zkSaaS: Zero-Knowledge SNARKs as a Service

Sanjam Garg[1], Aarushi Goel[2], Abhishek Jain[3], Guru-Vamsi Policharla[4], and Sruthi Sekar[4]

[1]UC Berkeley and NTT Research, sanjamg@berkeley.edu
[2]NTT Research, aarushi.goel@ntt-research.com
[3]Johns Hopkins University, abhishek@cs.jhu.edu
[4]UC Berkeley, {guruvamsip,sruthi}@berkeley.edu

Abstract

A decade of active research has led to practical constructions of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) that are now being used in a wide variety of applications. Despite this astonishing progress, overheads in proof generation time remain significant.

In this work, we envision a world where consumers with low computational resources can outsource the task of proof generation to a group of untrusted servers in a privacy-preserving manner. The main requirement is that these servers should be able to collectively generate proofs at a *faster* speed (than the consumer). Towards this goal, we introduce a framework called zk-SNARKs-as-a-service (zkSaaS) for faster computation of zk-SNARKs. Our framework allows for distributing proof computation across multiple servers such that each server is expected to run for a shorter duration than a single prover. Moreover, the privacy of the prover's witness is ensured against any minority of colluding servers.

We design custom protocols in this framework that can be used to obtain faster runtimes for widely used zk-SNARKs, such as Groth16 [EUROCRYPT 2016], Marlin [EUROCRYPT 2020] and Plonk [EPRINT 2019]. We implement proof of concept zkSaaS for the Groth16 and Plonk provers. In comparison to generating these proofs on commodity hardware, we show that not only can we generate proofs for a larger number of constraints (without memory exhaustion), but can also get $\approx 22\times$ speed-up when run with 128 parties for 2^{25} constraints with Groth16 and 2^{21} gates with Plonk.

Contents

1 Introduction

zk-SNARKs are zero-knowledge succinct non-interactive arguments of knowledge [11], that allow a prover to non-interactively convince a verifier of the knowledge of a witness attesting to the validity of an NP relation, without revealing any information about the witness. zk-SNARKs have been a topic of extensive research in recent years [19, 20, 81, 71, 21, 59, 58, 84, 22, 50, 28, 39]. Their flexibility and expressiveness make them applicable to a wide variety of scenarios such as private transactions [7, 66], roll-ups [87], private smart contracts [54, 23], access control in compliance with KYC regulations [68, 69], social networks with private reputation monitoring [26], proving existence of bugs in zero-knowledge [49], static program analysis [37], zero-knowledge middleboxes for enforcing network policies on encrypted traffic [52], verifiable inference of machine learning [83, 62, 60, 79] and verifiable database queries [85].

Despite recent advances [12, 53, 15, 14, 50, 28, 39, 30], generation of zk-SNARKs remains thousands of times [78, 64] slower than checking the relation directly for typical applications, with large memory usage — effectively gate keeping users without access to large machines. A natural way out for such users is to outsource proof generation to more powerful servers. While one could use a cloud server such as AWS, GCP or Azure to generate proofs, this approach requires sharing the witness in the clear with the cloud server. As such, this solution offers *no privacy* against insider threats such as rogue administrators [32] who may compromise data privacy for financial gains. Even if the cloud service provider were trusted, the witness might consist of sensitive data such as patient medical records that legally cannot be placed off-premises due to data protection laws.

To address the privacy problem, recently, Ozdemir et al. [64] introduced the idea of *collaborative zk-SNARKs* for distributed generation of zk-SNARKs. Collaborative zk-SNARKs are essentially secure multiparty computation (MPC) protocols that allow a group of parties holding shares of the witness to collectively generate a *single* succinct proof. The key security guarantee is that the witness remains hidden as long as only a subset of the parties collude. Ozdemir et al. design collaborative zk-SNARK analogs of Groth16 [50], Marlin [28] and Plonk [39]. In their protocols, all parties run in parallel and each of them performs as much work as the (single) prover of the underlying zk-SNARK. This results in approximately the same runtime as that in the single prover model.[1]

We posit that requiring each of the parties running in parallel to do as much work as the zk-SNARK prover is an overkill. Indeed, a previous work of Wu et al. [80] leveraged parallelism to distribute proof computation across different machines in a compute cluster to achieve *faster* proof generation times. Their approach, however, requires leaking the witness to the cluster, resulting in a loss of privacy.

In this work, we explore the possibility of combining the best features of collaborative zk-SNARKs [64] and the work of Wu et al. [80]. We ask:

Is it possible to outsource zk-SNARK proof generation to a group of parties in a privacy-preserving manner for faster proof generation?

Our Contributions. Our contributions are as follows:

1. We present a general framework for *zk-SNARKs-as-a-service* (zkSaaS), where a client delegates proof computation to a group of untrusted servers in a privacy preserving manner. Each of these servers is expected to run for a shorter duration than a single local prover.

2. We instantiate this framework with custom protocols to obtain faster runtimes than local provers for widely used zk-SNARKs, such as Groth16 [50], Marlin [28] and Plonk [39].

[1] This is interesting since they manage to avoid additional security parameter overhead that is usually incurred when *securely* computing a function.

3. Finally, we implement prototypes of zkSaaS for the Groth16 [50] and Plonk [39] proof systems. Concretely, we show that when creating a proof for 2^{25} constraints in the case of Groth16 (and 2^{21} constraints with Plonk), the zkSaaS protocol with 128 servers is $\approx$ 22× faster than a local prover. We also show that deploying more servers helps us get a further speed-up. For instance, when creating a Groth16 proof for 2^{19} constraints, we see an improvement from $\approx$ 1.9× to $\approx$ 22× when the number of servers is increased from 8 to 128. This is in contrast to collabortive zk-SNARKs [64] which do not obtain any speedup.

4. We also estimate the financial cost of using zkSaaS to compute a Groth16 proof for an instance of size 2^{19} to be under \$0.23 with 128 parties using a 64 Mbps link between servers.

1.1 Overview of Our Approach

Similar to [64], our initial idea is to identify common building blocks within widely used zk-SNARKs and design custom secure multiparty computation (MPC) protocols to compute them efficiently. We then stitch them together to obtain zkSaaS, an efficient MPC protocol for the corresponding zkSNARK prover.

An Important Observation. One of the key observations made in [64] is that it is possible to directly secret share points on the elliptic curve and fields and apply MPC techniques on these shares, which avoids the large overheads incurred when using generic MPC techniques with very few rounds of communication.[2] We go one step ahead and observe that these building blocks can actually be rewritten in a manner that allows us to leverage significant SIMD structure that appears within their computation.

To leverage this SIMD structure, we make use of a tool called *packed secret sharing* (PSS) [38].[3] This is a more efficient sibling of Shamir's polynomial-based secret sharing scheme [72], that allows secret-sharing a *vector of values* amongst a set of parties. In particular, at the cost of a slight reduction in the corruption threshold, using PSS we can "hide" $\ell = \mathcal{O}(n)$[4] secrets (where n is the total number of servers) in a polynomial and each of the n servers receives an evaluation at a single point in this polynomial. Such sharings allow parties to efficiently perform SIMD computations on secret shared data, while reducing the workload on each of them. We use this in the design of each of our sub-protocols.

Multi-Scalar Multiplications (MSM). One of the main building blocks in the zkSNARKs we consider is MSM, which are operations of the form $\Pi_{i\in[m]} g_i^{\alpha_t}$, where g_i's are points on an elliptic curve. This is by far the most expensive component.[5] We design a bespoke MPC protocol where the total work (as well as the asymptotic space requirement) of each server is a factor of ℓ less than that of a single prover.

Next we discuss the remaining components i.e., Product Check, Fast Fourier Transform (FFT) and polynomial computations, which exclusively involve field operations that are much cheaper than elliptic curve operations.

Product-Check. Product-Check requiring computations of the form $\prod_{j\in[i]} x_j$, for all $i \in [m]$, which are referred to as *partial products*. Similar to MSM, we design a special-purpose MPC protocol for partial products, that allows us to divide the work of each server by a factor of ℓ less than that of a single prover.

Fast Fourier Transform. The standard description of the FFT algorithm on a polynomial with m coefficients can be divided into $\log m$ steps, with $O(m)$ field multiplications at each step. For the first $\log m/\ell$ steps, we are able to divide the work of each server by a factor of ℓ. For the remaining $\log \ell$ steps, however,

[2] This is mainly due to generic MPC techniques making non-blackbox use of the elliptic curve.

[3] This is also the main building block used in the design of all of general-purpose MPC protocols that support some division of work (See Section 1.3 for more details.)

[4] In the implementation we set this to be $n/4$.

[5] In Figure 7 we show the fraction of time spent computing MSMs for Groth16.

we require one of the parties to do $O(m)$ field operations and have $O(m)$ memory, while the work and memory requirement of the remaining parties gets divided by ℓ.

Polynomial Multiplication and Division. Finally, we show how to combine standard packed secret sharing based subprotocols for addition and multiplication along with our custom MPC for FFT to enable secure distributed polynomial computations.

Composing different subprotocols based on packed secret sharing is not straightforward, and requires care. We show how to combine the above subprotocols to obtain zkSaaS for faster generation of zk-SNARKs such as Groth16, Marlin and Plonk.

Communication over a Star Topology Network. Our distributed sub-protocols for all of the functions described above, do not require servers to communicate with all other servers. Aside from receiving shares of the extended witness from the client, we require the servers to only communicate with the one large server, throughout the rest of the computation. As a result, we only need communication channels between the client and each server and between the large server and every other server.

Instantiating zkSaaS. As discussed above, the distributed FFT protocol requires one party which has memory proportional to the size of the relation but the computational resources demanded from all other parties is reduced by a factor of ℓ. Therefore, a zkSaaS deployment requires one large server. While not ideal, we argue that this is still reasonable for two reasons — (1) even if the private view of this large server is leaked, it does not compromise a client's secrets unlike when a client simply rents a large server to generate the proof. (2) it is very easy and quite cheap to rent a large server from a cloud service provider.

Finally, we remark that proof generation in the zk-SNARKs that we consider proceeds in two steps: first, the prover uses its (short) witness to evaluate the relation circuit and obtain a corresponding *extended witness*, which is then used to generate the proof (See Section 3 for a detailed discussion). Similar to collaborative zk-SNARKs [64], in this work, we focus on designing secure distributed protocols for proof generation and assume that the client computes and shares the extended witness with the servers. We view faster generation of the extended witness as an important orthogonal question but such protocols would need to essentially be redesigned for every application.

Security. We now discuss the key aspects of our security model and the guarantees provided by zkSaaS.

We assume that a majority of the servers are honest. Specifically, let n be the total number of servers and ℓ be the total number of secrets that we can pack in a single packed secret sharing. We require that at most $t < \frac{n}{2} - \ell$ of the servers can be corrupted. Further, we assume that the corruptions are *semi-honest*.

Our zkSaaS framework retains the soundness property of the underlying zk-SNARK and provides the following completeness and zero-knowledge guarantees:

— *Completeness*: For any true statement, an execution of zkSaaS involving an honest client and honest servers outputs an accepting proof.[6]

— *t-Zero Knowledge:* In any execution of zkSaaS, the view of the t corrupt servers can be efficiently simulated without the client's witness. This, in particular, implies that the corrupt servers learn nothing about the client's witness.

We conclude with a few remarks on security against *malicious* servers. We first note that proofs output by zkSaaS remain sound even when *all* servers are malicious. Next, we conjecture that our protocols can be augmented to achieve t-zero knowledge against malicious servers by using highly efficient compilers from the recent MPC literature [41, 40, 6, 46, 47]. In essence, these compilers show that semi-honest MPC protocols that are "secure against malicious corruptions up to linear attacks" can be compiled into

[6] The completeness property, in fact, holds even if the servers are semi-honest (since such servers follow protocol instructions.)

maliciously secure protocols with a small constant (typically, at most two) overhead. We conjecture that our semi-honest zkSaaS protocols already satisfy the properties required for these efficient compilers; a formal treatment of the same, however, is outside the scope of this work.

1.2 Example Applications of zkSaaS

We now discuss some real-world applications where we envision our zkSaas-framework to be useful.

Private Transactions and Smart Contracts. A simple spend transaction on a private chain such as ZCash[7] already involves $\approx 130,000$ R1CS constraints[7] which takes roughly 10 seconds on a high-end laptop in single threaded mode. This would take even longer on weaker devices such as smart phones making the process quite tedious for users.

Private smart contracts are immutable programs running on blockchains, which provide confidentiality of the computation carried out on blockchains [54, 23]. Although these chains can be designed more carefully to reduce the overhead for simple transactions, they aim to support general computation on the smart contracts which can blow up to a very large number of constraints as there may be very complicated logic that gets executed involving cryptographic functions such as signature verification. With zkSaaS, users can potentially pay a tiny transaction fee in exchange for a seamless experience akin to current centralized payment methods.

Statements involving Ethereum Wallets. Ethereum uses EdDSA signatures for authentication of transactions over the ed25519 elliptic curve which is not proof friendly. As a result, one needs to emulate non-native 256-bit field arithmetic which is quite expensive. The verification circuit of an EdDSA signature costs over 2.5 million R1CS constraints[8] and the proving key itself is 1.6 GB in size. Common tasks such as proof of membership viz. "I own an Ethereum wallet out of these set of 1024 wallets" become impractical on mobile devices as the statements are simply too large and lead to memory exhaustion.

Combating Disinformation. It was shown that zero-knowledge proofs can be used to prove that images appearing in news articles underwent an approved set of transformations from the time of creation [63]. This is particularly helpful in allowing reporters to hide sensitive content while at the same time proving authenticity of the image. While fast generation of zk-SNARKs is possible for images, doing the same for compute-heavy video files is currently far from practical and our zkSaaS-framework could aid in carrying out such a computation.

Verifiable Private ML Inference. A user can commit to a machine learning model and provide a proof of inference on this machine learning model, which can be used to verify accuracy of a machine learning model or to ensure that a certain entity who claims to use AI for a task is actually producing predictions using a machine learning model. Since circuits for inference can be quite large as the models grow in size, they quickly become impractical to prove even on consumer grade laptops. Again, the data used to train this model could be patient health records for example which cannot be placed on servers that do not comply with HIPAA[9]. Hence, a solution is to use zkSaaS to compute proofs of inferences where no server sees sensitive information.

