holds, where r_{i-1} is a challenge sent by verifier in the previous round. At the end of the protocol, the verifier checks the consistency between $\{f_i\}$ and the multivariate polynomial through a single query to f. We note that the authors in [16, 28] also studied a collaborative "sumcheck", but their target is a univariate polynomial, which is very different from ours. In particular, the core of their approach is a polynomial division facilitated by FFTs, while our goal is essentially a multi-round interactive proof without FFTs. Sumcheck on a secret-shared multivariate polynomial has never been studied before.

Zerocheck. Spartan [33] and HyperPlonk [7] rely on a protocol to check that an ℓ-variate polynomial f evaluates to zero on the hypercube $\{0,1\}^\ell$. In the protocol, the prover first constructs a polynomial $\tilde{\mathsf{eq}}(\boldsymbol{r}, \boldsymbol{x}) = \prod_{i=1}^{\ell}(1-x_i)(1-r_i) + x_i r_i$, where $\boldsymbol{r} \in \mathbb{F}^\ell$ is a challenge from the verifier. Then, the prover and verifier run a sumcheck protocol to verify that $0 = \sum_{\boldsymbol{x}\in\{0,1\}^\ell} f(\boldsymbol{x}) \cdot \tilde{\mathsf{eq}}(\boldsymbol{r}, \boldsymbol{x})$ holds. This protocol is essentially a special case of the sumcheck protocol.

Productcheck. Productcheck is another important building block that is widely used in [7, 34, 35]. Similar to sumcheck, the productcheck protocol aims to convince that $H = \prod_{\boldsymbol{x}\in\{0,1\}^\ell} f(\boldsymbol{x})$ for a given multivariate polynomial f and claim H. The core task is to compute an auxiliary polynomial v related to the evaluations of f on $\{0,1\}^\ell$, which will be elaborated in Section 4.2. The productcheck protocol can be used for constructing other useful primitives like permutation-check and multiset-check [7, Section 3].

Multivariate polynomial commitment. In the recipe for the aforementioned zk-SNARKs, it is essential to combine a multivariate polynomial commitment scheme with the interactive proofs to obtain a zk-SNARK. A multivariate polynomial commitment scheme allows the prover to commit to an ℓ-variate polynomial f and later evaluate it at some point $\boldsymbol{x}$. In particular, in this work we focus on a multilinear polynomial commitment scheme (mvPC) adapted from Papamenthous et al. [29]. It requires a trusted setup:

- $\mathsf{mvPC.Setup}(1^\lambda, \mathcal{F}) \to \mathsf{pp}$: It samples $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}^\ell$ as a trapdoor and outputs parameters $\mathsf{pp} = \{\{g^{\prod_{i\in W} s_i}\}_{W\in\mathcal{W}_\ell}\}$, where $\mathcal{W}_\ell$ is the collection of all subsets of $\{1, \ldots, \ell\}$ and g is a generator of the group $\mathbb{G}$.

The prover of mvPC consists of the following algorithms:

- $\mathsf{mvPC.Commit}(f, \mathsf{pp}) \to \mathsf{com}_f$: It outputs $\mathsf{com}_f = g^{f(\boldsymbol{s})}$. Note that the evaluation on the power of g can be computed with the aid of prepared parameters.
- $\mathsf{mvPC.Open}(f, \boldsymbol{u}, \mathsf{pp}) \to (z, \pi)$: It evaluates the polynomial at a given point $\boldsymbol{u}$ as $z = f(\boldsymbol{u})$. It also computes polynomials $\{Q_i\}_{i\in[\ell]}$ that satisfy $f(\boldsymbol{x}) = \sum_{i=1}^{\ell}(x_i - u_i)\cdot Q_i(x_{i+1}, \ldots, x_\ell) + z$ and outputs a proof $\pi = \{\pi_i\}_{i\in[\ell]}$, where $\pi_i = g^{Q_i(s_{i+1},\ldots,s_\ell)}$.

4.1 Collaborative sumcheck

The first task is to distribute the prover of the sumcheck protocol with only a secret-shared polynomial. First, we consider a case where the target polynomial f is multilinear, where the degree in each variable in f is at most one.

Prover for multilinear polynomial. Given a multilinear polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$, Thaler introduces an algorithm to execute the prover algorithm in $O(2^\ell)$ time [37]. Here we provide a concise overview of it: Given that $f_i(x)$ is linear, it is sufficient for the prover to simply provide $f_i(0)$ and $f_i(1)$ in each round. To expedite the computation of

$f_i(0)$ and $f_i(1)$, the core idea is to let the prover maintain a *bookkeeping table*. The first row of the table is initialized with 2^ℓ entries according to the evaluations of f on $\{0,1\}^\ell$. In the i-th row, there are $n_i = 2^{\ell-i+1}$ entries to be stored. An important observation to note is the existence of a relationship for entries in successive rows of the bookkeeping table. For $\boldsymbol{b^{(i)}} \in \{0,1\}^{\ell-i+1}$ in the i-th row:

$$f(r_1,\ldots,r_{i-1},r_i,\boldsymbol{b^{(i)}}) = (1-r_i)\cdot f(r_1,\ldots,r_{i-1},0,\boldsymbol{b^{(i)}}) + r_i \cdot f(r_1,\ldots,r_{i-1},1,\boldsymbol{b^{(i)}}) \tag{2}$$

Both $f(r_1,\ldots,r_{i-1},0,\boldsymbol{b^{(i)}})$ and $f(r_1,\ldots,r_{i-1},1,\boldsymbol{b^{(i)}})$ were computed in the previous round. Therefore, in the i-th round ($i \in [\ell]$) of the sumcheck protocol, given challenge r_i, prover utilizes Equation 2 to compute $2^{\ell-i}$ entries in row $i+1$ of the values $f(r_1,\ldots,r_i,\boldsymbol{b})$ for all $\boldsymbol{b} \in \{0,1\}^{\ell-i}$. These entries are stored in the table for the next round computation. To obtain the claims $f_i(0)$ and $f_i(1)$ in each round, prover sums up the first and second halves of the entries in the table, respectively.

In our setting, the servers cooperate together and compute the bookkeeping table without knowing the polynomial f. The start point is each server S_i receive only packed secret shares of witness, which is the evaluations of f on the hypercube $\{0,1\}^\ell$. More precisely, the $n_1 = 2^\ell$ entries, denoted as $x_1^{(1)},\ldots,x_{n_1}^{(1)}$, are divided into $\frac{n_1}{k}$ groups $\{\boldsymbol{x_j^{(1)}}\}_{j\in[\frac{n_1}{k}]}$, where k is the packing factor. And each vector is packed secret shared to the servers, denoted as $[\![\boldsymbol{x_j^{(1)}}]\!]_{j\in[\frac{n_1}{k}]}$.

Local computation via PSS. We hope to distribute the workload evenly among the servers, thereby accelerating the prover and reducing the memory cost. The key challenge is to design a protocol that allows each server to perform computations separately and minimize the communication overhead. To achieve this, a important observation is that Equation 2 exhibits a well-formed SIMD structure: in the i-th row, each pair of $x_j^{(i)}$ and $x_{j+\frac{n_i}{2k}}^{(i)}$ ($j \in [\frac{n_i}{2k}]$) are linearly combined to obtain an entry in the next row. This property is well captured by packed secret sharing: given shares, the computation can be done locally by each server over corresponding share pairs. Formally, in the i-th round, each server locally computes $O(\frac{n_{i+1}}{k})$ packed shares of entries needed in the next round as

$$[\![\boldsymbol{x_j^{(i+1)}}]\!] = (1-r_i)\cdot[\![\boldsymbol{x_j^{(i)}}]\!] + r_i\cdot[\![\boldsymbol{x_{j+\frac{n_i}{2k}}^{(i)}}]\!], \quad j \in [\frac{n_{i+1}}{k}] \tag{3}$$

Since the result of the above linear combination is still in the form of packed shares, the computation can be repeated round by round. Moreover, each server can obtain the packed shares of claims needed in each round as: $[\![\boldsymbol{a_1^{(i)}}]\!] = \sum_{j=1}^{\frac{n_i}{2k}}[\![\boldsymbol{x_j^{(i)}}]\!]$ and $[\![\boldsymbol{a_2^{(i)}}]\!] = \sum_{j=\frac{n_i}{2k}+1}^{\frac{n_i}{k}}[\![\boldsymbol{x_j^{(i)}}]\!]$, where $[\![\boldsymbol{a_1^{(i)}}]\!]$ and $[\![\boldsymbol{a_2^{(i)}}]\!]$ are two shares, each packing k elements.

Further computation. In the above phase, we eliminate the need for communication and evenly distribute the workload. However, it is easy to see that the local computations will be stuck in the $(\ell-s+1)$-th round, where $s = \log k$. This is because each round the number of shares is halved, and when it comes to the end, each server possesses only one share, making further computations unfeasible. To facilitate the remaining rounds of computation, we propose converting the last packed share into Shamir's secret shares through one round of communication. Specifically, each server obtains the Shamir's shares of the k elements in vector $\boldsymbol{x^{(\ell-s+1)}}$ as $\langle x_1\rangle,\ldots,\langle x_k\rangle$. Hereafter, the remaining work can be completed on the shares locally again, following the original sumcheck, within only $O(k)$ time. Note that, to facilitate this conversion, the servers need a standard procedure to convert packed shares to Shamir's shares, which is modeled as a functionality $\mathcal{F}_{\mathsf{PSSToSS}}$. We put the detailed descriptions of $\mathcal{F}_{\mathsf{PSSToSS}}$ and its protocol in Appendix A.3.

Our protocol. Putting the two phases together, we present a novel collaborative sumcheck protocol $\Pi_{\mathsf{dSumcheck}}$ in Figure 1. We denote by Sumcheck.Prove the prover algorithm of the original sumcheck protocol for a given ℓ-variate multilinear polynomial $f(\boldsymbol{x})$. We have the following theorem:

Theorem 1. *The protocol $\Pi_{\mathsf{dSumcheck}}$ depicted in Figure 1 is a secure MPC protocol that computes* Sumcheck.Prove *in the $\mathcal{F}_{\mathsf{PSSToSS}}$-hybrid world against a semi-honest adversary who corrupts at most t servers.*

Proof sketch. 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 ($i \in [l-s+1, l]$), $f_i(0) = a_1^{(i)}$ and $f_i(1) = a_2^{(i)}$ holds. Therefore, the correctness of the protocol is straightforward. Except for invoking $\mathcal{F}_{\mathsf{PSSToSS}}$, the servers only perform local computation. Therefore, there is no chance that the corrupted parties can learn others' private input. This ensures the security of the protocol. □

Efficiency. This protocol is both time-efficient and space-efficient. From round 1 to $\ell-s$, each server individually performs $O(\frac{n}{k}) = O(\frac{n}{N})$ field operations. From round $\ell-s+1$ to ℓ, each server individually does $O(k) = O(N)$

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

Let the packing factor $k = 2^s \geq 2$ for some positive integer s and $n_i = 2^{\ell-i+1}$ for $i \in [\ell]$. Let f be a ℓ-variate multilinear polynomial, and $\boldsymbol{x} \in \mathbb{F}^n$ is the evaluations of f on the hypercube $\{0,1\}^\ell$. Let N be the number of servers.