1.3 Related Work

Some prior works [55, 31] have considered building MPC-as-a-service, which involves deploying MPC in a volunteer-operated network (like blockchains). However, such protocols are built for generic function-

[7] https://github.com/zcash/librustzcash
[8] https://github.com/Electron-Labs/ed25519-circom
[9] https://www.hipaajournal.com/

alities and do not offer efficient solutions for our specific goal. In a different line of work (unrelated to our goal), MPC has been used to securely sample the common parameters used in zk-SNARKs [8, 24, 57].

A different line of work has considered the problem of speeding-up the zk-SNARK prover time, but they either do not hide the witness [80, 70], or lead to [25] linear verification time with a security guarantee that is weaker than both our framework and the collaborative zk-SNARK framework [64]. Some prior works have also studied other distributed models of proof systems, including ones where the statement is shared amongst multiple verifiers [1, 17, 18], or where there are two (or more) non-colluding provers [4, 13].

Our goal in some sense is very similar to the design of MPC protocols, where the total computation and communication is independent of the number of parties, which has been the focus of a significant line of research [35, 34, 40, 45, 6, 46, 48]. However for arithmetic circuits, most of these require round complexity linear in the multiplicative depth of the circuit, which is not ideal in our setting since the prover algorithms in zk-SNARKs typically don't have a constant multiplicative depth. Moreover, representing cryptographic operations such as group exponentiations as an arithmetic circuit and computing them inside an MPC is extremely inefficient. Therefore, naïvely using these protocols in computing a zk-SNARK will result in inefficient solutions.

More recently, in a concurrent and independent work, Chiesa et al. [29] also considered the problem of private delegation of zk-SNARKs for faster proof generation. However, their model is quite different from ours. In particular, they assume that the client remains online throughout the computation and actively participates in the zk-SNARK computation along with the servers. For this, they design an MPC protocol (that is run between the servers and the clients) for zk-SNARK computation leveraging the fact that one of the parties (i.e., the client) is always honest and the witness need not be hidden from them. Their goal is to essentially "reduce" the work done by the client. In contrast, in our setting, after sharing the extended witness, the client does not need to do any work and can delegate the *entire* zk-SNARK computation to the servers.

1.4 Future Directions

A promising direction for future work would be to eliminate the need of a single large server in zkSaaS. In our current solution, this large server is only needed for our distributed protocol for FTT. Potential approaches for avoiding this could be – (1) Designing a more efficient subprotocol for for distributed computation of FFT or (2) designing zkSaaS for zk-SNARKs that do not use FFT operations e.g. Orion [82], Brakedown [44], Hyperplonk [27]. Another interesting problem would be to enable faster generation of the extended witness in a similar framework. Finally, as discussed in Section 1.1, it would be interesting to formally demonstrate how the protocols developed in this work can be augmented to achieve security against malicious servers.

Paper Organization. We start by establishing some notations in Section 2. We then formally define zkSaaS- in Section 3, and give an overview of the popular zk-SNARKs of interest (Groth, Marlin and Plonk) in Section 4. A detailed technical exposition of each of our distributed sub-protocols (FFT, MSM and sum of partial products) appears in Section 5, and we show how to build a zkSaaS for a specific class of zk-SNARKs in Section 6. Finally, we discuss the concrete efficiency of our scheme in Section 7.

2 Preliminaries

For any $n \in \mathbb{N}$, we use $[n]$ to denote the set $\{1, \cdots, n\}$ and for $i, j \in \mathbb{N}$ with $i < j$, we use $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$. We denote a vector of ℓ elements from a field $\mathbb{F}$, $(x_1, \cdots, x_\ell)$, by $\mathbf{x}$. For any two vectors $\mathbf{x}$ and $\mathbf{y}$, the component-wise multiplication is denoted by $\mathbf{x} \odot \mathbf{y} := (x_1 \cdot y_1, \cdots, x_\ell \cdot y_\ell)$. We always use capital letters (e.g. X) to denote elements from a group $\mathbb{G}$, and correspondingly $\mathbf{X}$ denotes a vector of

ℓ group elements $(X_1, \cdots, X_\ell)$. We use a multiplicative notation for our group operations throughout the paper. We begin by describing the notations specific to linear secret sharing schemes used by us.

Linear Secret Sharing Schemes. In this work we make use of polynomial based, regular threshold secret sharing scheme as well as a packed secret sharing scheme. For regular threshold secret sharing, we use $[x]$, $\langle x \rangle$ to denote shares of a value x, w.r.t. to a degree t and $n-1$ polynomial respectively. For packed secret sharing, we use $[\![\mathbf{x}]\!]$, $\langle\!\langle \mathbf{x} \rangle\!\rangle$ to denote shares of a vector $\mathbf{x}$ w.r.t. to a degree D and $n-1$ polynomial respectively, where we assume that the length of $\mathbf{x}$ is $\ell \in \mathcal{O}(n)$ and $D = t + \ell$. We use $[x]_i, \langle x \rangle_i, [\![\mathbf{x}]\!]_i, \langle\!\langle \mathbf{x} \rangle\!\rangle_i$ to denotes shares held by a party P_i and $[x]_S, \langle x \rangle_S, [\![\mathbf{x}]\!]_S, \langle\!\langle \mathbf{x} \rangle\!\rangle_S$ to denote the shares held by a subset S of the parties. Finally, we use functions $[x] \leftarrow \mathsf{share}(\mathbb{F}, x, t)$ and $[\![\mathbf{x}]\!] \leftarrow \mathsf{pshare}(\mathbb{F}, \mathbf{x}, D)$ to compute shares and $\mathsf{open}(\mathbb{F}, [\![\mathbf{x}]\!], D)$ and $\mathsf{open}(\mathbb{F}, [x], t)$ to reconstruct shares.

In the subsequent sections below, we formally define bilinear groups, zk-SNARKs and secure multi-party computation.

2.1 Bilinear Groups

Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ be cyclic groups of prime order q with generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$. $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable and non-degenerate pairing, such that $e(h_1^\alpha, h_2^\beta) = e(h_1, h_2)^{\alpha\beta}$, for all $\alpha, \beta \in \mathbb{F}_q$, and all $h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$.

2.2 Succinct Non-Interactive Arguments of Knowledge

In this section, we provide a formal definition for the notion of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs).

Definition 1 (zk-SNARKs [50])**.** *Let $\mathcal{R}$ be a relation generator that given a security parameter λ in unary returns a polynomial time decidable binary relation R. For pairs $(\phi, w) \in R$ we call ϕ the statement and w the witness. We define $\mathcal{R}_\lambda$ to be the set of possible relations R that the relation generator may output given 1^λ. We will in the following for notational simplicity assume λ can be deduced from the description of R. The relation generator may also output some side information, an auxiliary input z, which will be given to the adversary. An efficient prover publicly verifiable non-interactive argument for $\mathcal{R}$ is a quadruple of probabilistic polynomial algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ *defined as:*

- $(\mathsf{crs}, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R)$*: The setup takes the security parameter λ and the relation R as input and produces a common reference string* crs *and a simulation trapdoor τ.*
- $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, R, \phi, w)$*: The prover algorithm takes as input a common reference string* crs *and $(\phi, w) \in R$ and returns an argument π.*
- $0/1 \leftarrow \mathsf{Ver}(\mathsf{crs}, R, \phi, \pi)$*: The verification algorithm takes as input a common reference string* crs*, a statement ϕ and an argument π and returns 0 (reject) or 1 (accept).*
- $\pi \leftarrow \mathsf{Sim}(R, \tau, \phi)$*: The simulator takes as input a simulation trapdoor τ and statement ϕ and returns an argument ϕ.*

We say that $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ *is a zk-SNARK if it satisfies completeness, , succinctness, computational knowledge soundness and zero-knowledge as described below:*

- ***Completeness.*** *Completeness says that, given any true statement, an honest prover should be able to convince an honest verifier. For all $\lambda \in \mathbb{N}$, $R \in \mathcal{R}_\lambda$, $(\phi, w) \in R$*

$$\Pr\left[(\mathsf{crs}, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, R, \phi, w) : \mathsf{Ver}(\mathsf{crs}, R, \phi, w) = 1\right] \geq 1 - \mathsf{negl}(\lambda).$$

— ***Zero-Knowledge.*** *An argument is zero-knowledge if it does not leak any information besides the truth of the statement. We say that* $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ *is zero-knowledge if for all* $\lambda \in \mathbb{N}$, $(R, z) \leftarrow \mathcal{R}(1^\lambda)$, $(\phi, w) \in R$ *and all adversaries* $\mathcal{A}$

$$\Pr\left[(\mathsf{crs}, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathsf{Prove}(\mathsf{crs}, R, \phi, w) : \mathcal{A}(\mathsf{crs}, R, \phi, \tau, \pi) = 1\right] \approx_c \Pr\left[(\mathsf{crs}, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R);\ \pi \leftarrow \mathsf{Sim}(R, \tau, \phi) : \mathcal{A}(\mathsf{crs}, R, \phi, \tau, \pi) = 1\right]$$

— ***Computational Knowledge Soundness.*** *We say that* $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ *is an argument of knowledge if there is an extractor that can compute a witness whenever the adversary produces a valid argument. The extractor gets full access to the adversary's state, including any random coins. Formally, we require that for all non-uniform polynomial time adversaries* $\mathcal{A}$ *there exists a non-uniform polynomial time extractor* $\mathcal{X}_\mathcal{A}$ *such that*

$$\Pr\left[\begin{array}{c}(R, z) \leftarrow \mathcal{R}(1^\lambda); (\mathsf{crs}, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \\ ((\phi, \pi); w) \leftarrow (\mathcal{A} \| \mathcal{X}_\mathcal{A})(\mathsf{crs}, R, z) : \\ (\phi, w) \notin R \text{ and } \mathsf{Ver}(\mathsf{crs}, R, \phi, \pi) = 1\end{array}\right] \leq \mathsf{negl}(\lambda).$$

— ***Succinctness.*** *A non-interactive argument of knowledge where the verifier runs in polynomial time in* $\lambda + |\phi| + \log(|R|)$ *and the proof size is polynomial in* $\lambda + \log(|R|)$ *is called a pre-processing SNARK. If we also restrict the common reference string to be polynomial in* $\lambda + \log(|R|)$ *we say that the non-interactive argument is a fully succinct SNARK.*

In recent years, several SNARK constructions have been proposed both in the random oracle model and the standard model based on a variety of assumptions. The above definition is in the crs-model, but can be easily adapted to the random oracle model [30].

3 zkSaaS Framework

As discussed earlier, zkSaaS is a collaborative zk-SNARK [64] framework, where a client delegates the task of computing a zk-SNARK to n-servers. To realize our goal of enabling fast computation of zk-SNARKs, the resultant zkSaaS protocol must ensure that – (1) the work done by the clients is minimized and (2) the work required to compute the zk-SNARK gets divided across all servers.

RICS Format. Let us briefly recall the structure of existing zk-SNARKs. Different zk-SNARKs work with different representations of the relation R - e.g., quadratic arithmetic programs [65, 70], low-depth circuits [16, 33, 43, 74, 75, 76, 77, 81, 86], binary arithmetic circuits [39], etc. The most popular representation amongst state-of-the-art proof systems [9, 28, 50, 51] is known as the rank-1 constraint systems (R1CS) that generalizes arithmetic circuits.

Proof systems working with this representation proceed in two steps: (1) First, extend the given (short) statement-witness pair (ϕ, w) into a satisfying assignment $\mathbf{z}$ for the RICS relation. The length of this satisfying assignment is proportional to the size of the relation. (2) second, give an argument of knowledge for this satisfying assignment for the RICS relation. Step 1 is inexpensive and only requires non-cryptographic field operations, while Step 2 requires more expensive cryptographic operations and is typically the bottleneck.

Our Framework. zkSaaS is essentially a secure multiparty computation (MPC) protocol for computing zk-SNARKs, between a client and n-servers.

Client. Since Step 1 of the proof generation is inexpensive, we assume that client performs this

Definition 2 (zkSaaS). *Let λ be the security parameter, n be the number of parties, $R \in \mathcal{R}_\lambda$ be an NP-relation and $\Sigma_R = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ be a zk-SNARK for R such that: the prover computation time is $T_{\mathsf{prover}} = T_{\mathsf{field}} + T_{\mathsf{crypto}}$, where T_{field} and T_{crypto} are the times taken by the prover for the field operations and cryptographic operations, respectively; the prover space complexity is S_{prover}. Let $(\mathsf{Preprocessing}, \Pi_{\mathsf{online}})$ be a tuple defined as follows:*

- $\mathsf{Preprocessing}(\mathsf{crs}, 1^n) \to \mathsf{pre}_1, \ldots, \mathsf{pre}_n$: *This is a PPT algorithm that takes the* crs *output by* Setup *and the number of servers n as input and outputs correlated randomness pre_i for each server P_i (for $i \in [n]$).*
- $\Pi_{\mathsf{online}}(\mathsf{crs}, \phi, w, \mathsf{pre}_1, \ldots, \mathsf{pre}_n) \to \pi$: *This is an MPC protocol between a client $\mathcal{C}$ and n servers $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The client has the statement ϕ and private input w. It sends messages to each of the n servers in a single round. Given these messages, ϕ and their respective correlated randomness $\mathsf{pre}_1, \ldots, \mathsf{pre}_n$, the servers then engage in an interactive protocol amongst each other to compute a proof π.*

We say that $\Pi = (\mathsf{Preprocessing}, \Pi_{\mathsf{online}})$ is a zkSaaS *for Σ_R, if the following properties are satisfied.*

1. ***Completeness:*** *For all $(\phi, w) \in R$, the following holds:*

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{pre}_1, \ldots, \mathsf{pre}_n \leftarrow \mathsf{Preprocessing}(\mathsf{crs}, 1^n) \\ \pi \leftarrow \Pi_{\mathsf{online}}(\mathsf{crs}, \phi, w, \mathsf{pre}_1, \ldots, \mathsf{pre}_n) \end{array} \middle| \mathsf{Ver}(\mathsf{crs}, \phi, \pi) = 0 \right] \leq \mathsf{negl}(\lambda)$$

2. ***t-zero-knowledge:*** *Let $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{pre}_1, \ldots, \mathsf{pre}_n \leftarrow \mathsf{Preprocessing}(\mathsf{crs}, 1^n)$. For all semi-honest PPT adversaries $\mathcal{A}$ controlling at most a t-sized subset $\mathsf{Corr} \subset [n]$ of the servers, there exists an efficient Simulator $\mathsf{Sim}_{\mathsf{zkSaaS}}$, such that the following holds for all ϕ, w (where $b \leftarrow R(\phi, w) \in \{0, 1\}$):*

$$\left\{\mathsf{view}^{\mathcal{A}}_{\Pi_{\mathsf{online}}}[\phi, w]\right\} \approx_c \left\{\mathsf{Sim}_{\mathsf{zkSaaS}}(\mathsf{crs}, \phi, b, \{\mathsf{pre}_i\}_{i \in \mathsf{Corr}})\right\}$$

   here $\mathsf{view}^{\mathcal{A}}_{\Pi_{\mathsf{online}}}[\phi, w]$ denotes the view of $\mathcal{A}$ in an execution of $\Pi_{\mathsf{online}}(\mathsf{crs}, \phi, w, \mathsf{pre}_1, \ldots, \mathsf{pre}_n)$, and we use $\approx_c$ to denote computational indistinguishability between the two distributions.

3. ***Efficiency:*** *$\Pi = (\mathsf{Preprocessing}, \Pi_{\mathsf{online}})$ satisfy the following efficiency requirements:*

   Preprocessing: The computation complexity of the Preprocessing *algorithm is $o(T_{\mathsf{field}})$. For each $i \in [n]$, size of the correlated randomness $|\mathsf{pre}_i| \in \mathcal{O}(S_{\mathsf{prover}}/n)$.*

   Client: Computation complexity of the client $o(T_{\mathsf{field}})$ field operations.

   Special Server P_1: The first server has a computation complexity of $o(T_{\mathsf{field}})$ field operations and $\mathcal{O}(T_{\mathsf{crypto}}/n)$ cryptographic operations. It's space and communication complexities are $\mathcal{O}(S_{\mathsf{prover}})$ and $o(T_{\mathsf{field}})$, resp.

   Other Servers $\mathsf{P}_2, \ldots, \mathsf{P}_n$: All other servers have computation complexity of $o(T_{\mathsf{field}}/n)$ field and $\mathcal{O}(T_{\mathsf{crypto}}/n)$ cryptgraphic operations. Their space and communication complexities are $\mathcal{O}(S_{\mathsf{prover}}/n)$ and $o(T_{\mathsf{field}}/n)$, resp.

step and "securely" shares the resulting satisfying assignment $\mathbf{z}$ with the servers at the start of the protocol. To compute this satisfying assignment, the client essentially needs to represent the original

relation R as an arithmetic circuit C_R and compute all the values induced on the intermediate wires in circuit C_R when evaluated on inputs (ϕ, w). These intermediate values form the satisfying assignment $\mathbf{z}$ for the corresponding RICS relation. Looking ahead, in our construction, the client (pack) secret shares [38] the vector $\mathbf{z}$ with the servers.

Computing and sharing this satisfying assignment requires $\mathcal{O}(|R| \log N) \approx \mathcal{O}(|R|)$ field operations and very little space. Indeed, observe that the client can do this computation in a streaming fashion, where it computes a fraction of the circuit at a time and secret shares the resulting wire values before proceeding with evaluating the next part of the circuit, thereby minimizing the required space. However, if the client is unwilling, this computation can also be done via an MPC protocol, where the client only needs to share the original statement-witness pair (ϕ, w) with the servers, who then run a generic MPC to compute C_R and obtain shares of $\mathbf{z}$.

In this work, we focus on designing an efficient protocol for Step 2, which is the main bottleneck in the computation of existing zk-SNARKs.

Servers. Given shares of the extended witness, Step 2 is executed via an MPC protocol between the servers. We want the *total* computation and space complexity of this protocol to be asymptotically identical to that of computing the zk-SNARK by a monolithic entity.

Let the space complexity of the underlying zk-SNARK be S_{prover} and the computation complexity be T_{field} field operations and T_{crypto} cryptographic operations. Then, our zkSaaS-framework guarantees the following efficiencies: first, in terms of space complexity, one of the servers requires a $\mathcal{O}(S_{\mathsf{prover}})$ space, while all others require a $\mathcal{O}(S_{\mathsf{prover}}/n)$ space; second, in terms of computation, the cryptographic operations are almost equally divided amongst all the n servers, i.e., all the servers perform $\mathcal{O}(T_{\mathsf{crypto}}/n)$ cryptographic operations, and the remaining field operations are divided amongst the servers such that the server with more memory performs $\mathrm{o}(T_{\mathsf{field}})$[10] field operations, while the remaining $(n-1)$ low-memory servers each perform $\mathcal{O}(T_{\mathsf{field}}/n)$ field operations.

Pre-Processing. Finally, we assume that the servers get access to some correlated-randomness at the start of this MPC protocol. This *relation-independent* correlated-randomness can be generated as part of a pre-processing step with $\mathrm{o}(T_{\mathsf{field}})$ computation complexity (requiring only non-cryptographic operations). This can be either be generated by the client or by the servers themselves using a generic MPC protocol. Since the computation in this pre-processing phase is independent of the relation, it can be pre-computed by the servers during downtime.

Communication and Round Complexity. Finally, we remark that in order to minimize the overhead from communication, we want to limit the total communication to $\mathcal{O}(T_{\mathsf{field}})$ and restrict the number of rounds of interaction between the servers to a constant value. zkSaaS-framework is formalized in Definition 2.

4 Overview of Groth, Marlin and Plonk

In this section, we review the design of the prover algorithms in three widely used zk-SNARK construction: Groth16 [50], Marlin [28] and Plonk [39]. We start by discussing the key components (that are often the main bottleneck) used in the generation of Groth16-, Marlin- and Plonk-proofs.

Fast Fourier Transform, Polynomial Multiplication and Division. One of the key components used to optimize prover computation is the Fast Fourier Transform (FFT), which helps evaluate a given

[10] We note that here we are hiding a logarithmic factor in the n

polynomial at m points in $\mathcal{O}(m \log m)$ time. The prover computations also typically require additional FFT-based computations:

1. *Inverse FFT (iFFT)* is used to convert m evaluations of a polynomial to the coefficient representation of the polynomial in time $\mathcal{O}(m \log m)$.

2. *Polynomial Multiplication*, which given the coefficient representation, can just be computed using one call each to FFT and iFFT along with m field multiplications and takes $\mathcal{O}(m \log m)$ time.

3. *Polynomial Division*, which can actually be run by making two calls to polynomial multiplication– takes $\mathcal{O}(m \log m)$ time, where m is degree of the dividend polynomial. For completion, we describe the polynomial division protocol below. We want to efficiently divide a polynomial $p_1(X) \in \mathbb{F}[X]$ of degree m_1 by a polynomial $p_2(X) \in \mathbb{F}[X]$ of degree m_2 and generate the quotient and remainder polynomials. In [10], it was shown that this can be done efficiently using two sequential calls of the polynomial multiplication algorithm as described below:

   – Let $p_1(x) = \sum_{i=0}^{m_1} a_i x^i$ and $p_2(x) = \sum_{i=0}^{m_2} b_i x^i$. The goal is to find $q(x) = \sum_{i=0}^{m_1-m_2} q_i x^i$ and $r(x) = \sum_{i=0}^{m_2-1} r_i x^i$ such that $p_1(x) = q(x)p_2(x) + r(x)$. This is equivalent to computing $q_0, \cdots, q_{m_1-m_2}$ and $r_0, \cdots, r_{m_2-1}$ such that:

$$\begin{bmatrix} a_{m_1} \\ a_{m_1-1} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} b_{m_2} & 0 & \dots & \\ b_{m_2-1} & \ddots & & \\ \vdots & & & \\ & \ddots & b_{m_2} & \\ b_0 & & & \\ 0 & & b_{m_2-1} & \\ \vdots & \ddots & \vdots & \\ & & b_0 & \end{bmatrix} \begin{bmatrix} q_{m_1-m_2} \\ q_{m_1-m_2+1} \\ \vdots \\ q_1 \\ q_0 \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ r_{m_2-1} \\ \vdots \\ r_0 \end{bmatrix}$$

   – Here, we consider the first $m_1 - m_2 + 1$ equations only, in which case there are no r_i's to solve for. Let B be the $(m_1 - m_2 + 1) \times (m_1 - m_2 + 1)$ sub-matrix (with first $m_1 - m_2 + 1$ rows) of the matrix (called a toeplitz matrix) with b_i's above. Then, by computing the inverse matrix[11] B^{-1} and multiplying with the sub-matrix of a_i's with first $m_1 - m_2 + 1$ entries[12], we get $q_0, \cdots, q_{m_1-m_2}$.

   – Now, we can find $r(x)$ directly by computing $p_1(x) - q(x)p_2(x)$.

   Thus, the total time taken to compute the inverse of B and the two polynomial multiplications (one for computing $q(x)$ and the other for $r(x)$) is $\mathcal{O}(m_1 \log m_1)$.

Multi-scalar Multiplications (MSMs). Multi-scalar multiplications are of the form $\prod_{j \in [m]} (X_j)^{y_j}$, where $y_1, \dots, y_m \in \mathbb{F}$, and $X_1, \dots, X_m \in \mathbb{G}$.

Polynomial Commitments. In interactive oracle proofs (IOP)-based zkSNARKs, like Marlin and Plonk, in each round, the prover sends polynomial oracles to the verifier, which are essentially encodings of the witness, that the verifier can query. To convert these polynomial-IOPs to SNARKs, the prover commits to these polynomial oracles using a polynomial commitment scheme [56]. These commitments allow a

[11] In [61], it is shown how to compute inverse of a triangular $k \times k$ toeplitz matrix in time $O(k \log k)$. Morever, the inverse matrix is also a triangular toeplitz matrix.

[12] Here we use a standard property of the triangular toeplitz matrix, B, described by the vector $(b_k, \cdots, b_0)$: The matrix product $B(a_0, \cdots, a_k)^\top$ represents convolution of $(b_k, \cdots, b_0)$ and $(a_0, \cdots, a_k)$, which is just the polynomial multiplcation of the corresponding polynomials with these coefficients.

prover to commit to a univariate polynomial $p(X) \in \mathbb{F}[X]$ and get a com, such that the prover can later open to an evaluation of $p(X)$ at any point z, while giving a proof π of correct evaluation. Formally, a polynomial commitment scheme consists of a tuple $(\mathsf{PC.Setup}, \mathsf{PC.Commit}, \mathsf{PC.Eval})$ such that:

- $\mathsf{PC.Setup}(d) \to \mathsf{pp}$: for polynomials up to degree d, the setup generates the public parameter.
- $\mathsf{PC.Commit}(\mathsf{pp}, p, r) \to \mathsf{com}$: generates the commitment of a polynomial p, using randomness r.
- $\mathsf{PC.Eval}(\mathsf{pp}, x, y, \mathsf{com}; p, r) \to 0/1$: this is a protocol between the prover and verifier, where verifier accepts and outputs 1 if and only if com commits to a p such that $p(x) = y$. Only the prover knows p & r.

The security requirement from such commitment schemes is that the encoded polynomial, p, corresponding to the witness remains hidden from the verifier, and on the other hand, a cheating prover should not be able to convince the verifier of a wrong evaluation (in fact a stronger extraction property is needed). The formal definitions of the security guarantees needed for the SNARKs can be found in [28, 39]. Keeping in mind our goal of designing a custom-made protocol for our zkSaaS-framework, we look at the specific polynomial commitment scheme used in these SNARKs. The most popular polynomial commitment scheme is the KZG commitment [56] scheme. To compute these commitments, the prover needs to perform the following functions: (1) to generate the commitment, the prover needs to run the MSM function on d field and group elements and (2) to generate the proof, the prover needs to perform polynomial division and then invoke the MSM function on d field and group elements. We now give a detailed description of KZG commitments explaining the above observations.

KZG Commitments. [56] This polynomial commitment scheme is used both in Marlin and Plonk[13]. Consider the bilinear groups from Section 2.1 with generators $g_1 \in \mathbb{G}_1$, and $g_2 \in \mathbb{G}_2$. To commit to a polynomial $p(X) \in \mathbb{F}[X]$ of degree d, do the following:

- $\mathsf{PC.Setup}(d)$: Sample $\alpha \leftarrow\$ \mathbb{F}$ and output $\mathsf{pp} := (g_1^{\alpha}, \ldots, g_1^{\alpha^d}, g_2^{\alpha})$. The α is a trapdoor, which must be discarded for security.
- $\mathsf{PC.Commit}(\mathsf{pp}, p)$: output $\mathsf{com} = g_1^{p(\alpha)}$, which can be computed as $\prod_{i=0}^{d} (g_1^{\alpha^i})^{p_i}$, where p_i's are the coefficients of the polynomial p.
- $\mathsf{PC.Prove}(\mathsf{pp}, p, x)$: compute the quotient and remainder polynomials $q(X)$ and $r(X)$ corresponding to the division $(p(X) - p(x))/(X - x)$, and output $\pi = g_1^{q(\alpha)}$, computed as $\prod_{i=0}^{d} (g_1^{\alpha^i})^{q_i}$ where q_i's are the coefficients of $q(X)$. Note here, the remainder $r(X)$ must be zero.
- $\mathsf{PC.Check}(\mathsf{pp}, \mathsf{com}, x, y, \pi)$: outputs 1 if and only if $e(\pi, g_2^{\alpha} \cdot g_2^{-x}) = e(\mathsf{com} \cdot g_1^{-y}, g_2)$.

the PC.Eval protocol involves the prover running the PC.Prove procedure and the verifier accepting if and only if PC.Check outputs 1. The security of the scheme holds from the fact that $(p(X) - p(x))$ is divisible by $X - x$, thus making the remainder $r(X)$ a zero polynomial, if and only if $p(x) = y$. The detailed security analysis can be found in [56].

Note that, in the above KZG commitments, the underlying operations that the prover performs are:

- To generate the commitment, the prover needs to run the multi-scalar multiplication on inputs $p_0, \ldots, p_d \in \mathbb{F}$ and $g_1, g_1^{\alpha}, \ldots, g_1^{\alpha^d} \in \mathbb{G}_1$ to get com.

[13] Extensions of this protocol are used in [28, 39] and our framework can be setup for these extensions as well, but for ease of exposition, we explain the KZG scheme without these optimizations.