Inputs: Each server $\mathsf{S}_1, ..., \mathsf{S}_N$ holds packed secret shares of vectors $\boldsymbol{x_j} = \boldsymbol{x_j^{(1)}} = \{x_{(j-1)k+i}\}_{i\in[k]}$, denoted as $[\![\boldsymbol{x_j^{(1)}}]\!]$, for $j \in [\frac{n_1}{k}]$,

Protocol:

1. In the i-th round, where $1 \leq i \leq \ell - s$,
   (a) Each server locally computes and outputs $[\![\boldsymbol{a_1^{(i)}}]\!] = \sum_{j=1}^{\frac{n_i}{2k}} [\![\boldsymbol{x_j^{(i)}}]\!], [\![\boldsymbol{a_2^{(i)}}]\!] = \sum_{j=\frac{n_i}{2k}+1}^{\frac{n_i}{k}} [\![\boldsymbol{x_j^{(i)}}]\!]$.
   (b) The servers receive a random challenge $r_i \in \mathbb{F}$ from verifier, then locally compute $\{[\![\boldsymbol{x_j^{(i+1)}}]\!]\}_{j\in[\frac{n_{i+1}}{k}]}$ by Equation 3.
2. The servers take $[\![\boldsymbol{x^{(l-s+1)}}]\!]$ as input and invoke $\mathcal{F}_{\mathsf{PSSToSS}}$ to get $\langle x_1^{(l-s+1)}\rangle, ..., \langle x_k^{(l-s+1)}\rangle$.
3. In the i-th round, where $\ell - s + 1 \leq i < \ell$,
   (a) Each server locally computes and outputs $\langle a_1^{(i)}\rangle = \sum_{j=1}^{\frac{n_i}{2}} \langle x_j^{(i)}\rangle, \langle a_2^{(i)}\rangle = \sum_{j=\frac{n_i}{2}+1}^{n_i} \langle x_j^{(i)}\rangle$.
   (b) The servers receive a random challenge $r_i \in \mathbb{F}$ from verifier, then locally compute $\{\langle x_j^{(i+1)}\rangle\}_{j\in[n_{i+1}]}$ by $\langle x_j^{(i+1)}\rangle = (1 - r_i) \cdot \langle x_j^{(i)}\rangle + r_i \cdot \langle x_{j+\frac{n_i}{2}}^{(i)}\rangle$
4. The servers output $\{[\![\boldsymbol{a_1^{(i)}}]\!], [\![\boldsymbol{a_2^{(i)}}]\!]\}_{i\in[1,\ell-s]}$ and $\{\langle a_1^{(i)}\rangle, \langle a_2^{(i)}\rangle\}_{i\in[\ell-s+1,\ell]}$.

Figure 1: Collaborative sumcheck $\Pi_{\mathsf{dSumcheck}}$ in the $\mathcal{F}_{\mathsf{PSSToSS}}$-hybrid world.

field operations. The cost of Π_{PSSToSS} is negligible. 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 $O(\frac{n}{N})$. Due to one-round $\mathcal{F}_{\mathsf{PSSToSS}}$, the round complexity is $O(1)$ and the total communication complexity is $O(N)$ field elements.

Extending to high-degree case. The protocol can be extended for a product of two multilinear polynomials, say, $H = \sum_{\boldsymbol{x}\in\{0,1\}^\ell} f(\boldsymbol{x})$, where $f(\boldsymbol{x}) = c_1(\boldsymbol{x}) \cdot c_2(\boldsymbol{x})$. To illustrate this, we show how to compute the share of $f_i(0)$ and $f_i(1)$ in the i-th round as an example. The idea is that each server first computes the (packed) shares of the $2^{\ell-i+1}$ entries in the i-th row of the bookkeeping table for c_1 and c_2 separately according to $\Pi_{\mathsf{dSumcheck}}$. Subsequently, the share of $f_i(0)$ and $f_i(1)$ can be computed by multiplying the corresponding shares in the same position of the two tables and summing them up. The overall time complexity for each server remains $O\left(\frac{n}{N}\right)$. Finally, a share of $f_i(2)$ will be computed by each server similarly using the dynamic programming technique introduced in [37,43]. These shares of the three points are sufficient for fixing a degree-2 polynomial $f_i(x)$. Note that in this case, since the packed secret shares are multiplied, the degree of the resulting share will be doubled. Therefore, one round of standard degree reduction is needed for recovering the degree.

In a similar vein, the above method can be extended to any degree-d polynomials that can be computed by multilinear polynomials. This result is important, as in the proof generation of the collaborative zk-SNARKs, the multivariate polynomials we encounter, e.g., the polynomial in zerocheck, are not necessarily multilinear. Looking ahead, in scalable collaborative zk-SNARKs studied in this work, the degree d is at most 4, therefore the round complexity remains constant throughout the proof generation.

4.2 Collaborative productcheck

Given an ℓ-variate polynomial f, Quark [34] provides a protocol for productcheck on f. A vital procedure involves the prover constructing a $(\ell+1)$-variate polynomial v such that $v(0, \boldsymbol{x}) = f(\boldsymbol{x})$ and $v(1, \boldsymbol{x}) = v(\boldsymbol{x}, 0) \cdot v(\boldsymbol{x}, 1)$ for any $\boldsymbol{x} \in \{0,1\}^\ell$. The prover then convinces the verifier about the correctness of v through zerocheck, and evaluates v at $(1, \cdots, 1, 0)$ to show that $v(1, \cdots, 1, 0) = H$ with the aid of a multivariate polynomial commitment. The key step is to compute the polynomial v, more precisely, the evaluations of v over $\{0,1\}^{\ell+1}$.

In our collaborative setting, each server initially receives packed secret shares of f's evaluations on the hypercube $\{0,1\}^\ell$, denoted by $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$, where $n = 2^\ell$ and k is the packing factor. The above step translates to

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

It 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 $(\textsc{Product}, m, [\![\boldsymbol{x_1}]\!], \ldots, [\![\boldsymbol{x_m}]\!])$ from $\mathsf{S}_1, \ldots, \mathsf{S}_N$, where $m = \frac{n}{k}$ is the number of packed shares:

- For $i \in [m]$, reconstruct $(x_{(i-1)k+1}, ..., x_{ik})$ from $[\![\boldsymbol{x_i}]\!]$.
- Set $\boldsymbol{x} := (x_1, \ldots, x_{m \cdot k})$ and run $\textsc{Prod-Tree}(\boldsymbol{x})$ to obtain $\boldsymbol{v}' \in \mathbb{F}^{mk-1}$. Set $\boldsymbol{v} := (\boldsymbol{v}', 0) \in \mathbb{F}^{mk}$.
- For $i \in [m]$:
  - set $\boldsymbol{v_i} := (v_{(i-1)k+1}, ..., v_{ik})$
  - Receive a set of shares $\{u_{i,j}\}_{j \in \mathcal{C}orr}$ from $\mathcal{S}$.
  - Sample a random packed shares $[\![\boldsymbol{v_i}]\!]$ of $\boldsymbol{v_i}$, such that the shares of the corrupted parties are identical to those received from $\mathcal{S}$, i.e., $\{u_{i,j}\}_{j \in \mathcal{C}orr}$.
- Distribute the shares $[\![\boldsymbol{v_1}]\!], \ldots, [\![\boldsymbol{v_m}]\!]$ to all servers.

Figure 2: The ideal functionality $\mathcal{F}_{\mathsf{ProdTree}}$.

letting the servers compute the packed secret shares of v's evaluations on $\{0,1\}^{\ell+1}$. Since $v(0, \boldsymbol{x}) = f(\boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^{\ell}$, it can be noticed that these shares at hand are already the first $\frac{n}{k}$ packed secret shares of the evaluations of v, i.e., the packed shares of $v(0, \boldsymbol{x})$. Therefore, the remaining task is to compute the packed shares of $v(1, \boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^{\ell}$. However, unlike the sumcheck protocol, this computation does not exhibit a well-formed SIMD structure, and it remains unclear how to figure out these elements in a way that the overhead can be evenly distributed.

A novel view for v's evaluations. Our important observation is that the formulation of polynomial v can be seen as building a depth-ℓ perfect binary tree, where the $n = 2^{\ell}$ leaves are actually the evaluations of f over the hypercube $\{0,1\}^{\ell}$ and the value in each node is the product of its two children. Therefore, the evaluations of $v(1, \boldsymbol{x})$ equal the $n-1$ internal nodes inside the tree, arranged in a level-order manner. The only exception is the point $(1, 1, \ldots, 1)$, which equals 0 directly. Thus, the key step involves computing such an *input-n product tree*, which can be completed according to Algorithm 1. This tree-like formulation provides a new perspective to compute v in a distributed manner: we split the n leaves into N subtrees, and each server is responsible for computing $\frac{n}{N} - 1$ nodes inside one subtree; After that, a server S_1 is selected to integrate the roots of the N subtrees and do the remaining computation for the last $N-1$ nodes, which is to compute another input-N product tree in $O(N)$ time. Considering $N \ll n$, in this process, each server bears only $O(\frac{n}{N})$ computation and memory overhead, and the round complexity remains constant.

Algorithm 1 Computing an input-n Product Tree

```
// Assume n = 2^ℓ for some positive integer ℓ.
function PROD-TREE(x_0 := {x_{0,j}}_{j∈[n]} ∈ F^n)
    for i = 1 to ℓ do
        x_{i,j} ← x_{i-1,2j-1} · x_{i-1,2j}, ∀j ∈ [2^{ℓ-i}]
        x_i := {x_{i,j}}_{j∈[2^{ℓ-i}]}  // x_ℓ := {r}, where r is the root.
    end for
    return z ← (x_1, ..., x_ℓ) ∈ F^{n-1}
end function
```

Handling packed shares. Although the above idea seems promising, a challenge arises: in a (packed) secret sharing context, the above process involves servers performing a series of $O(\ell)$ multiplications on the (packed) shares, which leads to logarithmic rounds of degree reduction, which is not desirable in our setting. To address this, we refer to the idea of computing unbounded multiplications within a constant-round protocol from [3]. Specifically, the servers first multiply the packed shares of the leaves by a group of carefully prepared masks, and then reveal the $\frac{i}{N}$-th part of "masked" leaves to S_i. As a result, each S_i receives the leaves of its subtree. This facilitates the servers to compute the "masked" product tree within constant-round communication. Finally, each server distributes its results using packed secret sharing again, and the servers multiply the received packed shares by another group of prepared elements to get the unmasked product tree.

Our protocol. For clarity, we provide the ideal functionality and protocol for computing the product tree in Figure

Protocol $\Pi_{\mathsf{dProdTree}}$

Let $N = 2^s$ be the number of servers for some positive integer s and $n = 2^\ell \geq N^2$. Let f be an ℓ-variate polynomial, and $\boldsymbol{x} \in \mathbb{F}^n$ is the evaluations of f on the hypercube $\{0,1\}^\ell$.

Inputs: Each server $\mathsf{S}_1, ..., \mathsf{S}_N$ holds packed secret shares of vectors $\boldsymbol{x_j} = \{x_{(j-1)k+i}\}_{i\in[k]}$, denoted as $[\![\boldsymbol{x_j}]\!]$, for $j \in [\frac{n}{k}]$, where k is the packing factor.

Preprocessing of masks:

1. Each server S_i receives packed secret shares of masks $\{[\![\boldsymbol{m_j}]\!]\}_{j\in[\frac{n}{N}]}$ and $\{[\![\boldsymbol{u_j}]\!]\}_{j\in[\frac{n}{N}]}$ from $\mathcal{F}_{\mathsf{Rand\text{-}ProdTree}}$.

Protocol:

1. Each server S_i computes $[\![\boldsymbol{y_j}]\!] := [\![\boldsymbol{x_j}]\!] \cdot [\![\boldsymbol{m_j}]\!]$ for $j \in [\frac{n}{k}]$, and send the $\frac{n}{k}$ shares to S_1.
2. S_1 opens $\boldsymbol{y}$, and send $\boldsymbol{y_i} := (y_{\frac{n(i-1)}{N}+1}, ..., y_{\frac{in}{N}})$ to server S_i, for $i \in [N]$.
3. Each server S_i invokes $\textsc{Prod-Tree}(\boldsymbol{y_i})$ to get $\frac{n}{N} - 1$ elements $\boldsymbol{z_i}$. S_i sends the last $k-1$ elements of $\boldsymbol{z_i}$ to S_1. The last one element r_i is the root of the subtree.
4. S_1 collects the $N(k-1)$ elements as $\boldsymbol{z_0}$ and invokes $\textsc{Prod-Tree}(\{r_i\}_{i\in[N]})$ to get $N-1$ elements. It expands $\boldsymbol{z_0}$ with the $(N-1)$ elements and 0.
5. Each server S_i packed secret shares $\boldsymbol{z_i}$ to others, except for the elements sent to S_1. S_1 additionally shares $\boldsymbol{z_0}$. As a result, each server holds $[\![\boldsymbol{z_j}]\!]_{j\in[\frac{n}{k}]}$.
6. The servers take $[\![\boldsymbol{z_j}]\!]$ and $[\![\boldsymbol{u_j}]\!]$ as input and invoke $\mathcal{F}_{\mathsf{PSSMult}}$ to output $[\![\boldsymbol{v_j}]\!] := [\![\boldsymbol{z_j}]\!] \cdot [\![\boldsymbol{u_j}]\!]$, for $j \in [\frac{n}{k}]$.

Figure 3: Collaborative product tree $\Pi_{\mathsf{dProdTree}}$.

2 and Figure 3, respectively. Note that here we assume a peer-to-peer network. To complete packed sharing multiplication, we recall a standard functionality $\mathcal{F}_{\mathsf{PSSMult}}$, which is described in Appendix A.2. The output of the protocol, namely $\{[\![\boldsymbol{v_j}]\!]\}_{j\in[\frac{n}{k}]}$, is exactly the packed secret shares of $v(1, \boldsymbol{x})$'s n evaluations on the hypercube. In the protocol, the servers first receive masks in the pre-processing phase. The masks $\boldsymbol{m}$ are carefully designed to ensure that no server can learn the original elements $\boldsymbol{x}$, and $\boldsymbol{u}$ promises that the unmasked result is calculated properly. These shares can be prepared by servers invoking the functionality $\mathcal{F}_{\mathsf{Rand\text{-}ProdTree}}$ described in Appendix A.4 offline when they are idle, or, in our setting, by a client's delivery directly. Finally, we have the following theorem:

Theorem 2. *The protocol $\Pi_{\mathsf{dProdTree}}$ depicted in Figure 3 is a secure MPC protocol that computes $\mathcal{F}_{\mathsf{ProdTree}}$ in the $\{\mathcal{F}_{\mathsf{PSSMult}}, \mathcal{F}_{\mathsf{Rand\text{-}ProdTree}}\}$-hybrid world against a semi-honest adversary who corrupts at most t servers.*

Proof sketch. For $x_i \in \boldsymbol{x}$, the mask m_i is prepared in the form $r_i r_{i+1}^{-1}$ where $\{r_i\}$ are uniform randomness to hide $\{x_i\}$. Taking the root of an input-n product tree as an example: the root $z_{n-1} = \prod_i^n m_i x_i = r_1 r_{n+1}^{-1} \prod_i^n x_i$. The corresponding unmask u_{n-1} is prepared as $r_1^{-1} r_{n+1}$. This ensures the unmasked output $v_{n-1} = z_{n-1} u_{n-1} = \prod_i^n x_i$ is computed properly. Similarly, all unmasks $\{[\![\boldsymbol{u_j}]\!]\}$ are designed to ensure that the randomness applied to $\{[\![\boldsymbol{z_j}]\!]\}$ is removed. The correctness of the remaining parts follows directly from the calculation of the product tree. For security, it is easy to conclude from the protocol that the servers only perform local computations, except for invoking $\mathcal{F}_{\mathsf{Rand\text{-}ProdTree}}$, $\mathcal{F}_{\mathsf{PSSMult}}$ and distributing the shares to others; therefore, the corrupted servers cannot learn others' private input. This ensures the security. □

Distributing communication. In the packed sharing multiplication $\mathcal{F}_{\mathsf{PSSMult}}$ and Step 1 of $\mathcal{F}_{\mathsf{ProdTree}}$, a specified server S_1 is responsible for receiving data from all other servers. When the circuit size or the number of servers is big, S_1 requires a large bandwidth to receive these elements, forming an efficiency bottleneck. To address this issue, we decided to distribute S_1's job to more servers. Taking $\mathcal{F}_{\mathsf{PSSMult}}$ as an example, since there are $O(\frac{n}{k})$ different shares needing degree reduction and these jobs are independent of each other, we can assign different shares to different servers, which amortizes the total $O(n)$ communication cost by a factor of $O(N)$. The step in $\mathcal{F}_{\mathsf{ProdTree}}$ can be optimized similarly. In practice, these jobs are assigned to those servers with better network conditions. These optimizations are feasible as we have assumed a peer-to-peer network.

Efficiency. The protocol depicted in Figure 3 is both time-efficient and space-efficient. The total proving work of N servers is $O(n)$. Each server S_i undertakes an input-$\frac{n}{N}$ subtree, while S_1 additionally undertakes an input-N tree. Although we have a server S_1 who undertakes more work than others, this job can be assigned to any server since it is cheap and does not have a distinguished complexity. Therefore, the time complexity and space

Protocol Π_{dMVPC}

Let the packing factor $k = 2^s \geq 2$ for some positive integer s and $n_i = 2^{\ell-i}$ for $i \in [0, \ell]$. Let f be ℓ-variate a multilinear polynomial, and $\boldsymbol{x} \in \mathbb{F}^n$ is the evaluations of f on the hypercube $\{0,1\}^\ell$. Let N be the number of servers.

Inputs: Each server $\mathsf{S}_1, ..., \mathsf{S}_N$ holds packed secret shares of vectors $\boldsymbol{x_j^{(0)}} = \{x_{(j-1)k+i}\}_{i\in[k]}$, denoted as $[\![\boldsymbol{x_j^{(0)}}]\!]$, for $j \in [\frac{n_0}{k}]$.

Procedure dMVPC.Setup**:** This procedure is executed by a trusted setup:

1. Sample $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}^\ell$ as the trapdoor.
2. For $i \in [0, \ell]$, 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 $\beta_{b_j}(s_j) = (1-s_j)(1-b_j) + s_j b_j$. This results in n_i group elements.
3. For $i \in [0, \ell - s]$, pack the n_i group elements from the Step 2 into k-size vectors $\mathbf{P_1^{(i)}}, ..., \mathbf{P}_{\frac{n_i}{k}}^{(i)}$. Each server receives $\{[\![\mathbf{P_j^{(i)}}]\!]\}_{j\in[\frac{n_i}{k}]}$ as pp_i.
4. For $i \in [\ell - s + 1, \ell]$, denote the n_i group elements from the Step 2 as $\{\mathsf{P}_j^{(i)}\}_{j\in[n_i]}$. Each server receives $\{\langle \mathsf{P}_j^{(i)} \rangle\}_{j\in[n_i]}$ as pp_i.

Procedure dMVPC.Commit**:**

1. Each server parse pp_0 as $\{[\![\mathbf{P_j^{(0)}}]\!]\}_{j\in[\frac{n_0}{k}]}$, and the servers send $(\textsc{Mult}, \frac{n_0}{k}, \{[\![\boldsymbol{x_j^{(0)}}]\!]\}_{j\in[\frac{n_0}{k}]}, \{[\![\mathbf{P_j^{(0)}}]\!]\}_{j\in[\frac{n_0}{k}]})$ to $\mathcal{F}_{\mathsf{dMSM}}$, which returns $\langle \mathsf{com}_f \rangle$ to the servers.

Procedure dMVPC.Open**:** Let $\boldsymbol{u}$ be the evaluation point.

1. In the i-th round, where $1 \leq i \leq \ell - s$,
   (a) Each server locally computes $[\![\boldsymbol{q_j^{(i)}}]\!] = [\![\boldsymbol{x}_{\boldsymbol{j+\frac{n_i}{k}}}^{\boldsymbol{(i-1)}}]\!] - [\![\boldsymbol{x_j^{(i-1)}}]\!]$ and $[\![\boldsymbol{x_j^{(i)}}]\!] = (1-u_i) \cdot [\![\boldsymbol{x_j^{(i-1)}}]\!] + u_i \cdot [\![\boldsymbol{x}_{\boldsymbol{j+\frac{n_i}{k}}}^{\boldsymbol{(i-1)}}]\!]$, for $j \in [\frac{n_i}{k}]$.
   (b) Each server parses pp_i as $\{[\![\mathbf{P_j^{(i)}}]\!]\}_{j\in[\frac{n_i}{k}]}$, and the servers send $(\textsc{Mult}, \frac{n_i}{k}, \{[\![\boldsymbol{q_j^{(i)}}]\!]\}_{j\in[\frac{n_i}{k}]}, \{[\![\mathbf{P_j^{(i)}}]\!]\}_{j\in[\frac{n_i}{k}]})$ to $\mathcal{F}_{\mathsf{dMSM}}$, which returns $\langle \pi_i \rangle$ to the servers.
2. The servers take $[\![\boldsymbol{x^{(l-s)}}]\!]$ as input and invoke $\mathcal{F}_{\mathsf{PSSToSS}}$ to get $\langle x_1^{(l-s)} \rangle, ..., \langle x_k^{(l-s)} \rangle$.
3. In the i-th round, where $\ell - s + 1 \leq i \leq \ell$,
   (a) Each server locally computes $\langle q_j^{(i)} \rangle = \langle x_{j+\frac{n_i}{k}}^{(i-1)} \rangle - \langle x_j^{(i-1)} \rangle$ and $\langle x_j^{(i)} \rangle = (1-u_i) \cdot \langle x_j^{(i-1)} \rangle + u_i \cdot \langle x_{j+\frac{n_i}{k}}^{(i-1)} \rangle$, for $j \in [n_i]$.
   (b) Each server parses pp_i as $\{\langle \mathsf{P}_j^{(i)} \rangle\}_{j\in[n_i]}$, and locally computes $\prod_{j\in[n_i]} \langle \mathsf{P}_j^{(i)} \rangle^{\langle q_j^{(i)} \rangle}$, which returns $\langle \pi_i \rangle$ as a result.
4. The servers output $\langle \pi \rangle = (\langle \pi_1 \rangle, ..., \langle \pi_\ell \rangle)$.