— To generate the proof, the prover needs to run the polynomial division to get quotient $q(X)$ and remainder $r(X)$ on dividing $p(X)-x$ by $(X-x)$. Finally the proof π is generated by again invoking the MSM function on $q_0, \ldots, q_d \in \mathbb{F}$ and $g_1, g_1^{\alpha}, \ldots, g_1^{\alpha^d} \in \mathbb{G}_1$.

Sumcheck Protocol. Marlin relies heavily on what is known as a sumcheck protocol for univariate polynomials. For a polynomial oracle $p(X) \in \mathbb{F}[X]$ sent by the prover, this involves giving a proof of the fact that evaluations of $p(X)$ on the set $S := \{1, \omega, \ldots, \omega^{m-1}\}$, sums to some value $\sigma \in \mathbb{F}$, where ω is the m-th primitive root of unity in the field $\mathbb{F}$. It was shown in [9] that $\sum_{x \in S} p(x) = \sigma$, if and only if $p(X)$ can be written as $q(X) \cdot X + \sigma/|S|$, for some $q(X) \in \mathbb{F}[X]$. Thus, the prover in the polynomial-IOP first evaluates the polynomial $q(X)$ by dividing the polynomial $p(X) - \sigma/|S|$ by X and sends $q(X)$ as an oracle. Hence, for running each sumcheck, the prover in Marlin invokes polynomial division before committing to this polynomial using the polynomial commitment.

Partial Products. In Plonk, the prover needs to compute (for reference, see round 2 of [39, Section 8.3]) partial products of the form $\prod_{i \in [j]} p(\omega^{i-1})$ for all $j \in |S|$, where $p(X) \in \mathbb{F}[X]$ is some polynomial (which in turn is some encoding of the witness), and $S := \{1, \omega, \ldots, \omega^{m-1}\}$, where ω is the m-th primitive root of unity in the field $\mathbb{F}$. The prover uses this in computing the polynomial $z(X)$ obtained by additional polynomial multiplication and addition operations and sends it as an oracle in the polynomial-IOP protocol. In the final Plonk protocol z is committed using the polynomial commitment.

4.1 Groth16, Marlin, and Plonk Provers

We now briefly describe how each of the three zk-SNARKs that we consider can be computed via some combination of the above functions.

Groth [50]. Groth16 is the smallest, non-interactive zk-SNARK, where the prover only sends 3 group elements to the verifier. This construction makes use of a structured CRS consisting of group elements proportional to the number of constraints. To generate the proof, the prover needs to compute a polyomial multiplication, polynomial division and a constant number of multi-scalar multiplications with the group elements in the CRS. As a concrete example, we give a detailed description of this proof system in Section A and explain in detail how one would compute this using our zkSaaS framework.

Marlin [28]. Marlin is a six round protocol, where overall, the prover generates the KZG polynomial commitments of 21 polynomials, and requires the following operations, each called for a small constant number of times: three sequential calls to the sumcheck protocol with each call additionally needing a call to polynomial division, all involving polynomials with a degree bound of the size of the relation, and polynomial additions. As a final step, this interactive protocol is converted to a SNARK using the Fiat-Shamir transformation.

Plonk [39]. Plonk on the other hand is a five round protocol, where overall the prover generates the KZG polynomial commitments of 9 polynomials, and requires the following operations, each called a small constant number of times, to generate these polynomials: polynomial multiplication and division involving polynomials with a degree bound of the size of the relation, partial products, and polynomial additions. This interactive protocol is also converted to a SNARK in the RO model. As a concrete example, we give a detailed description of this proof system in Section B and explain in detail how one would compute this using our zkSaaS framework.

5 Distributed Sub-Protocols for the zkSaaS Framework

For each of the sub-functions (FFT, MSM, Sum of Partial Products) discussed in section 4, we build custom MPC protocols. Looking ahead, we show how to compose these protocols, to design a zkSaaS for a specific

subclass of zk-SNARKs.

For our protocols in this section, we rely heavily on some standard sub-protocols, from the secret sharing based-MPC literature. A description of these sub-protocols can be found in Section C. For readability, we include here, a list of the ideal functionalities that these sub-protocols realize and that our custom protocols in this section invoke. f_{rand} (and f_{prand} resp.) are used for generating secret shares (and packed shares resp.) of random values (and vectors resp.). $f_{\mathsf{double-prand}}$ is used for generating two packed sharings of a random vector w.r.t., degrees D and $n-1$. f_{mult} (and $f_{\mathsf{pack-mult}}$ resp.) are used for multiplying secret shares (and packed shares resp.) of random values (and vectors resp.). f_{psstoss} is used for transforming packed shares to regular threshold shares. f_{sstopss} can be used for transforming regular shares to packed shares. Finally, f_{permute} is for permuting a set of pack secret shared vectors.

5.1 Distributed Protocol for Fast Fourier Transform

Before formally defining our distributed FFT protocol, we give an overview of our key technical ideas.

5.1.1 Overview of our Protocol

The fast Fourier transform (FFT) algorithm is a recursive divide-and-conquer algorithm that helps evaluate a polynomial at multiple points efficiently. In particular, to evaluate a polynomial $p(x) \in \mathbb{F}[X]$ of degree $m-1$ at the points $S = \{\omega^i : i \in [m]\}$, where ω is the m-th primitive root of unity in the field $\mathbb{F}$, FFT does the following: At level $i = \log m$, each of the m polynomials will be evaluated at a single point, which is the identity element in $1 \in \mathbb{F}$. Subsequently, given the evaluations at level i (for any $i \in [\log(m), 1]$), the FFT algorithm gives us evaluations at the level $i-1$ in the following way: For each $j \in [2^{i-1}]$, and each $k \in [m/2^i]$,

$$x_j^{i-1,k} := x_{2j-1}^{i,k} + \omega^{k2^{i-1}} \cdot x_{2j}^{i,k} \quad \&$$
$$x_j^{i-1,(m/2^i)+k} := x_{2j-1}^{i,k} + \omega^{((m/2^i)+k)2^{i-1}} \cdot x_{2j}^{i,k}$$

At the end ($i = 0$), the algorithm outputs all the m evaluations of p at S. We represent the recursion by the following function defined for each $i = \log m, \ldots, 1$:

$$F_{\mathsf{FFT}}^i(\{x_j^{i,k}\}_{j\in[2^i],k\in[m/2^i]}) := \{x_j^{i-1,k}\}_{j\in[2^{i-1}],k\in[m/2^{i-1}]}, \tag{1}$$

Here, the input to $F_{\mathsf{FFT}}^{\log m}$ are the values $x_j = x_j^{\log m,1}$ for each $j \in [m]$, representing the evaluations[14] of each of the m polynomials of level $i = \log m$ at the single point 1. Note that F_{FFT}^1 outputs the required evaluations $\{p(1), p(\omega), \ldots, p(\omega^{m-1})\}$. Using this notation, the FFT algorithm can be written as $F_{\mathsf{FFT}}(x_1, \ldots, x_m) = F_{\mathsf{FFT}}^1(F_{\mathsf{FFT}}^2(\ldots F_{\mathsf{FFT}}^{\log m}(x_1, \ldots, x_m)))$.

An Alternate View of the FFT algorithm. Our goal is to compute F_{FFT} via a secure MPC protocol. Notice that FFT is a logarithmic step algorithm, while we want a constant round MPC protocol for computing it. Towards designing such a protocol, we begin by making some key observations about FFT, and abstracting the main idea behind our final MPC protocol. For ease of exposition, consider an example where the input size is $m = 32$. We want to convert the linear function evaluation on each pair of values (at each recursive level i), into a linear function evaluation on a pair of vectors (of say $\ell = 4$ values) and be able to recurse on these vectors. Looking ahead, this will help us to pack share these vectors together and locally compute on them.

[14] For $p(X) = \sum_{i\in[m]} c_{i-1}x^{i-1}$, the x_i's are just a rearrangement of the c_i's, obtained by recursively reordering the c_i's as: put the even indexed terms at each level on the left and the odd indexed terms on the right. Continue the recursion for $\log m$ steps.

1. *FFT Step I:* Since $\log m = \log 32 = 5$, the inputs to the FFT algorithm will be $x_1^{5,1}, \ldots, x_{32}^{5,1}$. The next level $i = 4$ of the recursion is computed as: $x_j^{4,1} = x_{2j-1}^{5,1} + \omega^{16} \cdot x_{2j}^{5,1}$ and $x_j^{4,2} = x_{2j-1}^{5,1} + \omega^{32} \cdot x_{2j}^{5,1}$, for each $j \in [16]$. We now look at the same step as being computed on a vector instead of individual values. Suppose we group $\ell = 4$ elements to get the following 8 vectors at level $i = 5$: $\mathbf{x}_1^{5,1} = (x_1^{5,1}, x_3^{5,1}, x_5^{5,1}, x_7^{5,1})$, $\mathbf{x}_2^{5,1} = (x_2^{5,1}, x_4^{5,1}, x_6^{5,1}, x_8^{5,1}), \ldots, \mathbf{x}_8^{5,1} = (x_{26}^{5,2}, x_{28}^{5,2}, x_{30}^{5,2}, x_{32}^{5,2})$.

   Observation I. The key observation here is that, each pair of vectors $\mathbf{x}_{2j}^{5,1}$ and $\mathbf{x}_{2j-1}^{5,1}$ are used to compute two vector evaluations $\mathbf{x}_j^{4,1}$ (which uses ω^{16}) and $\mathbf{x}_j^{4,2}$ at the level $i = 4$ (which uses ω^{32}), for each $j \in [4]$. In other words:

$$\forall j \in [4],\ \mathbf{x}_j^{4,1} = \mathbf{x}_{2j}^{5,1} + \omega^{16} \cdot \mathbf{x}_{2j-1}^{5,1} \ \&\ \mathbf{x}_j^{4,2} = \mathbf{x}_{2j}^{5,1} + \omega^{32} \cdot \mathbf{x}_{2j-1}^{5,1}.$$

2. *FFT Step II:* We want to continue to compute on pairs of vectors linearly to obtain the next recursive step $i = 3$. However, note that now $\mathbf{x}_{2j-1}^{4,1}$ and $\mathbf{x}_{2j}^{4,1}$ do not contain the values that are linearly combined in the next recursive step of FFT. For instance we have vectors $\mathbf{x}_1^{4,1} = (x_1^{4,1}, x_2^{4,1}, x_3^{4,1}, x_4^{4,1})$ and $\mathbf{x}_2^{4,1} = (x_5^{4,1}, x_6^{4,1}, x_7^{4,1}, x_8^{4,1})$, while the values that we want to combine are $x_1^{4,1}$ with $x_2^{4,1}$, and $x_3^{4,1}$ with $x_4^{4,1}$.

   Observation II. The key observation that we make to resolve this issue is that, if at level $i = 5$, we had started with vectors: $\mathbf{x}_1^{5,1} = (x_1^{5,1}, x_5^{5,1}, x_9^{5,1}, x_{13}^{5,1})$, $\mathbf{x}_2^{5,1} = (x_2^{5,1}, x_6^{5,1}, x_{10}^{5,1}, x_{14}^{5,1})$, etc., then Step 1 would have led to the vectors (at $i = 4$): $\mathbf{x}_1^{4,1} = (x_1^{4,1}, x_3^{4,1}, x_5^{4,1}, x_7^{4,1})$, $\mathbf{x}_2^{4,1} = (x_2^{4,1}, x_4^{4,1}, x_6^{4,1}, x_8^{4,1})$, etc. These vectors $\mathbf{x}_1^{4,1}$ and $\mathbf{x}_2^{4,1}$ can now be combined using a linear combination, the same way as we did in step 1, to get $\mathbf{x}_1^{3,1}$ and $\mathbf{x}_1^{3,2}$ (and similarly others) of level $i = 3$. But, this reordering will only help us for this level and we run into the same issue when computing the next level $i = 2$.

MPC for FFT. The key point from observations I and II above is that in order to continue doing the FFT computations for each recursive level as a linear combination of vectors, instead of individual values, we must take the initial vectors at level $i = 5$ to be such that the values $x_j^{5,k}$'s in the same vector have the j's as far away as possible– this ensures that we push our problem down to as far a recursive layer as possible. However, even with the best ordering that we start with, we would reach a recursive level beyond which we cannot hope to compute on the vectors through a linear combination. Our MPC protocol combines the above key idea of packing the inputs into vectors such that local computations can be performed on them for as long as possible, along with additional techniques to overcome the challenge in computing the remaining recursive layers.

For m inputs, we start with a packed sharing of ℓ-sized vectors at level $i = \log m$: $\mathbf{x}_j^{\log m,1} = (x_j^{\log m,1}, x_{\frac{m}{\ell}+j}^{\log m,1}, \ldots, x_{\frac{m(\ell-1)}{\ell}+j}^{\log m,1})$, for each $j \in [m/\ell]$. As discussed above, this allows us to locally compute on the shares at each recursive layer (as in Steps 1 and 2) until level $i = \log \ell + 1$, beyond which we cannot do a local computation.

Beyond $i = \log \ell$: One approach that comes to mind is to rearrange the elements packed together (using some interaction) in such a way that a similar local computation can be done. However, one can observe that doing such a rearrangement actually leads to another problem– each pair of vectors are combined now using a vector of the ω^i's (instead of a single one), which leads to an "interactive" multiplication protocol at each level. This approach does give a feasible solution, but requires the parties to communicate at each of the remaining $\log \ell$ levels. Furthermore, the total communication in each round is $\mathcal{O}(m)$ which will become a bottleneck when dealing with a large constraint size.

We minimize the number of communication rounds and give a more efficient solution than the above by making use of our all powerful server and just two rounds of communication. On a high level, this uses the fact that FFT (and each of its recursive layer) is a linear function, i.e.: $F_{\mathsf{FFT}}^i((x_1 + r_1), \ldots, (x_m + r_m)) = F_{\mathsf{FFT}}^i(x_1, \ldots, x_m) + F_{\mathsf{FFT}}^i(r_1, \ldots, r_m)$. Suppose that the parties get packed shares of random values

$(r_1, \ldots, r_m)$ and packed shares of $(s_1, \ldots, s_m)$ generated as:

$$(s_1, \ldots, s_m) = F^1_{\mathsf{FFT}}(F^2_{\mathsf{FFT}}(\ldots F^{\log \ell}_{\mathsf{FFT}}(r_1, \ldots, r_m)))$$

Then, the packed shares of the level $i = \log \ell$ are masked using the packed shares of $(r_1, \ldots, r_m)$ locally, and sent to the powerful party P_1. Now, P_1 reconstructs, computes the remaining recursive levels of FFT until $i = 1$, and sends the packed shares of the output to all parties. By virtue of linearity, the parties can obtain packed shares of the FFT output by locally subtracting the packed shares of $(s_1, \ldots, s_m)$. This securely reduces the communication rounds to two[15].