Figure 4: Collaborative multilinear polynomial commitment in the $\{\mathcal{F}_{\mathsf{dMSM}}, \mathcal{F}_{\mathsf{PSSToSS}}\}$-hybrid world.

complexity for each server S_i is consistently $O(\frac{n}{N})$. The round complexity is $O(1)$, and the total communication complexity is $O(n)$ field elements which can be shared among servers.

Finally, in a productcheck, the servers also need to obtain packed shares of $v(\boldsymbol{x}, 0)$ and $v(\boldsymbol{x}, 1)$. Since by Step 5 of the protocol the servers have computed the evaluations of the entire product tree, these shares can be acquired by adjusting the pattern in which each server distributes its elements and unmasking the results accordingly.

4.3 Collaborative multilinear polynomial commitment

The zk-SNARKs we study often encode the witness as polynomials and commit to them with multivariate polynomial commitments [29]. Suppose $\boldsymbol{x}$ is a vector of $n = 2^\ell$ elements, corresponding to the witness to encode. This vector can be effectively represented by a function $V : \{0,1\}^\ell \to \mathbb{F}$. The prover uses mvPC to commit to an ℓ-variate polynomial f, which is a multilinear extension of V. In our setting, we assume each server initially holds $\frac{n}{k}$ packed shares, represented as $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$, where each $\boldsymbol{x_j}$ is a size-k vector and k is the packing factor we choose. The primary challenge is two-fold: (i) generating commitment for f, and (ii) generating proof for evaluations.

Generating commitment. Leveraging Equation 1, 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 be computed by inputs $x_1, \ldots, x_n \in \mathbb{F}$ and the corresponding group elements in an MSM manner. However, in the collaborative setting, a problem is that each server possesses only the packed shares of $\boldsymbol{x}$, rather than the scalars. To tackle this, we recall the technique of distributed MSM introduced by [16], which enables servers to compute MSM in a collaborative fashion. To facilitate this, we propose to prepare $g^{\prod_{i=1}^{\ell}\beta_{b_i}(s_i)}$ for all possible $\boldsymbol{b} \in \{0,1\}^\ell$ as parameters, also in a packed secret sharing form. With these parameters and the packed shares of $\boldsymbol{x}$, the servers can employ the technique of distributed MSM to collaboratively compute the commitment. For clarity and completeness, we borrow the functionality $\mathcal{F}_{\mathsf{dMSM}}$ and protocol Π_{dMSM} from [16] and put them in Appendix A.5 with minor modifications. This protocol is essentially time-efficient and space-efficient, as each server averagely bears an overhead of $O\left(\frac{n}{k}\right)$ group exponentiation and a space cost of $O\left(\frac{n}{k}\right)$.

Generating opening proof. During mvPC.Open, in order to generate a proof for the claim $z = f(\boldsymbol{u})$, the prover takes the following two steps:

- First, it undertakes ℓ polynomial divisions to obtain a sequence of quotient polynomials $\{Q_i(x_{i+1}, \ldots, x_\ell)\}_{i\in[\ell]}$ and remainder polynomials $\{R_i(x_{i+1}, \ldots, x_\ell)\}_{i\in[\ell]}$. Let $R_0 := f$ denote the original polynomial. In the i-th division, the operation is performed on the remainder polynomial of the last round, R_{i-1}, with respect to the divisor $(x_i - u_i)$. Formally, for $i \in [\ell]$,

$$R_{i-1}(x_i, x_{i+1}, \ldots, x_\ell) = Q_i(x_{i+1}, \ldots, x_\ell)(x_i - u_i) + R_i(x_{i+1}, \ldots, x_\ell) \tag{4}$$

- After obtaining these polynomials, it computes the proof as $\{g^{Q_i(\boldsymbol{s})}\}_{i\in[\ell]}$. It is easy to see that, in the collaborative setting, given the packed shares of Q_i's evaluations on corresponding hypercubes, this computation can be carried out by the servers in a manner similar to the commitment generation phase described earlier.

In [16], a similar difficulty is encountered when dealing with KZG polynomial commitment [23] for univariate polynomials, where the polynomial divisions are done by the use of distributed FFT. However, a major drawback of applying this technique is its reliance on a powerful server, which imposes increased communication costs and potential memory bottleneck. This is contrary to our goal of establishing a scalable collaborative zk-SNARK where time and space complexity is shared equally among all servers.

To tackle polynomial divisions, a new method leveraging the algebraic property of multilinear polynomials is proposed. Note that in Equation 4, both the quotient and remainder polynomials Q_i, R_i are multilinear. Let $(x_{i+1}, \ldots, x_\ell)$ take values $\boldsymbol{b} \in \{0,1\}^{\ell-i}$, and x_i take values 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,

$$Q_i(\boldsymbol{b}) = R_{i-1}(1, \boldsymbol{b}) - R_{i-1}(0, \boldsymbol{b}), \quad R_i(\boldsymbol{b}) = (1-u_i) \cdot R_{i-1}(0, \boldsymbol{b}) + u_i \cdot R_{i-1}(1, \boldsymbol{b}) \tag{5}$$

Our observation is, similar to Equation 2, the above formula also has a SIMD property that can be exploited. Initially, each server holds $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$. After receiving a specific point $\boldsymbol{u} \in \mathbb{F}^\ell$, the packed shares of the first quotient polynomial Q_1's evaluations on the hypercube $\{0,1\}^{\ell-1}$ can be computed locally by each server. This involves subtracting the first half of the shares $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$ from the second half. The packed shares of evaluations for the first remainder polynomial R_1 can also be determined by linear combinations of the packed shares within $\{[\![\boldsymbol{x_j}]\!]\}_{j\in[\frac{n}{k}]}$, analogous to Equation 3. This computation is recursively done for each of the quotient polynomials similar to the collaborative sumcheck protocol. In the second step, after obtaining packed shares of the evaluations of Q_i on the hypercube $\{0,1\}^{\ell-i}$, the servers collaborate to compute $g^{Q_i(s_{i+1}, \ldots, s_\ell)}$ using Π_{dMSM} protocol. Note that the above computation can also get stuck in a given round. In this case, we allocate the remaining work, which results in a small computational overhead, to each server by using the $\mathcal{F}_{\mathsf{PSSToSS}}$ technique again, similar to the collaborative sumcheck protocol.

Our protocol. Combining the above discussions, we provide the protocol of our collaborative multilinear polynomial commitment Π_{dMVPC} as detailed in Figure 4. Formally, we have the following theorem:

Theorem 3. *The protocols* dMVPC.Commit, dMVPC.Open *depicted in Figure 4 are secure MPC protocols that compute* mvPC.Commit, mvPC.Open *in the* $\{\mathcal{F}_{\mathsf{dMSM}}, \mathcal{F}_{\mathsf{PSSToSS}}\}$*-hybrid world against a semi-honest adversary who corrupts at most* t *servers.*

Proof sketch. The correctness is straightforward. Except for invoking $\mathcal{F}_{\mathsf{dMSM}}$ and $\mathcal{F}_{\mathsf{PSSToSS}}$, the servers only perform local computations during dMVPC.Commit and dMVPC.Open. In other words, no corrupted parties have the chance to learn other parties' private input. Therefore, the security is guaranteed. □

Batching distributed MSMs. In the dMVPC.Open protocol, there are $O(\ell) = O(\log n)$ invocations of $\mathcal{F}_{\mathsf{dMSM}}$, leading to non-constant round complexity. We provide an important optimization that reduces the complexity to one round. We found that the distributed MSMs are independent of each other, so that they can be batch together. More precisely, in Step 1 of protocol dMVPC.Open, each server computes only the entries needed for the MSM. At the end of the protocol, $O(\ell)$ distributed MSMs are executed together: each server executes its own part of $O(\ell)$ Π_{dMSM} and sends $O(\ell)$ elements to S_1 of Π_{dMSM} in one go. Later, S_1 may computes the remaining part of the $O(\ell)$ Π_{dMSM} in parallel, further speeding up the computation. The reduced round complexity of dMVPC.Open is $O(1)$.

Efficiency. The protocol depicted in Figure 4 is both time-efficient and space-efficient. In dMVPC.Commit, the servers invoke Π_{dMSM} to compute the commitment where each server performs $O(\frac{n}{N})$ group exponentiation, has a space complexity of $O(\frac{n}{N})$, and communicates $O(N)$ group elements in total. In dMVPC.Open, the total proving work of N servers is $O(n)$. The computation overhead for each server S_i is $O(\frac{n}{N})$. The space complexity for each server is consistently $O(\frac{n}{N})$. The round complexity is $O(1)$ because of one round $\mathcal{F}_{\mathsf{PSSToSS}}$ and the batched $\mathcal{F}_{\mathsf{dMSM}}$ technique. The total communication complexity among servers is $O(N \log n)$ group elements due to the distributed MSMs.

5 Instantiating Scalable Collaborative Proofs

We aim to instantiate scalable collaborative zk-SNARKs for Libra [43] and HyperPlonk [7] as concrete examples.

HyperPlonk. It is a novel zk-SNARK with $O(n)$ prover time for *general circuits* of size n. Due to its expressiveness, HyperPlonk is applied for certain program executions with tremendously large circuits, like zkEVM [4]. The prover of HyperPlonk mainly consists of zerocheck, productcheck, and several multilinear polynomial evaluations.

Libra. It is a GKR-based zk-SNARK designed by combining the famous GKR [19] protocol with a multilinear polynomial commitment. Libra has $O(n)$ prover time, where n is the size of a depth-d layered circuit. For Libra, we mainly care about *data-parallel circuits*, where a circuit consists of many identical copies of sub-circuit. This is because GKR-based SNARKs have better concrete prover efficiency compared to HyperPlonk and are mainly optimized for data-parallel circuits for real-world applications [31, 42, 43]. Libra's prover mainly consists of $O(d)$ rounds of sumcheck and one-time multilinear polynomial evaluation.

For HyperPlonk, we can implement scalable collaborative proof for general circuits, while for Libra, we achieve this for data-parallel circuits. This is done by replacing prover's primitives with corresponding collaborative primitives. Since the collaborative primitives have evenly distributed the time complexity and space complexity among the servers, the proof generation of the corresponding collaborative zk-SNARKs inherits the same properties. Note that, both Libra and HyperPlonk are public-coin, so the verifier's messages can be removed by applying the Fiat-Shamir transformation [13]. In our collaborative setting, the transformation can be accomplished by having each server reconstruct the proof transcript and use it to query a random oracle to obtain the same random elements as the verifier's messages. Formally, we have the following theorems:

Theorem 4. *If* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is the Libra zk-SNARK for a data-parallel circuit* $\mathcal{C}$*. There exists a collaborative zk-SNARK* $(\mathsf{Setup}, \Pi_{\mathsf{Libra}}, \mathsf{Verify})$ *for* $\mathcal{C}$*, where* Π_{Libra} *is a secure MPC protocol that computes* Prove *in the* $\{\mathcal{F}_{\mathsf{PSSMult}}, \mathcal{F}_{\mathsf{dMSM}}, \mathcal{F}_{\mathsf{PSSToSS}}\}$*-hybrid world against a semi-honest adversary who corrupts at most* t *servers.*

Theorem 5. *If* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is the HyperPlonk zk-SNARK for a general circuit* $\mathcal{C}$*. There exists a collaborative zk-SNARK* $(\mathsf{Setup}, \Pi_{\mathsf{HyperPlonk}}, \mathsf{Verify})$ *for* $\mathcal{C}$*, where* $\Pi_{\mathsf{HyperPlonk}}$ *is a secure MPC protocol that computes* Prove *in the* $\{\mathcal{F}_{\mathsf{PSSMult}}, \mathcal{F}_{\mathsf{dMSM}}, \mathcal{F}_{\mathsf{PSSToSS}}, \mathcal{F}_{\mathsf{ProdTree}}\}$*-hybrid world against a semi-honest adversary who corrupts at most* t *servers.*

Proof sketch. We defer the concrete construction of Π_{Libra} and $\Pi_{\mathsf{HyperPlonk}}$ to Appendix B and C, respectively. The correctness follows the construction directly. Both Π_{Libra} and $\Pi_{\mathsf{HyperPlonk}}$ can be divided into two components: (i) local computations and invoking of the sub-protocols for multivariate polynomials primitives, which are already proven to be secure in the $\{\mathcal{F}_{\mathsf{PSSMult}}, \mathcal{F}_{\mathsf{dMSM}}, \mathcal{F}_{\mathsf{PSSToSS}}, \mathcal{F}_{\mathsf{ProdTree}}\}$-hybrid world; (ii) accomplishing the Fiat-Shamir transform collaboratively. In the latter component, the servers reconstruct the proof transcript and make queries to the random oracle; by the zero-knowledge property of the underlying zk-SNARK, the corrupted servers can learn nothing from the obtained proof transcript. This guarantees the security. □

Efficiency. Both Π_{Libra} and $\Pi_{\mathsf{HyperPlonk}}$ are time-efficient and space-efficient. As we discussed, the total proving work of both zk-SNARKs is $O(n)$. Due to the application of the collaborative primitives in Section 4, the time complexity and space complexity for each server S_i in Π_{Libra} and $\Pi_{\mathsf{HyperPlonk}}$ are both $O\left(\frac{n}{N}\right)$. The round complexity of Π_{Libra} and $\Pi_{\mathsf{HyperPlonk}}$ is $O(d)$ and $O(1)$, respectively. For both protocols, the total communication complexity among servers is $O(n)$ field elements due to the necessary degree reductions, and $O(N \log n)$ group elements due to the distributed MSMs. The communication complexity can be distributed among the servers.

6 Implementation and Evaluations

6.1 Evaluation setup

We build the codebase on top of the `mpc-net` library [27] for network communication and the `arkworks` library [1] for finite field and elliptic curve operations. Besides the implementation of our protocols, we also provide a prototype of monolithic Libra and HyperPlonk for comparison purposes. Overall, our implementation involves about 5000 lines of Rust code and other scripts. We provide our codebase anonymously, which is available at https://github.com/LBruyne/Scalable-Collaborative-zkSNARK.

Experiment setup. One of the merits for our framework is that there is no need for a powerful server to dominate the proof generation. To demonstrate this, our protocols are evaluated with up to 256 consumer-level machines of instance type `c7.large`, each with only 4 GB of RAM. For the PSS scheme we rely on, the packing factor is set as $k = \frac{N}{4}$, and the protocols are secure against at most $t = \frac{N}{4}$ corrupted servers. This is the same setting as in [16]. As discussed in Section 3, pre-processing is not evaluated, which is similarly handled in [16, 28].

Experiment design. We design the following experiments to evaluate our scalable collaborative zk-SNARKs:

1. Performance comparison between our protocols and corresponding monolithic zk-SNARKs. The running time of our protocols decreases linearly with the number of servers, while the maximum circuit that can be handled scales linearly.

2. Performance and economic analysis of our results and collaborative Plonk from [16]. The result highlights the benefits of eliminating the specified powerful server. It allows all servers to take only a proportional share of the total workload, so that proof generation is not limited by a single server's capability.

3. Performance under different network conditions. The efficiency maintains even under limited network conditions. Moreover, with more normal-equipped servers, we can achieve better efficiency. Therefore, our approaches are robust to apply in proof outsourcing where participants are diverse and from different locations.

Looking ahead, the results demonstrate that our protocols are efficient, scalable, and cost-effective, making them suitable for real-world proof outsourcing applications.

6.2 Comparison with local prover

We commence by comparing the performance of our implementation with the corresponding local prover of Libra and HyperPlonk. Since collaborative proofs are needed when outsourcing large-scale circuits that are infeasible for clients, we vary the total number of circuit gates from 2^{20} to 2^{28} to represent complex circuits in real-world applications. For data-parallel circuits, which are the targets of collaborative Libra, we choose a 64-copy circuit, each copy with a depth of 8, and alter the number of gates in each layer to simulate different circuit sizes. For local provers, we use a `c7.large` machine with 4 GB of RAM to act as a low-specification PC machine. Servers in the collaborative setup are linked through a 4 Gbps network.

Time-efficiency. In Figure 5, we present the performance evaluation of collaborative Libra and HyperPlonk with different server counts ranging from 16 to 128. Our protocols are time-efficient, both achieving a linear improvement with the number of servers. The efficiency gain is more significant for larger circuits. Specifically, for a circuit with 2^{24} gates and $N = 128$ servers, the running times of Libra and HyperPlonk are reduced by factors of $21\times$ and $24\times$, respectively. We note that this efficiency is lower than the theoretical $k = 32$ times improvement, mainly due to the additional degree reductions and communication overhead, which are absent in the local prover.

Space-efficiency. It can be noted from the figure that a local prover encounters a memory bottleneck when the circuit size is relatively large, while our protocols remain space-efficient. A collaborative proof with N servers can

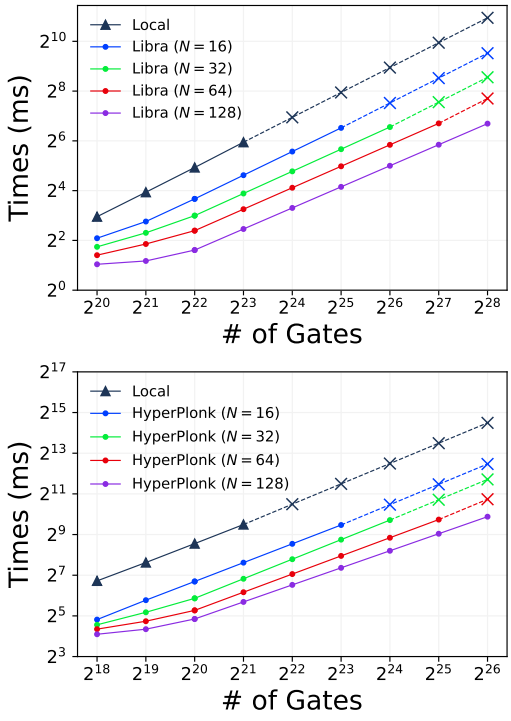

Figure 5: Performance comparison for collaborative proofs of our protocols and local provers. The slash lines indicate estimated data due to memory limitations.

scale to $k = \frac{N}{4}$ times larger circuits. Specifically, for HyperPlonk and Libra with 128 servers, the largest circuit they can handle is $32\times$ larger than that of a local prover. Moreover, the servers we use are equipped with only 4 GB of RAM, which is much lower than a typical server for production use. Therefore, it is expected that if more servers with higher memory capacity are available, the circuit size that can be handled will be further increased.

6.3 Comparison with [16]

We choose our collaborative HyperPlonk and the collaborative Plonk implementation [30] from [16] for comparison, as they both adopts general arithmetic circuits as the computation model. We note that monolithic HyperPlonk and Plonk essentially have *different* properties: the latter has more expensive prover time, while both proof systems have similar space complexity. Therefore, for a fair comparison, we fix the circuit size at 2^{20} and vary the number of servers from 16 to 128, measuring the running time and memory usage savings factor *with respect to a local prover* for both proposals. Collaborative Plonk is evaluated in a cluster comprising a powerful leader, which is a `g7.8xlarge` instance with 128 GB RAM, and other workers, which are `c7.large` instances with 4 GB RAM. We do not consider any multi-threading optimizations for both proposals. The evaluation results are summarized in Figure 6.

Memory usage. It is noted that the memory usage of the leader server in collaborative Plonk remains high, approximately equal to the overhead of a local prover, regardless of servers count. In contrast, the memory usage of a single server in HyperPlonk decreases linearly with the number of servers. This confirms the statement that the space complexity of the two proposals is $O(S_{\mathsf{P}})$ and $O\left(\frac{S_{\mathsf{P}}}{N}\right)$, respectively. Therefore, the memory usage of the leader server will become the main bottleneck for collaborative Plonk to deal with larger circuits. In contrast, our protocol can scale to larger-scale circuits by adding more servers, as the total memory usage is well distributed among all servers.