We now present a detailed description of this protocol and show how to use the above protocol to devise distributed protocols for inverse FFT, polynomial multiplication and polynomial division.

5.1.2 Our Protocol

Our goal is to compute F_{FFT} in a distributed and secure way. Let ℓ be the packing constant. For each $i \in [m/\ell]$, let $\mathbf{x}_i = (x_i, x_{\frac{m}{\ell}+i}, \ldots, x_{\frac{m(\ell-1)}{\ell}+i})$. The ideal functionality f_{FFT} from Figure 1 takes $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_{m/\ell}]\!]$ as input and computes the packed shares of the output of the FFT algorithm described above, i.e., the evaluations of p on S. We now describe our complete protocol in Figure 2, and the ideal functionality that this protocol realizes is described in Figure 1. This protocol requires parties to start with some correlated randomness.

The functionality $f_{\mathsf{FFT}}(F_{\mathsf{FFT}})$

The functionality f_{FFT}, running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{FFT} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{FFT} receives shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_{m/\ell}]\!]$ from all parties.
- For each $j \in [m/\ell]$, f_{FFT} receives from the adversary a set of shares $\{u_{i,j}\}_{i \in \mathsf{Corr}}$.
- For each $i \in [m/\ell]$, f_{FFT} runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{x}_i]\!], t)$ to obtain $(x_i, x_{\frac{m}{\ell}+i}, \ldots, x_{\frac{m(\ell-1)}{\ell}+i})$.
- f_{FFT} computes $z_1, \ldots, z_m = F_{\mathsf{FFT}}(x_1, \ldots, x_m)$. For each $j \in [m/\ell]$, let $\mathbf{z}_j = (z_i)_{i \in [(j-1)\ell+1, j\ell]}$.
- For each $j \in [m/\ell]$, f_{FFT} samples a random sharing of $\mathbf{z}_j$, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_{i,j}\}_{i \in \mathsf{Corr}}$.
- f_{FFT} distributes the shares of $\mathbf{z}_1, \ldots, \mathbf{z}_{m/\ell}$ to the honest parties.

Figure 1: Ideal Functionality for Fast Fourier Transform

The protocol π_{FFT}

Inputs: Every party P_i holds packed shares $([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_{m/\ell}]\!])$.

Pre-Processing: For each $k \in [m/\ell]$, every party P_i holds packed shares $\langle\!\langle \mathsf{mask}^k \rangle\!\rangle$ and $[\![\mathsf{unmask}^k]\!]$. Here $\mathsf{mask}^k = \{r^k_1, \ldots, r^k_\ell\}$ are random vectors unknown to any party and $\mathsf{unmask}^k = \{s^k_1, \ldots, s^k_\ell\}$ is computed as: $\{s^k_1, \ldots, s^k_\ell\}_{k \in [m/\ell]} := F^1_{\mathsf{FFT}}(F^2_{\mathsf{FFT}}(\ldots F^{\log \ell}_{\mathsf{FFT}}(\{r^{\log \ell, k}_j\}_{j \in [\ell], k \in [m/\ell]})))$, where $r^{\log \ell, k}_j :=$

[15] A curious reader might wonder why the linearity doesn't help us use the powerful server to compute all the recursive levels. For input size m (which is as large as the constraint size), this solution leads to a $\mathcal{O}(m \log m)$ compute time for both the pre-processing step and the server time, as opposed to our demand of $\mathcal{O}(m)$ compute time for both.

r_j^k for each $k \in [m/\ell]$, $j \in [\ell]$ and the F_{FFT}^i's are as defined in equation 1.

Protocol: The parties $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ proceed as follows:

1. Set $[\![\mathbf{x}_j^{\log m,1}]\!] := [\![\mathbf{x}_j]\!]$ for each $j \in [m/\ell]$. For each $i \in [\log(m), \log(\ell)+1]$, $j \in [2^{i-1}/\ell]$, $k \in [m/2^i]$, all parties compute
$$[\![\mathbf{x}_j^{i-1,k}]\!] = [\![\mathbf{x}_{2j-1}^{i,k}]\!] + \omega^{k2^{i-1}} [\![\mathbf{x}_{2j}^{i,k}]\!]$$
$$[\![\mathbf{x}_j^{i-1,(m/2^i)+k}]\!] = [\![\mathbf{x}_{2j-1}^{i,k}]\!] + \omega^{((m/2^i)+k)2^{i-1}} [\![\mathbf{x}_{2j}^{i,k}]\!]$$
2. For each $k \in [m/\ell]$, all parties compute $\langle\!\langle \mathbf{y}_1^{\log \ell,k} \rangle\!\rangle = [\![\mathbf{x}_1^{\log \ell,k}]\!] + \langle\!\langle \mathsf{mask}^k \rangle\!\rangle$ and send them to P_1.
3. For each $k \in [m/\ell]$, P_1 runs $\mathsf{open}(\mathbb{F}, \langle\!\langle \mathbf{y}_1^{\log \ell,k} \rangle\!\rangle)$ to get $(y_1^{\log \ell,k}, \ldots, y_\ell^{\log \ell,k})$.
4. P_1 computes $\{y_1^{0,k}\}_{k\in[m]} := F_{\mathsf{FFT}}^1(F_{\mathsf{FFT}}^2(\ldots F_{\mathsf{FFT}}^{\log \ell}(\{y_j^{\log \ell,k}\}_{j\in[\ell],k\in[m/\ell]})))$.
5. For each $j \in [m/\ell]$, let $\mathbf{z}_j = (y_1^{0,k})_{k\in[(j-1)\ell+1,j\ell]}$. P_1 computes $[\![\mathbf{z}_j]\!] \leftarrow \mathsf{pshare}(\mathbb{F}, \mathbf{z}_j, t)$ and sends $[\![\mathbf{z}_j]\!]_i$ to party P_i (for each $i \in [2, n]$).
6. For each $j \in [m/\ell]$, parties compute $[\![\mathbf{out}_j]\!] = [\![\mathbf{z}_j]\!] - [\![\mathsf{unmask}^j]\!]$.

Output: Each party P_i outputs $\{[\![\mathbf{out}_j]\!]_i\}_{j\in[m/\ell]}$.

Figure 2: Distributed Protocol for Fast Fourier Transform

Lemma 1. *Protocol π_{FFT} securely computes functionality f_{FFT} (c.f. figure 1) against a semi-honest adversary who corrupts at most t parties.*

Proof. Correctness. The correctness of the protocol follows directly from the discussion in the protocol overview and the fact that every recursive step, F_{FFT}^i, of the FFT protocol is a linear function.

Security Proof. Let $\mathsf{Corr} \subset [n]$, with $|\mathsf{Corr}| = t$, be the set of corrupted parties. We show how to simulate the view of Corr in the ideal world, given the input shares $([\![\mathbf{x}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{x}_{m/\ell}]\!]_{\mathsf{Corr}})$ of the corrupted parties. The simulator $\mathcal{S}$ does the following:

— For the pre-processing, $\mathcal{S}$ samples the random vectors $\{\mathsf{mask}^k\}_{k\in[m/\ell]}$, and computes $\{\mathsf{unmask}^k\}_{k\in[m/\ell]} := F_{\mathsf{FFT}}^1(F_{\mathsf{FFT}}^2(\ldots F_{\mathsf{FFT}}^{\log \ell}(\{\mathsf{mask}^k\}_{k\in[m/\ell]})))$, as in the main protocol. Then, $\mathcal{S}$ generates the shares $\langle\!\langle \mathsf{mask}^1 \rangle\!\rangle_{\mathsf{Corr}}, \ldots, \langle\!\langle \mathsf{mask}^k \rangle\!\rangle_{\mathsf{Corr}}, [\![\mathsf{unmask}^1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathsf{unmask}^k]\!]_{\mathsf{Corr}}$, corresponding to the corrupted parties.

— For Steps 1 and 2, $\mathcal{S}$ sets $[\![\mathbf{x}_j^{\log m,1}]\!]_{\mathsf{Corr}} := [\![\mathbf{x}_j]\!]_{\mathsf{Corr}}$ for each $j \in [m/\ell]$ and locally computes: first, the recursive steps to get $[\![\mathbf{x}_1^{\log \ell,k}]\!]_{\mathsf{Corr}}$, for each $k \in [m/\ell]$; second, adds $\langle\!\langle \mathsf{mask}^k \rangle\!\rangle_{\mathsf{Corr}}$ to get $\langle\!\langle \mathbf{y}_1^{\log \ell,k} \rangle\!\rangle_{\mathsf{Corr}}$ for each $k \in [m/\ell]$.

— For Steps 3,4 and 5, $\mathcal{S}$ first generates $(y_1^{\log \ell,k}, \ldots, y_\ell^{\log \ell,k})$ for each $k \in [m/\ell]$ at random, then computes $\{y_1^{0,k}\}_{k\in[m]} := F_{\mathsf{FFT}}^1(F_{\mathsf{FFT}}^2(\ldots F_{\mathsf{FFT}}^{\log \ell}(\{y_j^{\log \ell,k}\}_{j\in[\ell],k\in[m/\ell]})))$, as in the actual protocol and sets $\mathbf{z}_j := (y_1^{0,k})_{k\in[(j-1)\ell+1,j\ell]}$ for each $j \in [m/\ell]$. Note that these intermediate values can be used when $P_1 \in \mathsf{Corr}$. Then, $\mathcal{S}$ generates $[\![\mathbf{z}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{z}_{m/\ell}]\!]_{\mathsf{Corr}}$. This can be done as the $y_j^{\log \ell,k}$'s look random to Corr, by the security of packed secret sharing.

— Finally, $\mathcal{S}$ computes the output shares of Corr as $[\![\mathbf{out}_j]\!]_{\mathsf{Corr}} := [\![\mathbf{z}_j]\!]_{\mathsf{Corr}} - [\![\mathsf{unmask}^j]\!]_{\mathsf{Corr}}$ for each $j \in [m/\ell]$.

Note here that for each step, by security of the corresponding packed secret sharing scheme (wherever mentioned) and as the random mask_j's are hidden from all parties, $\mathcal{S}$ generates a distribution that is identical to the views of Corr in the real world. Hence, the ideal world distribution $\mathsf{IDEAL}_{f_{\mathsf{FFT}},\mathsf{Corr},\mathcal{S}}(1^\lambda, (\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_{m/\ell} \rrbracket))$, corresponding the functionality f_{FFT} is identical to the real world distribution $\mathsf{REAL}_{\pi_{\mathsf{FFT}},\mathsf{Corr},\mathcal{A}}(1^\lambda, (\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_{m/\ell} \rrbracket))$. □

Sub-protocols relying on FFT. We require three distributed sub-protocols: for iFFT, polynomial multiplication and polynomial division as described below.

- *Inverse FFT (iFFT)*: The protocol π_{IFFT} is required to convert the packed shares of the evaluation representation of a polynomial back to its coefficient representation. π_{IFFT} on input $\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_{m/\ell} \rrbracket$, runs $f_{\mathsf{FFT}}(F_{\mathsf{IFFT}})$ on these inputs (i.e. replaces the use of F_{FFT} with F_{IFFT}), where $F_{\mathsf{IFFT}}(x_1, \ldots, x_m)$ does the following: first, runs F_{FFT} on $(x_1, \ldots, x_m)$ using ω^{-1} (inverse in $\mathbb{F}$) in place of ω, to get $(z_1, \ldots, z_m)$; second, outputs $-m' \cdot (z_1, \ldots, z_m)$, where m' is such that $m \cdot m' = |\mathbb{F}| - 1$.
- *Polynomial Multiplication*: The protocol π_{polymult} is used to multiply $p_1(X), p_2(X) \in \mathbb{F}[X]$, of degrees m_1 and m_2 respectively (such that $m_1 + m_2 < |\mathbb{F}| - 1$), given the packed shares of their coefficients as input. It does the following: first, use $f_{\mathsf{FFT}}(F_{\mathsf{FFT}})$ to convert and obtain packed shares $(\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_m \rrbracket)$ and $(\llbracket \mathbf{y}_1 \rrbracket, \ldots, \llbracket \mathbf{y}_m \rrbracket)$ of evaluations of $p_1(x)$ and $p_2(x)$, respectively, at m-points, where m is a divisor of $|\mathbb{F}| - 1$, such that $m > m_1 + m_2$; second, run $f_{\mathsf{pack-mult}}$, m times to get packed shares of $(\llbracket \mathbf{x}_1 \odot \mathbf{y}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_m \odot \mathbf{y}_m \rrbracket)$; finally run $f_{\mathsf{FFT}}(F_{\mathsf{IFFT}})$ on these packed shares to get the packed shares of coefficients of $p_1(X) \cdot p_2(X)$. Let the ideal functionality corresponding to polynomial multiplication be denoted by $f_{\mathsf{poly-mult}}$.
- *Polynomial Division*: This protocol $\pi_{\mathsf{polydivide}}$ takes as input the packed shares of coefficients of $p_1(X) \in \mathbb{F}[X]$ (of degree m_1), a public polynomial (known to all parties) $p_2(X) \in \mathbb{F}[X]$ (of degree $m_2 < m_1$), and outputs the packed shares of $q(X)$ (with $m_1 - m_2$ coefficients) and $r(X)$ (with $m_2 - 1$ coefficients), such that $p_1(X) = p_2(X)q(X) + r(X)$. The computation of the packed shares $q(X)$ and $r(X)$ can essentially be done by two sequential invocations of the polynomial multiplication functionality $f_{\mathsf{poly-mult}}$. Let the ideal functionality corresponding to a polynomial division be denoted by $f_{\mathsf{poly-divide}}$.

Complexity of our distributed FFT. Our protocol runs in two rounds, where in the first round each party communicates $\mathcal{O}(m/\ell)$ field elements and in the second round, party P_1 communicates $\mathcal{O}(m/\ell)$ field elements to each of the remaining parties. P_1 does $\mathcal{O}((\log \ell + \log n)m + m(\log m - \log \ell)/\ell)$ field operations and has a space complexity of $\mathcal{O}(m)$. The remaining parties perform $\mathcal{O}(m(\log m - \log \ell)/\ell)$ field operations and require $\mathcal{O}(m/\ell)$ space.

5.2 Distributed Protocol for Partial Products

Before we formally define our distributed protocol for partial products, we give an on overview of our key ideas.

5.2.1 Overview of our Protocol

We want to securely compute functions of the form $F_{\mathsf{part}}(x_1, \ldots, x_m) = (\prod_{j \in [i]} x_j)_{i \in [m]}$, in a distributed way. When computing on a single machine, this function requires computing the $x_{[1,i]} := x_1 \cdots x_i$ values for each $i \in [m]$ in a sequential order. Simply implementing this approach inside an MPC protocol will require $\mathcal{O}(m)$ rounds. Moreover, since each step only requires multiplying two values at a time (i.e., $x_{[1,i-1]}$ and x_i), it is unclear how to leverage packed sharing to get a division of work amongst the parties.

Our goal is to design a computation mechanism that is more amenable to parallelism and where we can meaningfully use an approach based on packed secret sharing.

The key idea to achieve this comes from rewriting F_{part} in the following way: $F_{\mathsf{part}}(x_i, \ldots, x_j) = (x_i, x_{[i,i+1]}, \ldots, x_{[i,j]}) = (F_{\mathsf{part}}(x_{\frac{(i-1)m}{\ell}+1}, \ldots, x_{\frac{im}{\ell}}) \cdot x_{[1, \frac{(i-1)m}{\ell}]})_{i \in [\ell]}$. Observe that the $F_{\mathsf{part}}(x_{\frac{(i-1)m}{\ell}+1}, \ldots, x_{\frac{im}{\ell}})$'s depend on disjoint subsets of the input x_i's. Thus, they can all be computed in parallel. In fact, since each of these $F_{\mathsf{part}}(x_{\frac{(i-1)m}{\ell}+1}, \ldots, x_{\frac{im}{\ell}})$ are computed identically, albeit on a different set of inputs, this is exactly the kind of SIMD computation for which packed secret sharing is most helpful.

MPC for F_{part}. We start with a packed secret sharing of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{m/\ell}$, where $\mathbf{x}_j$ is an ℓ-sized vector consisting of the j^{th} inputs i.e., $\mathbf{x}_j = (x_j, x_{\frac{m}{\ell}+j}, \ldots, x_{\frac{m(\ell-1)}{\ell}+j})$, for each $j \in [m/\ell]$. We now compute $F_{\mathsf{part}}([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_{m/\ell}]\!])$ to obtain packed shares of $(\mathbf{y}_j = (x_{[\frac{(i-1)m}{\ell}+1, \frac{(i-1)m}{\ell}+j]})_{i \in [\ell]})_{j \in [m/\ell]}$ using known techniques with $\mathcal{O}(m)$ total computation and communication.

A careful reader might have observed that while the above idea allows us to compute $\{\mathbf{y}_j\}_{j \in [m/\ell]}$ simultaneously, doing this naïvely will still require $\mathcal{O}(m/\ell)$ rounds. To avoid this, we observe that a slightly modified version of Bar-Ilan and Beaver's [5] constant-round MPC for unbounded multiplication can be used to compute this in a constant number of rounds.[16] We defer more details about this protocol and the modification to the technical sections.

Finally, to compute $F_{\mathsf{part}}(x_1, \ldots, x_m)$, given the packed secret sharings of $\{\mathbf{y}_j\}_{j \in [m/\ell]}$ from the previous step, we note that while computing these packed shares of $\{\mathbf{y}_j\}_{j \in [m/\ell]}$, the parties also inevitably end up computing a packed secret sharing of the vector $\mathbf{z} = (x_{[1,m/\ell]}, x_{[\frac{m}{\ell}+1, \frac{2m}{\ell}]}, \ldots, x_{[\frac{m(\ell-1)}{\ell}, m]})$.
Given $\{[\![\mathbf{y}_j]\!]\}_{j \in [m/\ell]}$ and $[\![\mathbf{z}]\!]$, our final step computes the shares of desired output:

1. Convert a packed sharing of $\mathbf{z}$ into regular threshold shares of the individual elements in $\mathbf{z}$, i.e., $[x_{[1,m/\ell]}], \ldots, [x_{[\frac{m(\ell-1)}{\ell}, m]}]$.

2. Use the above modified version of Bar-Ilan and Beaver's protocol on these shares to compute shares $[x_{[1,m/\ell]}], [x_{[1,2m/\ell]}] \ldots, [x_{[1,m]}]$.

3. Finally, for each $j \in [m/\ell]$, compute an inner product between $[\![\mathbf{y}_j]\!]$ and packed shares of vector $(1, x_{[1,m/\ell]}, x_{[1,2m/\ell]}, \ldots, x_{[1, \frac{m(\ell-1)}{\ell}]})$.

5.2.2 Our Protocol

In this section, we formally describe our MPC protocol to compute F_{part}. Let ℓ be the packed secret sharing constant. We assume the parties have packed secret sharings of the following vectors: Let $k = m/\ell$. For each $i \in [k]$, $\mathbf{x}_i = (x_i, x_{\frac{m}{\ell}+i}, \ldots, x_{\frac{m(\ell-1)}{\ell}+i})$. We describe the ideal functionality $f_{\mathsf{part-product}}$ in figure 3 which computes the shares of the required partial product of $(x_1, \cdots, x_m)$, when the packed shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]$ are given as input. Our complete protocol is described in figure 4.

The functionality $f_{\mathsf{part-product}}$

The functionality $f_{\mathsf{part-product}}$, running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

[16] We note that this protocol crucially relies on the fact that none of the values being multiplied are zero. Which is indeed the case (w.h.p.) for our use-case in Plonk.

- $f_{\mathsf{part-product}}$ receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- $f_{\mathsf{part-product}}$ receives shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]$ from all parties.
- $f_{\mathsf{part-product}}$ receives from the adversary a set of shares $\{u_i\}_{i \in \mathsf{Corr}}$.
- For each $j \in [k]$, $f_{\mathsf{part-product}}$ runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{x}_j]\!], D)$ to obtain $(x_j, x_{\frac{m}{\ell}+j}, \ldots, x_{\frac{m(\ell-1)}{\ell}+j})$.
- $f_{\mathsf{part-product}}$ computes $(z_1, \ldots, z_m) := (\prod_{i \in [j]} x_i)_{j \in [m]}$. For each $j \in [m/\ell]$, let $\mathbf{z}_j = (z_j, z_{\frac{m}{\ell}+j}, \ldots, z_{\frac{m(\ell-1)}{\ell}+j})$.
- $f_{\mathsf{part-product}}$ samples a random sharing of $\mathbf{z}_j$, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_{i,j}\}_{i \in \mathsf{Corr}}$.
- $f_{\mathsf{part-product}}$ distributes the shares of $\mathbf{z}_1, \ldots, \mathbf{z}_{m/\ell}$ to the honest parties.

Figure 3: Ideal Functionality for Partial Products

The protocol $\pi_{\mathsf{part-prod}}$

Inputs: Every party P_i holds packed shares of the values to be multiplied $([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!])$.

Protocol: The parties $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ proceed as follows:

1. Invoke f_{prand} $2k$ times to receive $([\![\mathbf{s}_1]\!], \ldots, [\![\mathbf{s}_k]\!], [\![\mathbf{w}_1]\!], \ldots, [\![\mathbf{w}_k]\!])$.
2. **Computing Inverse:** For each $j \in [k]$
   - Run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{s}_j]\!], [\![\mathbf{w}_j]\!])$ to receive $[\![\mathbf{v}_j]\!]$, where $\mathbf{v}_j = \mathbf{s}_j \odot \mathbf{w}_j$ and send $[\![\mathbf{v}_j]\!]$ to P_1.
   - P_1 runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{v}_j]\!], D)$ to obtain $\mathbf{v}_j$ and sends $\mathbf{v}_j$ to all other parties.
   - Compute $[\![\mathbf{s}_j^{-1}]\!] = \mathbf{v}_j^{-1} \odot [\![\mathbf{w}_j]\!]$.
3. Compute $[\![\mathbf{y}_j]\!]$ where $\mathbf{y}_j = \mathbf{x}_j \odot \mathsf{mask}_j$ by computing the following for all $j \in [k]$
   - Run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{s}_j]\!], [\![\mathbf{x}_j]\!])$, to receive $[\![\mathbf{x}_j \odot \mathbf{s}_j]\!]$.
   - Run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{x}_j \odot \mathbf{s}_j]\!], [\![\mathbf{s}_j^{-1}]\!])$ to receive $[\![\mathbf{y}_j]\!]$ and send $[\![\mathbf{y}_j]\!]$ to P_1.
   - P_1 runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{y}_j]\!], D)$ to obtain $\mathbf{y}_j$.
4. For each $j \in [k]$, P_1 computes $\mathbf{y}_{[1,j]} = \mathbf{y}_1 \odot \ldots \odot \mathbf{y}_j$. Then run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{s}_1^{-1}]\!], \mathbf{y}_{[1,j]})$ to receive $[\![\mathbf{s}_1^{-1} \odot \mathbf{y}_{[1,j]}]\!]$ and finally run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{s}_1^{-1} \odot \mathbf{y}_{[1,j]}]\!], [\![\mathbf{s}_{j+1}]\!])$ to receive $[\![\mathbf{z}_{[1,j]}]\!]$.
5. Run $f_{\mathsf{psstoss}}(\mathbb{F}, [\![\mathbf{z}_{1,k}]\!])$ to get $[h_1], \ldots, [h_\ell]$.
6. Invoke f_{rand} $2\ell + 2$ times to receive $([s_1], \ldots, [s_{\ell+1}], [w_1], \ldots, [w_{\ell+1}])$.
7. **Computing Inverse:** For each $j \in [\ell + 1]$
   - Run $f_{\mathsf{mult}}(\mathbb{F}, [s_j], [w_j])$ to receive $[v_j]$, where $v_j = s_j \cdot w_j$ and send $[v_j]$ to P_1.
   - P_1 runs $\mathsf{open}(\mathbb{F}, [v_j], t)$ to receive v_j and sends v_j to all other parties.
   - Compute $[s_j^{-1}] = v_j^{-1}[w_j]$.

8. Compute $[y_j]$ where $y_j = s_j \cdot h_j \cdot s_{j+1}^{-1}$ by computing the following for all $j \in [\ell]$
   - Run $f_{\mathsf{mult}}(\mathbb{F}, [s_j], [h_j])$, to receive $[h_j \cdot s_j]$.
   - Run $f_{\mathsf{mult}}(\mathbb{F}, [h_j \cdot s_j], [s_{j+1}^{-1}])$ to receive $[y_j]$ and send $[y_j]$ to P_1.
   - P_1 runs $\mathsf{open}(\mathbb{F}, [y_j], t)$ to receive y_j and sends y_j to all other parties.
9. For each $j \in [\ell]$ compute $y_{[1,j]} = y_1 \cdot \ldots \cdot y_j$. Then run $f_{\mathsf{mult}}(\mathbb{F}, [s_1^{-1} \cdot y_{1,\ldots,j}], [s_{j+1}])$ to receive $[q_{[1,j]}]$.
10. Run $f_{\mathsf{sstopss}}(\mathbb{F}, [1], [q_{[1,1]}], \ldots, [q_{[1,\ell-1]}])$ to get $[\![\mathbf{Q}]\!]$.
11. For each $j \in [k]$, run $f_{\mathsf{pack-mult}}(\mathbb{F}, [\![\mathbf{z}_{[1,j]}]\!], [\![\mathbf{Q}]\!])$ to obtain $[\![\mathbf{out}_j]\!]$.

Output: Each party P_i outputs $\{[\![\mathbf{out}_j]\!]_i\}_{j\in[k]}$.

Figure 4: Distributed Protocol for Computing Partial Products

Lemma 2. *Protocol $\pi_{\mathsf{part-prod}}$ securely computes functionality $f_{\mathsf{part-product}}$ (c.f. figure 3) in the $f_{\mathsf{prand}}, f_{\mathsf{pack-mult}}, f_{\mathsf{rand}}, f_{\mathsf{mult}}, f_{\mathsf{psstoss}}, f_{\mathsf{sstopss}}$-hybrid model against a semi-honest adversary who corrupts at most t parties.*

Proof. Correctness. The correctness of the protocol follows directly from the correctness of the underlying functionalities, $f_{\mathsf{prand}}, f_{\mathsf{pack-mult}}, f_{\mathsf{rand}}, f_{\mathsf{mult}}, f_{\mathsf{psstoss}}, f_{\mathsf{sstopss}}$ and the discussion in the protocol overview.