Running time. We also analyze the running time with 100 Mbps network and 4 Gbps network separately. It is noted that our protocol achieves better speedups. For collaborative Plonk, the speedup is limited without further assumptions on better network conditions and multi-threading optimization for the leader server. Moreover, with 100 Mbps bandwidth, the efficiency gain of HyperPlonk is minimally influenced, while Plonk cannot achieve any speedup. This is because the communication overhead of a server in our protocol is relatively small. Specifically,

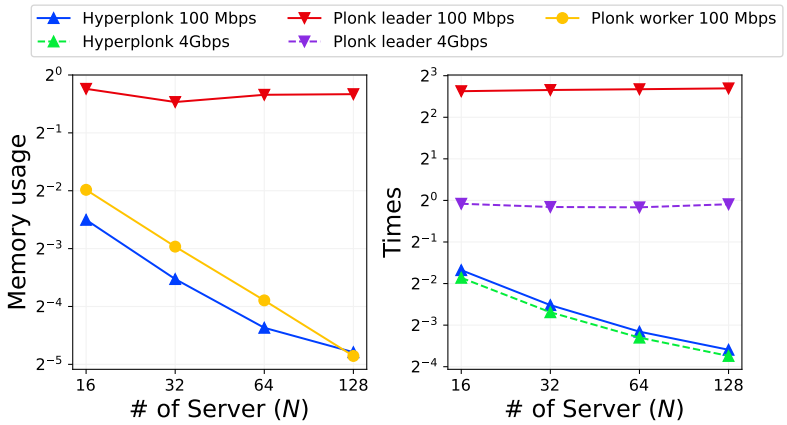

Figure 6: Performance comparison for our protocols and Plonk in [16]. The data are measured as $\frac{S}{S_\mathsf{P}}$ and $\frac{T}{T_\mathsf{P}}$, where S, T and $S_\mathsf{P}, T_\mathsf{P}$ are the memory usage and running time of the protocols and a local prover, respectively. Neither protocol consider multi-threading optimizations.

for a 128-server case, the communication costs are 39 MB and 22 GB for a single server of HyperPlonk and the leader of Plonk, respectively.

Financial cost. We complement with a financial cost calculation for the two schemes, referencing Google Cloud's pricing for spot instances and network services[1]. Concretely, it takes 28 seconds and 277 seconds for 128-servers HyperPlonk and Plonk to generate a proof, with overall communication costs of 128 servers being 2.43 GB and 22 GB, respectively. Considering the low-memory instances we use and the high-capacity server required by Plonk, this translates to an overall cost of $0.21 and $2, respectively, where the network fee dominates. Besides faster proof generation, the substantial savings are due to two factors: (i) our protocols only require lower-spec machines, avoiding the expensive high-capacity server; (ii) The lower communication cost leads to less network fees.

6.4 Performance under different networks

Finally, we fix the circuit size at 2^{24} and consider three different network settings: (i) Local network with 10 Gbps bandwidth and 0.1 ms latency, (ii) LAN network with 1 Gbps bandwidth and 1 ms latency, and (iii) WAN network with 100 Mbps bandwidth and 50 ms latency. We vary the number of servers from 32 to 256 and measure the communication cost for a single server and the corresponding speedup with respect to a local prover. The evaluation results are summarized in Table 2. A decrease in bandwidth leads to fewer efficiency gains, as the communication overhead takes longer time. This efficiency loss is less obvious for HyperPlonk, and HyperPlonk still achieves a $20\times$ speedup with 128 servers in the WAN network. The reasons are: (i) the concrete computation time of HyperPlonk is larger than that of Libra; (ii) collaborative HyperPlonk is constant-round, so its performance is less affected when switching to a slower network. Moreover, it is observed that the efficiency loss can be mitigated by adequately adding servers, as in our protocols the communication overhead will be better distributed if more servers are available.

7 Discussion

Achieving malicious security. In this work, we only consider a semi-honest adversary. However, we note that the zk-SNARK proofs output by our collaborative zk-SNARK framework is still *sound* even when all servers are corrupted by a malicious adversary. Furthermore, we make a conjecture, which is similar to [16], that our semi-honest protocols are secure against malicious corruptions up to linear attacks [18], that is, all the malicious parties can do is to cause some additive errors into the output of the protocols. We conjecture that our semi-honest protocols can be compiled to be malicious secure by augmenting with some lightweight verification protocols to detect the potential malicious behaviors, as in [17, 18, 20].

[1]Pricing at $0.0042 per GB hour for RAM of custom N1 machines, and network costs at $0.08 per GB.

Protocol	#Servers	Comm. (MB)	Speedup (×)		
			Local	LAN	WAN
Libra	32	1440	5.6	4.5	1.5
	64	747	10.4	8.5	2.9
	128	393	15.6	12.7	4.4
	256	250	18.6	13.7	3.7
HyperPlonk	32	994	6.9	6.8	6.3
	64	760	13.0	12.8	11.4
	128	397	22.9	22.6	20.3
	256	251	39.8	39.3	35.0

Table 2: Performance comparison under different networks. Communication cost is measured for a single server. Speedup is measured as $\frac{T_{\mathsf{P}}}{T}$, where T and T_{P} are the running time of the protocols and a local prover, respectively.

Future works. We do not achieve scalable collaborative proof for Libra with general circuits, mainly because the linear-time algorithm introduced by [43, Section 3.3] is difficult to distribute when the circuit has an arbitrary form. It is interesting to explore how to evenly distribute the prover of Libra with respect to general circuits. Another future work will be instantiating a scalable collaborative proof for zk-SNARKs where the relation $\mathcal{R}$ is represented in rank-1 constraint systems (R1CS). A representative is Spartan and its variants [33,34], where the primitives they used can also be replaced by collaborative tools studied in this work. Finally, it would be meaningful to explore realizing malicious security for the protocols and applying them to real-world applications.

Acknowledgement

We thank Alex Ozdemir for his helpful discussion and comments on this work. We thank Guru-Vamsi Policharla for his helpful advice regarding this work.

References

[1] arkworks contributors, "`arkworks` zksnark ecosystem," 2022. [Online]. Available: https://arkworks.rs

[2] A. Arun, S. Setty, and J. Thaler, "Jolt: Snarks for virtual machines via lookups," in *EUROCRYPT 2024*.

[3] J. Bar-Ilan and D. Beaver, "Non-cryptographic fault-tolerant computing in constant number of rounds of interaction," in *PODC 1989*.

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

[5] R. Canetti, "Security and composition of multiparty cryptographic protocols," in *Journal of Cryptology*, 2000.

[6] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," Cryptology ePrint Archive, Report 2022/1355, 2022, https://eprint.iacr.org/2022/1355.

[7] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "HyperPlonk: Plonk with linear-time prover and high-degree custom gates," in *EUROCRYPT 2023*.

[8] A. Chiesa, R. Lehmkuhl, P. Mishra, and Y. Zhang, "Eos: Efficient private delegation of zkSNARK provers," in *USENIX Security 23*.

[9] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *CRYPTO 2007*.

[10] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO 2012*.

[11] S. Das, R. Fernando, I. Komargodski, E. Shi, and P. Soni, "Distributed-prover interactive proofs," in *TCC 2023*.

[12] P. Dayama, A. Patra, P. Paul, N. Singh, and D. Vinayagamurthy, "How to prove any NP statement jointly? Efficient distributed-prover zero-knowledge protocols," in *PoPETs 2022*.

[13] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO 1986*.

[14] M. K. Franklin and M. Yung, "Communication complexity of secure computation (extended abstract)," in *STOC 1992*.

[15] A. Gabizon, Z. J. Williamson, and O. 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.

[16] S. Garg, A. Goel, A. Jain, G.-V. Policharla, and S. Sekar, "zkSaaS: Zero-Knowledge SNARKs as a service," in *USENIX Security 23*.

[17] D. Genkin, Y. Ishai, and A. Polychroniadou, "Efficient multi-party computation: From passive to active security via secure SIMD circuits," in *CRYPTO 2015*.

[18] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer, "Circuits resilient to additive attacks with applications to secure computation," in *STOC 2014*.

[19] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," in *STOC 2008*.

[20] V. Goyal, A. Polychroniadou, and Y. Song, "Unconditional communication-efficient MPC via hall's marriage theorem," in *CRYPTO 2021*.

[21] V. Goyal, Y. Song, and C. Zhu, "Guaranteed output delivery comes free in honest majority MPC," in *CRYPTO 2020*.

[22] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT 2016*.

[23] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT 2010*.

[24] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang, "Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs," in *IEEE Symposium on Security and Privacy 2024*.

[25] T. Liu, X. Xie, and Y. Zhang, "zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *ACM CCS 2021*.

[26] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," in *FOCS 1990*.

[27] A. Ozdemir, "collaborative-zkSNARK implementation," 2022. [Online]. Available: https://github.com/alex-ozdemir/collaborative-zksnark

[28] A. Ozdemir and D. Boneh, "Experimenting with collaborative zk-SNARKs: Zero-knowledge proofs for distributed secrets," in *USENIX Security 2022*.

[29] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation," in *TCC 2013*.

[30] G.-V. Policharla, "zkSaaS implementation," 2023. [Online]. Available: https://github.com/guruvamsi-policharla/zksaas

[31] Polyhedra, "Expander: The fastest zk proof system to date," https://expander.polyhedra.network/, 2024.

[32] B. Schoenmakers, M. Veeningen, and N. de Vreede, "Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation," in *ACNS 16*.

[33] S. Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," in *CRYPTO 2020*.

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

It interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$.

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

- Receive shares $\{u_i, v_i\}_{i \in Corr}$ 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., $\{u_i, v_i\}_{i \in Corr}$.
- Send the shares $[\![\boldsymbol{r}]\!]_d$ and $[\![\boldsymbol{r}]\!]_{2d}$ to all parties.

Figure 7: The functionality $\mathcal{F}_{\text{Double-Rand}}$.

[34] S. Setty and J. Lee, "Quarks: Quadruple-efficient transparent zkSNARKs," Cryptology ePrint Archive, Report 2020/1275, 2020, https://eprint.iacr.org/2020/1275.

[35] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with lasso," in *EUROCRYPT 2024*.

[36] A. Shamir, "How to share a secret," *Communications of the Association for Computing Machinery*, 1979.

[37] J. Thaler, "Time-optimal interactive proofs for circuit evaluation," in *CRYPTO 2013*.

[38] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *IEEE Symposium on Security and Privacy 2013*.

[39] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *IEEE Symposium on Security and Privacy 2018*.

[40] R. Wang, C. Hazay, and M. Venkitasubramaniam, "Ligetron: Lightweight scalable end-to-end zero-knowledge proofs. post-quantum zk-snarks on a browser," in *IEEE Symposium on Security and Privacy 2024*.

[41] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A distributed zero knowledge proof system," in *USENIX Security 2018*.

[42] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkBridge: Trustless cross-chain bridges made practical," in *ACM CCS 2022*.

[43] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *CRYPTO 2019*.

[44] J. Zhang, Z. Fang, Y. Zhang, and D. Song, "Zero knowledge proofs for decision tree predictions and accuracy," in *ACM CCS 2020*.

[45] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *IEEE Symposium on Security and Privacy 2020*.

[46] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "A zero-knowledge version of vSQL," Cryptology ePrint Archive, Report 2017/1146, 2017, https://eprint.iacr.org/2017/1146.

A Helper Functionalities

Here we provide some helper functionalities that are used in the main body of this paper.

A.1 Functionalities for double random shares

Here we provide the ideal functionality for generating double-packed shares of a batch of random vectors, which is denoted as $\mathcal{F}_{\text{Double-Rand}}$ and is described in Figure 7. This functionality is commonly used in prior works, e.g., [16]; therefore, we omit its protocol realization here.

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

It interacts with a set of the 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 $(\text{PSSMULT}, [\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$ 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 $\boldsymbol{a}$ and $\boldsymbol{b}$ from $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$.
- Compute $\boldsymbol{c} := \boldsymbol{a} * \boldsymbol{b}$, 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 8: The functionality $\mathcal{F}_{\mathsf{PSSMult}}$.

Protocol Π_{PSSMult}

Let $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$ be the packed secret shares that the servers hold. The protocol allows N servers to collaboratively compute $[\![\boldsymbol{c}]\!]$, where $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.

Preprocessing of double randoms:

1. The servers invoke $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$ to obtain $[\![\boldsymbol{r}]\!]_d, [\![\boldsymbol{r}]\!]_{2d}$.

Protocol:

1. The servers locally compute $[\![\boldsymbol{c}]\!]_{2d} := [\![\boldsymbol{a}]\!]_d \cdot [\![\boldsymbol{b}]\!]_d$.
2. The servers locally compute $[\![\boldsymbol{m}]\!]_{2d} := [\![\boldsymbol{c}]\!]_{2d} + [\![\boldsymbol{r}]\!]_{2d}$ and send $[\![\boldsymbol{m}]\!]_{2d}$ to S_1. S_1 reconstructs $\boldsymbol{m}$.
3. S_1 computes and sends $[\![\boldsymbol{m}]\!]_d$ to all servers.
4. The servers locally compute $[\![\boldsymbol{c}]\!]_d := [\![\boldsymbol{m}]\!]_d - [\![\boldsymbol{r}]\!]_d$.

Figure 9: The protocol Π_{PSSMult}.

A.2 Functionality for PSS multiplication

Here we provide the ideal functionality fpssmult for PSS multiplication: given two PSS $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$, the goal is to compute $[\![\boldsymbol{c}]\!]$ such that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Formally, we present the detailed description of $\mathcal{F}_{\mathsf{PSSMult}}$ and its protocol Π_{PSSMult} in Figures 8 and 9, respectively. The security is proven through Theorem 6. The communication complexity of the online phase is $O(N)$ field elements.

Theorem 6. *The protocol Π_{PSSMult} securely realizes $\mathcal{F}_{\mathsf{PSSMult}}$ in the $\mathcal{F}_{\mathsf{Double\text{-}Rand}}$-hybrid world against a semi-honest adversary corrupting up to t servers.*

Proof. We refer readers to see the proof in [16]. □

A.3 Functionality for transforming PSS to SS

Here we provide the ideal functionality $\mathcal{F}_{\mathsf{PSSToSS}}$ for converting packed shares to regular shares. Formally, we present the detailed description of $\mathcal{F}_{\mathsf{PSSToSS}}$ and its protocol Π_{PSSToSS} in Figure 10 and 11, respectively. The security is proven through Theorem 7. The communication complexity of the online phase is $O(N)$ field elements.

Theorem 7. *The protocol Π_{PSSToSS} depicted in Figure 11 securely realizes $\mathcal{F}_{\mathsf{PSSToSS}}$ depicted in Figure 10 against a semi-honest adversary corrupting up to t servers.*

Proof sketch. The correctness is straightforward. Here we focus on the security. By the randomness extraction in [9], we conclude that for $[\![\boldsymbol{r^{(1)}}]\!]$ and $\langle r_1^{(1)} \rangle, \ldots, \langle r_k^{(1)} \rangle$ generated in the preprocessing phase, $\boldsymbol{r^{(1)}}$ is uniformly random and no server knows $\boldsymbol{r^{(1)}}$. Therefore, during the online phase, no server can learn anything about $\boldsymbol{x}$ from $\boldsymbol{y}$, since $\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{r^{(1)}}$. This completes the proof. □

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

It interacts with a set of the servers $\mathsf{S}_1, \ldots, \mathsf{S}_N$ and an adversary $\mathcal{S}$.

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

- For each $j \in [k]$, receive 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 10: The functionality $\mathcal{F}_{\mathsf{PSSToSS}}$.

Protocol Π_{PSSToSS}

Let $[\![\boldsymbol{x}]\!]$ be the PSS to be converted, where $\boldsymbol{x} \in \mathbb{F}^k$. Let $\boldsymbol{V}_{N,N-t}$ be a public Vandermonde matrix with N rows and $N-t$ columns. The protocol allows N servers to collaboratively compute $\langle x_1 \rangle, \ldots, \langle x_k \rangle$.

Preprocessing of randoms:

1. Each server S_i picks an uniformly random $\boldsymbol{u^{(i)}} \in \mathbb{F}^k$ and computes $[\![\boldsymbol{u^{(i)}}]\!], \langle u_1^{(i)} \rangle, \ldots, \langle u_k^{(i)} \rangle$.
2. Each server S_i sends the j-th shares $[\![\boldsymbol{u^{(i)}}]\!]_j, \langle u_1^{(i)} \rangle_j, \ldots, \langle u_k^{(i)} \rangle_j$ to S_j, for $j \in [N]$.
3. Each server S_i locally computes $([\![\boldsymbol{r^{(1)}}]\!]_i, \ldots, [\![\boldsymbol{r^{(N-t)}}]\!]_i) = \boldsymbol{V}_{N,N-t}^{\mathrm{T}} \cdot ([\![\boldsymbol{u^{(1)}}]\!]_i, \ldots, [\![\boldsymbol{u^{(N)}}]\!]_i)$ and $(\langle r_j^{(1)} \rangle_i, \ldots, \langle r_j^{(N-t)} \rangle_i) = \boldsymbol{V}_{N,N-t}^{\mathrm{T}} \cdot (\langle u_j^{(1)} \rangle_i, \ldots, \langle u_j^{(N)} \rangle_i)$ for $j \in [k]$.

Protocol:

1. Each server S_i locally computes $[\![\boldsymbol{y}]\!] := [\![\boldsymbol{x}]\!] - [\![\boldsymbol{r^{(1)}}]\!]$ and send $[\![\boldsymbol{y}]\!]$ to S_1.
2. S_1 reconstructs $\boldsymbol{y} \in \mathbb{F}^k$ and sends $\boldsymbol{y}$ to all servers.
3. The servers compute $\langle x_j \rangle := y_j + \langle r_j^{(1)} \rangle$ for $j \in [k]$.

Figure 11: The protocol Π_{PSSToSS}.

A.4 Functionality for generating masks

Here we provide details for generating masks and unmasks used in the preprocessing phase of $\Pi_{\mathsf{dProdTree}}$. The functionality and protocol are presented in Figures 12 and 13. The security is proven through Theorem 8. The protocol needs $O(\log N)$ round complexity to complete the PSS multiplications, and the communication complexity is $O(n)$.

Theorem 8. *The protocol $\Pi_{\mathsf{Rand\text{-}ProdTree}}$ securely realizes $\mathcal{F}_{\mathsf{Rand\text{-}ProdTree}}$ in the $\mathcal{F}_{\mathsf{PSSMult}}$-hybrid world against a semi-honest adversary corrupting up to t servers.*

Proof sketch. The correctness is straightforward. Except for invoking $\mathcal{F}_{\mathsf{PSSMult}}$ and sharing some random values, the servers only perform local computation, and the corrupted parties have no chance to learn $\boldsymbol{m_j}, \boldsymbol{u_j}$ for $j \in [\frac{n}{k}]$. Hence, the security is guaranteed. □

A.5 Functionality for distributed MSM

In [16], the authors provide a functionality for distributing the computation of MSM, i.e., given $A_1, ..., A_n$ as n group elements in $\mathbb{G}$ and $b_1, ..., b_n$ as n field elements in $\mathbb{F}$, the N servers collaborate to compute out $= \prod_{i \in [n]} A_i^{b_i}$. The functionality and protocol are presented in Figures 14 and 15. The security of the protocol Π_{dMSM} is proven through Theorem 9. The protocol is both time-efficient and space-efficient, as each server only computes $\frac{n}{N}$ group elements, and the total communication cost is $O(N)$.

[2] $\tau(\cdot)$ and $\tau^{\mathsf{inv}}(\cdot)$ are two functions, mapping $[n] \to [n+1]$. Let $n = 2^\ell$. For each $j \in [n-1]$, $\tau(j)$ and $\tau^{\mathsf{inv}}(j)$ can be computed by following steps. First, let $s = \lfloor \log(n-j) \rfloor$. Then, set $\tau(j) = (j - \sum_{t=1}^{\ell-s-1} 2^{\ell-t}) \cdot 2^{\ell-s} + 1$ and $\tau^{\mathsf{inv}}(j) = (j - \sum_{t=1}^{\ell-s-1} 2^{\ell-t} - 1) \cdot 2^{\ell-s} + 1$. Finally, set $\tau(n) = \tau^{\mathsf{inv}}(n) = 0$.

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

It interacts with a set of the 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 $(\text{RANDPRODTREE}, n)$ from the servers, do:

- Sample $\{r_i\}_{i\in[n+1]} \xleftarrow{\$} \mathbb{F}^{n+1}$ and construct $\boldsymbol{r} = (r_1, r_2 \cdots, r_n)$, $\boldsymbol{r}^{\mathsf{inv}} = (r_2^{-1}, \cdots, r_n^{-1}, r_{n+1}^{-1})$.
- Compute $\tilde{\boldsymbol{r}} = \{\tilde{r_j}\}_{j\in[n]}$, $\boldsymbol{r}^{\tilde{\mathsf{inv}}} = \{r_j^{\tilde{\mathsf{inv}}}\}_{j\in[n]}$, where $\tilde{r_j} = r_{\tau(j)}$ and $r_j^{\tilde{\mathsf{inv}}} = r_{\tau^{\mathsf{inv}}(j)}^{-1}$ for $j \in [n]$[2].
- Compute $\boldsymbol{m} := \boldsymbol{r} * \boldsymbol{r}^{\mathsf{inv}}$ and $\boldsymbol{u} = \tilde{\boldsymbol{r}} * \boldsymbol{r}^{\tilde{\mathsf{inv}}}$.
- For $i \in [\frac{n}{k}]$:
  - Set $\boldsymbol{m_i} := (m_{(i-1)k+1}, ..., m_{ik})$, $\boldsymbol{u_i} := (u_{(i-1)k+1}, ..., u_{ik})$
  - Receive a set of shares $\{v_{i,j}, w_{i,j}\}_{j\in\mathcal{C}orr}$ from $\mathcal{S}$.
  - Sample random PSS $[\![\boldsymbol{m_i}]\!]$ of $\boldsymbol{m_i}$ and $[\![\boldsymbol{u_i}]\!]$ of $\boldsymbol{u_i}$ s.t. the shares of the corrupted parties are identical to those received from $\mathcal{S}$, i.e., $\{v_{i,j}, w_{i,j}\}_{j\in\mathcal{C}orr}$.
- Distribute the shares $\{[\![\boldsymbol{m_i}]\!]\}, \{[\![\boldsymbol{u_i}]\!]\}$ to all servers.