Security Proof. The security of this protocol only holds when $x_j \neq 0$, for all $j \in [m]$, except with a negligible probability.[17] Let $\mathsf{Corr} \subset [n]$, with $|\mathsf{Corr}| = t$, be the set of corrupted parties. We show how to simulate the view of Corr in the ideal world, given the input shares $([\![\mathbf{x}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{x}_k]\!]_{\mathsf{Corr}})$ of the corrupted parties. The simulator $\mathcal{S}$ does the following:

- For Step 1, the parties run $2k + 2$ invocations of the ideal functionality f_{prand}, while $\mathcal{S}$ samples vectors of random values $\mathbf{s}_1, \ldots, \mathbf{s}_{k+1}, \mathbf{w}_1, \ldots, \mathbf{w}_{k+1}$ to generate the packed shares and get $[\![\mathbf{s}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{s}_{k+1}]\!]_{\mathsf{Corr}}, [\![\mathbf{w}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{w}_{k+1}]\!]_{\mathsf{Corr}}$, corresponding to the corrupted parties.
- For Step 2, the parties run the ideal functionality $f_{\mathsf{pack-mult}}$. $\mathcal{S}$ can generate $\mathbf{v}_j = \mathbf{s}_j \odot \mathbf{w}_j$, for each $j \in [k+1]$, generate the $\mathbf{s}_j^{-1}$'s, and get $[\![\mathbf{s}_1^{-1}]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{s}_{k+1}^{-1}]\!]_{\mathsf{Corr}}$.
- For Step 3, the parties call the $f_{\mathsf{pack-mult}}$-functionality twice. $\mathcal{S}$ can just set the shares of outputs for each of these calls as shares of some random vectors (by security of the packed secret sharing) and picks $\mathbf{y}_j$ at random for each $j \in [k]$ (since the s_j's look random to Corr, the y_j's also look random to Corr. This is true only because the x_j's are all non-zero!).
- For Step 4, for each $j \in [k]$, $\mathcal{S}$ can generate $\mathbf{y}_{[1,j]} = \mathbf{y}_1 \odot \ldots \odot \mathbf{y}_j$, compute $\mathbf{s}_1^{-1} \odot \mathbf{y}_{[1,j]}$ and $\mathbf{z}_{[1,j]} = \mathbf{s}_1^{-1} \odot \mathbf{y}_{[1,j]} \odot \mathbf{s}_{j+1}$, and get $[\![\mathbf{s}_1^{-1} \odot \mathbf{y}_{[1,j]}]\!]_{\mathsf{Corr}}$ and $[\![\mathbf{z}_{[1,j]}]\!]_{\mathsf{Corr}}$.
- For Step 5, $\mathcal{S}$ can output $[h_1]_{\mathsf{Corr}}, [h_\ell]_{\mathsf{Corr}}$, as shares of random values (by security of secret sharing).
- For Steps 6-9, $\mathcal{S}$ does exactly the same thing as it did for Steps 1-4, except for replacing the role of packed shares with regular shares, and the parties invoke the corresponding functionalities on regular shares.

[17] Looking ahead at our application of this protocol, we use this sub-protocol only for inputs that are all non-zero w.h.p.

— Finally, for Steps 10 and 11, $\mathcal{S}$ sets $[\![\mathbf{Q}]\!]_{\mathsf{Corr}}$ and the final output shares of Corr, $[\![\mathsf{out}_j]\!]_{\mathsf{Corr}}$ for each $j \in [k]$, to be shares of random values (by security of packed secret sharing).

Note here that for each step, by the security of the secret sharing and packed secret sharing schemes (wherever mentioned) and since the y_j's look random to Corr (under our assumption that the x_j's are all non-zero w.h.p.), $\mathcal{S}$ generates a distribution that is identical to the views of Corr in the real world, assuming that the x_j's are all non-zero. Hence, the ideal world distribution $\mathsf{IDEAL}_{f_{\mathsf{part-product}},\mathsf{Corr},\mathcal{S}}(1^\lambda, ([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]))$, corresponding the functionality $f_{\mathsf{part-product}}$ is statistically close to $\mathsf{REAL}_{\pi_{\mathsf{part-prod}},\mathsf{Corr},\mathcal{A}}(1^\lambda, ([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]))$, where the parties are all given access to the ideal functionality $f_{\mathsf{pack-mult}}$. □

Complexity of our distributed Partial Products Protocol. Our protocol runs in constant rounds, where each of the small servers communicates $\mathcal{O}(m/\ell)$ field elements. They perform $\mathcal{O}(m/\ell)$ field operations and require a space complexity of $\mathcal{O}(m/\ell)$. While the big server, P_1 has a space complexity of $\mathcal{O}(m)$ and performs $\mathcal{O}(m)$ field operations and communicates $\mathcal{O}(m)$ field elements. This is because of the sub-protocol that we adapt from Bar-Ilan and Beaver's [5] constant-round MPC for unbounded multiplication. Since our distributed FFT protocol already assumes that one of the servers has more memory and computational resources, this is the version we use in our implementation of Plonk. However, in Section D we present an alternate protocol for distributed computation of partial products, where the total work gets equally divided amongst all parties. In particular, *each server* in that protocol requires $\mathcal{O}(m/\ell)$ field operations, a space complexity of $\mathcal{O}(m/\ell)$, and each server communicates $\mathcal{O}(m/\ell + n)$ field elements. We note however, that unlike all of our other protocols (where the weak servers only need to communicate with the client and the large server), that protocol requires point-to-point communication between every pair of servers.

5.3 Distributed Protocol for Multi-Scalar Multiplications

Before we formally describe our distributed protocol for multi-scalar multiplication protocol, we give an overview of our key ideas.

5.3.1 Overview of our Protocol

Polynomial-based secret sharing schemes typically only support arithmetic operations over a finite field. Several zk-SNARKs perform many elliptic curve group operations, such as multiplying group elements or group exponentiations. Representing these group operations as an arithmetic circuit over a finite field and computing it inside an MPC protocol is not feasible.

Prior works [73, 64] have explored generalizations of polynomial-based secret sharing schemes for group operations. Let $\mathbb{G}$ be a group of order p, with generator g, such that each element $A \in \mathbb{G}$ can be represented as g^a, where $a \in \mathbb{Z}_p$. The main idea in these works is to first compute secret shares (say $s_1, \ldots, s_n$) of the field element a, and then compute the shares of A as $g^{s_1}, \ldots, g^{s_n}$. This allows us to perform arithmetic field operations in the exponent which can be used for group exponentiation and for multiplying group elements.

— *Addition in the exponent.* Given packed secret shares of another vector $\mathbf{B} = (B_1, \ldots, B_\ell) \in \mathbb{G}^\ell$, each party P_i can locally multiply their shares $[\![\mathbf{A}]\!]_i \cdot [\![\mathbf{B}]\!]_i$, to get a valid packed secret sharing of $\mathbf{C} = (A_1 \cdot B_1, \ldots, A_\ell \cdot B_\ell)$.

— *Multiplication in the exponent.* Given packed secret shares of another vector of field elements $\mathbf{b} = (b_1, \ldots, b_\ell) \in \mathbb{Z}_p^\ell$, each party P_i can locally compute $[\![\mathbf{A}]\!]_i^{[\![\mathbf{b}]\!]_i}$ to get a packed secret sharing of $\mathbf{C} = (A_1^{b_1}, \ldots, A_\ell^{b_\ell})$. However, in this case, since the shares of $\mathbf{a}$ and $\mathbf{b}$ get multiplied in the exponent, the

degree of the resulting sharing will be twice that of the original sharings. To reduce the degree, we can use the standard ideas for degree reduction, albeit in the exponent.

MPC for MSM. Given the above observations, our idea for computing multi-scalar multiplications of the form $F_{\mathsf{MSM}}(A_1, b_1 \ldots, A_m, b_m) = \prod_{i\in[m]} A_i^{b_i}$ is to first observe that this decomposes as is quite intuitive. Observe, this computation can be decomposed as: $\prod_{i\in[\ell]} \left(F_{\mathsf{MSM}}(A_i, b_i, A_{\ell+i}, b_{\ell+i}, \ldots, A_{(\frac{m}{\ell}-1)\ell+i}, b_{(\frac{m}{\ell}-1)\ell+i})\right)$. This is essentially equivalent to computing ℓ instances of F_{MSM} in parallel and then multiplying the ℓ outputs. We compute this using PSS as follows:

1. Assuming that the parties have packed secret shares of vectors $\mathbf{A}_j = (A_{(j-1)\ell+i})_{i\in[\ell]}$ and $\mathbf{b}_j = (b_{(j-1)\ell+i})_{i\in[\ell]}$ for each $j \in [m/\ell]$, the parties compute F_{MSM} function on these packed shares to get packed shares of a vector $\mathbf{C}$.

2. Convert $[\![\mathbf{C}]\!]$ to regular threshold shares $[C_1], \ldots, [C_\ell]$ of the individual elements in $\mathbf{C}$.

3. Finally, the parties locally multiply these shares to get a sharing of the desired output.

5.3.2 Our Protocol

In this section, we formally describe our MPC protocol to compute the multi-scalar multiplication, F_{MSM}. Let ℓ be the packed secret sharing constant. We assume that the parties have packed secret shares of the following vectors: Let $k = m/\ell$. For each $i \in [k]$ $\mathbf{y}_i = (y_{(i-1)\ell+1}, \ldots, y_{i\ell})$ and $\mathbf{X}_i = (X_{(i-1)\ell+1}, \ldots, X_{i\ell})$. We describe the ideal functionality f_{MSM} in Figure 5, which computes the shares of the required multi-scalar multiplication of $(y_1, \ldots, y_m)$ and $(X_1, \ldots, X_m)$, given the input $([\![\mathbf{y}_1]\!], \ldots, [\![\mathbf{y}_k]\!]), ([\![\mathbf{X}_1]\!], \ldots, [\![\mathbf{X}_k]\!])$. Our complete protocol is described in Figure 6.

The functionality f_{MSM}

The functionality f_{MSM}, running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{MSM} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{MSM} receives shares $([\![\mathbf{y}_1]\!], \ldots, [\![\mathbf{y}_k]\!]), ([\![\mathbf{X}_1]\!], \ldots, [\![\mathbf{X}_k]\!])$ from all parties.
- f_{MSM} receives from the adversary a set of shares $\{u_i\}_{i\in\mathsf{Corr}}$.
- For each $j \in [k]$, f_{MSM} runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{X}_j]\!], t)$ to obtain $(X_{(i-1)\ell+1}, \ldots, X_{i\ell})$ and $\mathsf{open}(\mathbb{F}, [\![\mathbf{y}_j]\!], t)$ to obtain $(y_{(i-1)\ell+1}, \ldots, y_{i\ell})$.
- f_{MSM} computes $\mathsf{out} = \prod_{j\in[m]}(X_j)^{y_j}$.
- f_{MSM} samples a random sharing of out, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_i\}_{i\in\mathsf{Corr}}$.
- f_{MSM} distributes the shares of out to the honest parties.

Figure 5: Ideal Functionality for Multi-Scalar Multiplications

The protocol π_{MSM}

Inputs: Every party P_i holds shares of the values $[\![\mathbf{y}_1]\!], \ldots, [\![\mathbf{y}_k]\!]), [\![\mathbf{X}_1]\!], \ldots, [\![\mathbf{X}_k]\!])$.
Protocol: The parties $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ proceed as follows:

1. Parties run $f_{\mathsf{double-prand}}$ to receive $([\![\mathbf{r}]\!], \langle\!\langle \mathbf{r} \rangle\!\rangle)$.
2. Parties run $f_{\mathsf{psstoss}}(\mathbb{F}, [\![\mathbf{r}]\!])$ to receive $[r_1], \ldots, [r_\ell]$.
3. Each party P_i computes $\langle\!\langle \mathbf{C} \rangle\!\rangle_i = \prod_{j \in [k]} [\![\mathbf{X}_j]\!]_i^{[\![\mathbf{y}_j]\!]_i}$.
4. Each party P_i computes $\langle\!\langle \mathbf{D} \rangle\!\rangle_i = \langle\!\langle \mathbf{C} \rangle\!\rangle_i \cdot g^{\langle\!\langle \mathbf{r} \rangle\!\rangle_i}$.
5. Each party P_i sends $\langle\!\langle \mathbf{D} \rangle\!\rangle_i$ to P_1.
6. P_1 runs $\mathsf{open}(\mathbb{F}, \langle\!\langle \mathbf{D} \rangle\!\rangle, n-1)$ to receive $\mathbf{D} = (D_1, \ldots, D_\ell)$.
7. P_1 computes $E = \prod_{j \in [\ell]} D_j$ and sends E to all other parties.
8. Parties compute $[\mathsf{out}] = \frac{E}{\prod_{j \in [\ell]} g^{[r_j]}}$

Output: Each party P_i outputs $[\mathsf{out}]_i$.

Figure 6: Multi-Scalar Multiplications

Lemma 3. *Protocol π_{MSM} securely computes functionality f_{MSM} (c.f. figure 5) in the $f_{\mathsf{double-prand}}$,$f_{\mathsf{psstoss}}$-hybrid model against a semi-honest adversary who corrupts at most t parties.*

Proof. Correctness. The correctness of the protocol follows directly from the correctness of the underlying functionalities $f_{\mathsf{double-prand}}$,$f_{\mathsf{psstoss}}$ and from the discussion in the protocol overview.
Security Proof. Let $\mathsf{Corr} \subset [n]$, with $|\mathsf{Corr}| = t$, be the set of corrupted parties. We show how to simulate the view of Corr in the ideal world, given the input shares $([\![\mathbf{y}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{y}_k]\!]_{\mathsf{Corr}}), ([\![\mathbf{X}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{X}_k]\!]_{\mathsf{Corr}})$ of the corrupted parties. The simulator $\mathcal{S}$ does the following:

— For Steps 1 and 2, $\mathcal{S}$ picks a random vector $\mathbf{r} = (r_1, \ldots, r_\ell)$ and generates the shares to get $[\![\mathbf{r}]\!]_{\mathsf{Corr}}, \langle\!\langle \mathbf{r} \rangle\!\rangle_{\mathsf{Corr}}$ and $[r_1]_{\mathsf{Corr}}, \ldots, [r_\ell]_{\mathsf{Corr}}$.

— For Steps 3 and 4, $\mathcal{S}$ does local computations on $([\![\mathbf{y}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{y}_k]\!]_{\mathsf{Corr}}), ([\![\mathbf{X}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{X}_k]\!]_{\mathsf{Corr}})$ and $\langle\!\langle \mathbf{r} \rangle\!\rangle_{\mathsf{Corr}}$, to generate $\langle\!\langle \mathbf{D} \rangle\!\rangle_{\mathsf{Corr}}$.

— For Steps 6 and 7, $\mathcal{S}$ can pick $\mathbf{D}$ at random (each component picked as a random group element), which is directly needed if $P_1 \in \mathsf{Corr}$, and evaluate $E = \prod_{j \in [\ell]} D_j$. This can be done as the vector $\mathbf{D}$ looks random to Corr by the security of packed secret sharing.

— Finally, the output shares of Corr are evaluated as $[\mathsf{out}]_{\mathsf{Corr}} := E / g^{\sum_{j \in [\ell]} [r_j]_{\mathsf{Corr}}}$.