Figure 12: The functionality $\mathcal{F}_{\text{Rand-ProdTree}}$.

Protocol $\Pi_{\text{Rand-ProdTree}}$

The protocol allows N servers to collaboratively compute packed secret shares of masks and unmasks.

Protocol:

1. Each server S_i samples $\{r_j\}_{j\in[n+1]}$ and constructs $\boldsymbol{r} = (r_1, r_2 \cdots, r_n)$, $\boldsymbol{r}^{\mathsf{inv}} = (r_2^{-1}, \cdots, r_n^{-1}, r_{n+1}^{-1})$.
2. Each server S_i computes $\tilde{\boldsymbol{r}} = \{\tilde{r_j}\}_{j\in[n]}$, $\boldsymbol{r}^{\tilde{\mathsf{inv}}} = \{r_j^{\tilde{\mathsf{inv}}}\}_{j\in[n]}$, where $\tilde{r_j} = r_{\tau(j)}$ and $r_j^{\tilde{\mathsf{inv}}} = r_{\tau^{\mathsf{inv}}(j)}^{-1}$.
3. Each server S_i computes $\boldsymbol{v_i} := \boldsymbol{r} * \boldsymbol{r}^{\mathsf{inv}}$, $\boldsymbol{w_i} = \tilde{\boldsymbol{r}} * \boldsymbol{r}^{\tilde{\mathsf{inv}}}$, computes and sends $[\![\boldsymbol{v_{i,j}}]\!], [\![\boldsymbol{w_{i,j}}]\!]$ to others for $j \in [\frac{n}{k}]$.
4. Servers invoke $\mathcal{F}_{\text{PSSMult}}$ to compute $[\![\boldsymbol{m_j}]\!]$ and $[\![\boldsymbol{u_j}]\!]$ where $\boldsymbol{m_j} = \bigodot_{i=1}^{N} \boldsymbol{v_{i,j}}$ and $\boldsymbol{u_j} = \bigodot_{i=1}^{N} \boldsymbol{w_{i,j}}$ for $j \in [\frac{n}{k}]$. Here $\bigodot$ denotes a consecutive coordinate-wise product.

Figure 13: The protocol $\Pi_{\text{Rand-ProdTree}}$.

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

The functionality $\mathcal{F}_{\text{dMSM}}$ 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 corrupted servers. It does:

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

1. For $i \in [m]$, 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 $\mathsf{out} = \prod_{i\in[m\cdot k]} A_i^{b_i}$.
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 14: The functionality $\mathcal{F}_{\text{dMSM}}$

Theorem 9. *The protocol Π_{dMSM} securely realizes $\mathcal{F}_{\text{dMSM}}$ in the $\{\mathcal{F}_{\text{Double-Rand}}, \mathcal{F}_{\text{PSSToSS}}\}$-hybrid world against a semi-honest adversary corrupting up to t servers.*

Proof. We refer readers to see the proof in [16]. □

Protocol Π_{dMSM}

Let $A_1, ..., A_n$ be n group elements in $\mathbb{G}$ and $b_1, ..., b_n$ be n field elements in $\mathbb{F}$. We assume $k|n$, where k is the packing factor, and we set $m := n/k$.

Inputs: 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 of double randoms:

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 $\mathcal{F}_{\mathsf{PSSToSS}}$, which returns $\langle r_1\rangle, ..., \langle r_k\rangle$ to the servers.

Protocol:

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 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 output.

Figure 15: The Π_{dMSM} Protocol .

B Collaborative Libra

This section offers an overview of Libra [43] and how to implement a collaborative Libra for data-parallel circuits. We concentrate on a version without the zero-knowledge property, and it can be incorporated following the methodologies outlined in prior works [43, 46].

GKR protocol. The Libra protocol is based on the well-known GKR protocol [19]: For a layered circuit, define a function $V_i : \{0,1\}^\mu \to \mathbb{F}$ that takes a gate label $\boldsymbol{b} \in \{0,1\}^\mu$ and returns the output of gate $\boldsymbol{b}$ in layer i. With this definition, V_0 corresponds to the output of the circuit and V_d corresponds to the input layer. We define two additional functions $add_i, mult_i$: $\{0,1\}^{3\mu} \to \{0,1\}$, referred to as wiring predicates in the literature. add_i ($mult_i$) takes one gate label $\boldsymbol{z} \in \{0,1\}^\mu$ in layer $i-1$ and two gate labels $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^\mu$ in layer i, and outputs 1 if and only if the gate $\boldsymbol{z}$ is an addition (multiplication) gate that takes the output of gate $\boldsymbol{x}, \boldsymbol{y}$ as input. With these definitions, for any $\boldsymbol{g} \in \mathbb{F}^\mu$, $\tilde{V}_i$ can be written as the following GKR relation:

$$\tilde{V}_i(\boldsymbol{g}) = \sum_{\boldsymbol{x},\boldsymbol{y}\in\{0,1\}^\mu} \tilde{add}_{i+1}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})(\tilde{V}_{i+1}(\boldsymbol{x}) + \tilde{V}_{i+1}(\boldsymbol{y})) + \tilde{mult}_{i+1}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{y})\tilde{V}_{i+1}(\boldsymbol{x})\tilde{V}_{i+1}(\boldsymbol{y}) \ .$$

Consider a data-parallel circuit comprising B identical copies, where each sub-copy is a d-depth layered circuit. Assume there are in total n gates in each layer of the circuit, where $n = 2^\mu$. In the collaborative setting, the critical step for achieving collaborative Libra for data-parallel circuits involves organizing the values at the corresponding positions across different sub-copies of the circuit into the same vectors. These vectors are then distributed among the servers using packed secret sharing. We assume that each server receives packed secret shares of $\tilde{V}_i$'s evaluations on the hypercube. Specifically, the variables in the same position of each sub-copy are packed together.

Setup. The setup of Libra involves invoking mvPC.Setup to the parameter of polynomial commitments. We assume servers receive parameters from dMVPC.Setup.

Prove. The prover interacts with the verifier in $O(d)$ rounds.

1. The prover commits to $\tilde{V}_d$. We assume servers invoke dMVPC.Commit to generate the commitment.
2. The prover sends the claimed output of the circuit to the verifier. The verifier defines the polynomial $\tilde{V}_0$ and computes $\tilde{V}_0(\boldsymbol{g})$ for a randomly chosen $\boldsymbol{g} \in \mathbb{F}^\mu$. The prover and verifier engage in a sumcheck about $\tilde{V}_0(\boldsymbol{g})$ on the GKR relation. At the end of the sumcheck, the claim is reduced to $\tilde{V}_1(\boldsymbol{u}^{(1)}), \tilde{V}_1(\boldsymbol{v}^{(1)})$, where $\boldsymbol{u}, \boldsymbol{v}$

are selected randomly in $\mathbb{F}^\mu$. We assume servers invoke a collaborative sumcheck protocol to support this process. This is feasible if the circuit is data-parallel.

3. From layer i to $d-1$, the two claims $\tilde{V}_i(\boldsymbol{u}^{(i)}), \tilde{V}_i(\boldsymbol{v}^{(i)})$ are combined using a random linear combination. The prover and verifier conduct a new sumcheck about the newly formulated claim. At the end of the sumcheck, the verifier receives two claims about $\tilde{V_{i+1}}$, and this process is recursively continued until reaching the input layer. We assume that, at each layer, servers invoke a collaborative sumcheck protocol to support this process.

4. At the input layer, the prover evaluates the polynomial $\tilde{V}_d$ at a random point to validate its claim. We assume servers use dMVPC.Open to support this process.

C Collaborative HyperPlonk

This section provides an overview of HyperPlonk [7] and how to implement a collaborative HyperPlonk for general circuits. We concentrate on a version without the zero-knowledge property, though it can be incorporated following the methodologies outlined in [6, Appendix A]. We also omit the optimizations introduced in [7] for batching protocols and supporting customized gates and lookup operations.

Arithmetization. Consider a general circuit n comprising n inputs and m gates, each with a fan-in of two, performing either addition or multiplication. Let $2^\mu = m + n + 1$. The arithmetization involves an input polynomial I, selector polynomials S_1, S_2, and a permutation polynomial $\hat{\sigma}$. The computation trace is captured by a set $\hat{M}$ of triples $\{(L_i, R_i, O_i) \in \mathbb{F}^3\}_{i=0,\ldots,n+m}$, where each triple represents the wires of the i-th gate. The prover defines a $(\mu+2)$-variate polynomial M as the multilinear extension of $\hat{M}$, such that for all $i \in \{0, \ldots, n+m\}$,

$$M(0, 0, \langle \boldsymbol{i} \rangle) = L_i, \quad M(0, 1, \langle \boldsymbol{i} \rangle) = R_i, \quad M(1, 0, \langle \boldsymbol{i} \rangle) = O_i$$

Here $\langle \boldsymbol{i} \rangle$ denotes the binary representation of an integer i. Given these, to prove the correctness of computation, the prover needs to check

- Gate identity: $\forall \boldsymbol{x} \in \{0,1\}^\mu$,

$$\begin{aligned} &F(\boldsymbol{x}) = S_1(\boldsymbol{x}) \cdot (M(0,0,\boldsymbol{x}) + M(0,1,\boldsymbol{x})) + \\ &S_2(\boldsymbol{x}) \cdot M(0,0,\boldsymbol{x}) \cdot M(0,1,\boldsymbol{x}) - M(1,0,\boldsymbol{x}) + I(\boldsymbol{x}) = 0 \end{aligned}$$

- Wire identity: $\forall \boldsymbol{x} \in \{0,1\}^{\mu+2}$, $M(\boldsymbol{x}) = M(\hat{\sigma}(\boldsymbol{x}))$.

We assume servers receive packed secret shares of all the above multilinear polynomials' evaluations on their respective hypercubes.

Setup. The setup of HyperPlonk involves invoking mvPC.Setup to prepare parameters for the polynomial commitment of each multilinear polynomial. We assume servers receive parameters according to dMVPC.Setup.

Prove. The prover interacts with verifier in constant rounds.

1. The prover commits to polynomials separately. We assume servers invoke dMVPC.Commit to generate commitments.

2. The prover and verifier engage in a permutation-check concerning wire identity, which can be reduced to a product check. We assume servers invoke a collaborative productcheck to support this process, involving collaboratively computing the product tree, a collaborative zerocheck and dMVPC.Open.

3. The prover and verifier run a zerocheck protocol on $F(\boldsymbol{x})$. We assume that, after performing field operations on the corresponding packed shares, servers invoke a collaborative zerocheck protocol to support this process.

4. The prover evaluates multiple multilinear polynomials at random points to validate its claim. We assume servers use dMVPC.Open to support this process.