Note here that for each step, by security of the corresponding packed secret sharing scheme (wherever mentioned) and as the random $\mathbf{r}$ is hidden from all parties, $\mathcal{S}$ generates a distribution that is identical to the views of Corr in the real world. Hence, the ideal world distribution $\mathsf{IDEAL}_{f_{\mathsf{MSM}},\mathsf{Corr},\mathcal{S}}(1^\lambda, ([\![\mathbf{y}_1]\!], \ldots, [\![\mathbf{y}_k]\!]), ([\![\mathbf{X}_1]\!], \ldots, [\![\mathbf{X}_k]\!]))$, corresponding the functionality f_{MSM} is identical to the real world distribution $\mathsf{REAL}_{\pi_{\mathsf{MSM}},\mathsf{Corr},\mathcal{A}}(1^\lambda, ([\![\mathbf{y}_1]\!], \ldots, [\![\mathbf{y}_k]\!]), ([\![\mathbf{X}_1]\!], \ldots, [\![\mathbf{X}_k]\!]))$. □

Complexity of distributed MSM. Our protocol runs in constant rounds, where each party communicates $\mathcal{O}(1)$ group elements. All parties perform $\mathcal{O}(m/\ell)$ group exponentiations and have a space complexity of $\mathcal{O}(m/\ell)$.

6 zkSaaS for Admissible zk-SNARKs

In this section, we formally define a notion of admissible zk-SNARKs and show that our techniques from Section 5 can be used to obtain a zkSaaS for them.

Admissible zk-SNARKs. We start by formalizing a class of zk-SNARKs that are amenable to our zkSaaS framework and refer to them as admissible zk-SNARKs. Informally speaking, we say that a zk-SNARK with computation complexity $T_{\mathsf{field}} + T_{\mathsf{crypto}}$ is admissible if the prover algorithm is composed of some combination of a subset or all of the following six types of operations on the satisfying assingment $\mathbf{z}$ for the RICS relation R — (1) multi-scalar multiplications (MSMs), (2) Fast Fourier Transforms (FFT), (3) sum of partial-products, (4) multiplication/Hadamard product, (5) additions and (6) permutations.

To formally capture this, our initial idea is to say that the prover algorithm in admissible zk-SNARKs can be represented as a polynomial-sized circuit consisting of special gates with "multi-ary" inputs and outputs, where each of these special gates correspond to one of the above six operations. However, this is alone is not sufficient. To capture the efficiency requirements of a zkSaaS (as discussed in Section 3), we need to further restrict the number of times a particular gate with a certain ari-ty can appear in this circuit.

Indeed, consider for instance a circuit, where two-input multiplication gates appear $\mathcal{O}(T_{\mathsf{field}})$ times in the circuit. Since the only distributed protocol that we can use for evaluating such gates is π_{mult} (c.f. Figure 20), which requires a total communication and computation of $\mathcal{O}(n)$, the total communication and computation incurred in evaluating $\mathcal{O}(T_{\mathsf{field}})$ such gates would be $\mathcal{O}(n.T_{\mathsf{field}})$. This clearly violates the efficiency requirement of zkSaaS. Therefore, we must limit the number of such gates with low-ary inputs that appear in this circuit to ensure that the cost of computing them does not surpass the asymptotic bound that we have on the total computation complexity of zkSaaS. More concretely, in order to use our packed secret sharing based sub-protocols, we must limit the number of gates with $o(n)$ inputs. Hence, we define the notion of admissibility w.r.t. the number of parties n. This is formalized in Definition 3.

Definition 3 (n-Admissible zk-SNARKs)**.** *Let λ be the security parameter, $R \in \mathcal{R}_\lambda$ be an NP-relation and $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ be a zk-SNARK for R. We say that Σ is n admissible if $n < T_{\mathsf{field}}$, $n < T_{\mathsf{crypto}}$ and the* Prove *algorithm can be represented as a circuit* C *comprising of gates implementing the following functionalities:*

Multi-Scalar Multiplication: $F_{\mathsf{MSM}}(y_1, \ldots, y_m, X_1, \ldots, X_m) = \prod_{j\in[m]}(X_j)^{y_j}$.
Fast Fourier Transform: $F_{\mathsf{FFT}}(x_1, \ldots, x_m) = F^1_{\mathsf{FFT}}(F^2_{\mathsf{FFT}}(\ldots F^{\log m}_{\mathsf{FFT}}(x_1, \ldots, x_m)))$, *where each* F^i_{FFT} *is the recursive function described in equation 1.*
Sum of Partial Products: $F_{\mathsf{part-prod}}(x_1, \ldots, x_m) = \sum_{j\in[m]} \prod_{i\in[j]} x_i$.
Multiplication/Hadamard Product: $F_{\mathsf{prod}}(x_1, \ldots, x_m, y_1, \ldots, y_m) = (x_1 \cdot y_1), \ldots, (x_m \cdot y_m)$.
Addition: $F_{\mathsf{sum}}(x_1, \ldots, x_m, y_1, \ldots, y_m) = (x_1 + y_1), \ldots, (x_m + y_m)$.
Permutation: $F_{\mathsf{perm}}(x_1, \ldots, x_m) := (x_{\mathsf{perm}(1)}, \ldots, x_{\mathsf{perm}(m)})$, *where* perm *is a permutation function on* $[m]$.

Furthermore, the total number of MSM gates with $m \in o(n)$ inputs is limited to $\mathcal{O}(T_{\mathsf{crypto}}/n)$

and all other types of gates $m \in o(n)$ inputs are limited to $\mathcal{O}(T_{\mathsf{crypto}}/n)$.

We now present our main composition theorem and show that the three zk-SNARKs that we discussed in Section 4.1 are admissible.

Theorem 1. *Let λ be the security parameter, $R \in \mathcal{R}_\lambda$ be an NP-relation and $\Sigma = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Ver}, \mathsf{Sim})$ be an n-admissible zk-SNARK for relation R. Then, there exists a secure n-server* zkSaaS *Π for Σ, which securely computes* Prove *in the $f_{\mathsf{double-prand}}, f_{\mathsf{prand}}, f_{\mathsf{pack-mult}}, f_{\mathsf{rand}}, f_{\mathsf{psstoss}}, f_{\mathsf{mult}}, f_{\mathsf{sstopss}}, f_{\mathsf{part-product}}, f_{\mathsf{poly-mult}}, f_{\mathsf{poly-divide}}, f_{\mathsf{FFT}}, f_{\mathsf{MSM}}, f_{\mathsf{permute}}$-hybrid model*[18].

Proof of Theorem 1. As defined in Defintion 3, the Prove function/algorithm in Σ only comprises of the following functionalities: FFT, MSM, sum of partial products, multiplication/hadamard product, addition and permutation. It is easy to see that for computing such a function, we can design a non-interactive MPC protocol Π where the parties only make oracle access to the ideal functionalities $f_{\mathsf{double-prand}}, f_{\mathsf{prand}}, f_{\mathsf{pack-mult}}, f_{\mathsf{rand}}, f_{\mathsf{mult}}, f_{\mathsf{psstoss}}, f_{\mathsf{sstopss}}, f_{\mathsf{FFT}}, f_{\mathsf{MSM}}, f_{\mathsf{permute}}, f_{\mathsf{part-product}}, f_{\mathsf{poly-mult}}, f_{\mathsf{poly-divide}}$, besides performing local operations on the shares of the input (which in this case is a sharing of the satisfying assignment $\mathbf{z}$ for the RICS representation of the relation). The security of this protocol follows trivially.

Efficiency: Each of these ideal functionalities can be realized using the sub-protocols described in Sections 5 and C. Recall that for computing field operation based functions where the number of inputs are limited to $m \in o(n)$, we cannot leverage packed secret sharing to get computational savings and based on current techniques, such functions must inevitably be computed with a total computation of $\mathcal{O}(n)$. Since the number of gates with $m \in o(n)$-inputs is limited to $\tilde{\mathcal{O}}(T_{\mathsf{field}}/n)$, the total computation required for computing these gates is at most $\tilde{\mathcal{O}}(T_{\mathsf{field}})$. The same argument extends to MSM gates with fewer inputs. For all the remaining gates, we can use our packed secret sharing based subprotocols that allow division of work amongst the servers and comply with the efficiency requirements of our zkSaaS framework. □

Instantiation. We note that, as discussed in Section 4.1, the provers of Groth16 [50], Marlin [28] and Plonk [39] only call the functionalities listed in defintion 3. Furthermore, the number of gates with $o(n)$-inputs in each of these is at most a constant number. Hence, using Theorem 1 we can directly get a zkSaaS for Groth16, Marlin and Plonk. Note here that Marlin and Plonk are described as interactive protocols, but as mentioned in Section 4.1 they can be converted to non-interactive protocols in the random oracle model, by using the Fiat-Shamir transformation. Specifically, this would require the prover to make random oracle queries on parts of the transcript– in our zkSaaS this translates to each party reconstructing shares of the transcript, to make these random oracle queries locally. The protocol clearly remains zero-knowledge.

7 Implementation and Evaluation

To evaluate the concrete performance of our techniques, we implemented a proof-of-concept zkSaaS framework supporting Groth16 [50] and Plonk [39] (the protocol described in Section A and B) in Rust. We use the `arkworks` [3] library for finite field, pairing-friendly curves and, FFT implementations and the `mpc-net` crate from the collaborative-snarks implementation [2] to facilitate communication between parties. Our code is available on Github.[19] All of our experiments are run on the Google Cloud Platform (GCP) using two types of machines – all servers with low memory requirements are custom `N1` instances with 1 vCPU and 2GB of memory, while the powerful server is a custom `N1` instance with 96 vCPUs and 128 GB of RAM.

[18] A formal description of these ideal functionalities can be found in Sections 5 and C.

[19] https://github.com/guruvamsi-policharla/zksaas

We compare the performance of our zkSaaS protocol against a prover running locally on an `N1` instance with 1 vCPUs and 4 GB of RAM, emulating a mid tier consumer laptop and hence refer to this as the consumer machine. Our VM configuration choices aim to reflect realistic deployment scenarios for zkSaaS, where one powerful instance is hired to aid many weak — volunteer-run nodes, often on outdated and older hardware. Throughout the analysis when the zkSaaS protocol is run with n parties, the corruption threshold is set to be $t = n/4 - 1$ and the number of secrets packed together to be $\ell = n/4$. All numbers reported are the average of five trials.

We view our implementation as a proof-of-concept to estimate running times and network delays and do not implement multi-threading on the powerful server. In a production-level implementation, we expect the powerful server to use multi-threading in the FFT protocol which includes packing/opening shares and communicating with parties. The data we present takes this into account by dividing the time taken during computation by the number of threads on the server and the time spent during communication by $\min(n, \#\text{ of threads})$. Finally, we implement a variant of our distributed partial products protocol which avoids *all-to-all* communication but the king party does linear work. This does not affect our speedup as we assume the king is multi-threaded and simplifies the communication to a *star* like structure.

Pre-processing. Our goal is to analyze the *online* work carried out by the servers. In both the single prover baseline and the zkSaaS protocol, we do not benchmark the time taken to prepare the witness, since we assume this is given to the zkSaaS servers by the clients. We do not evaluate pre-processing as it can be carried out during periods of low demand when spare compute and bandwidth are available. Prior work [64] also omits this analysis.

Evaluation. We now evaluate the performance of our zkSaaS framework against a prover running locally on a consumer machine for Groth16 and Plonk. In particular, we aim to answer three main questions: (1) How does the performance of zkSaaS compare to a Groth16/Plonk prover running on a single consumer machine? We are interested in two main metrics — (a) the largest number of constraints that can be supported without memory exhaustion and (b) the time required to generate the proof. (2) Does the performance of zkSaaS improve as we increase the number of servers? (3) How does the performance of zkSaaS vary with network bandwidth?

Varying Constraints. For our first experiment, we run benchmarks on a high speed network (4Gbps). We compare the running time of zkSaaS with 128 parties against a local execution of Groth16/Plonk prover on a single consumer machine, by varying the number of constraints. We summarize our results in Figure 7.

The performance of our zkSaaS for larger constraints is approximately 22× better better than the consumer machine. Here, we incur a loss from the theoretically expected savings of 32× due to a few factors:

— Pippenger's algorithm [67] provides a way to compute a multi-scalar multiplication $\Pi_{i=1}^{N} g_i^{\alpha_i}$ using $O(N/\log N)$ group operations as opposed to $O(N)$ in the naive strategy. However, Pippenger's algorithm is not conducive to our packed secret sharing MPC techniques as it lacks sufficient SIMD structure. With an optimal division of work, each weak server would carry out $O(N/(\ell \cdot \log N))$ group operations. While we do not attain an optimal division of work, we come very close with a per server complexity of $O(N/\ell \cdot \log N/\ell)$. In fact, since the constants inside the big O notation are the same, we can theoretically predict the percentage *loss* in performance under infinite bandwidth conditions by simply dividing the two asymptotics. As can be seen in Figure 8, a 4Gbps connection closely emulates infinite bandwidth and our implementation indeed comes very close to the theoretical prediction.

— In our distributed FFT/partial product protocol, during degree reduction, the King unpacks and repacks shares adding additional overhead. Our FFT implementation is also not as carefully optimized as the `arkworks` implementation which is used by the consumer machine.

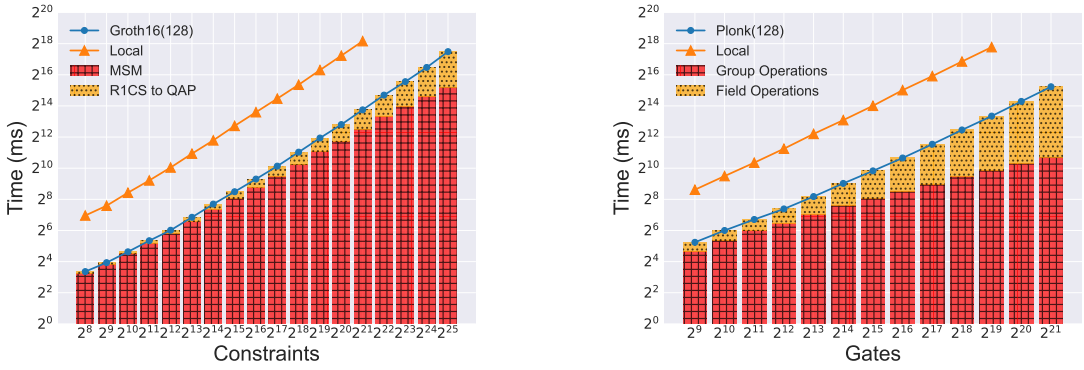

Figure 7: *Comparison between proof generation time for a local prover (Groth16 and Plonk resp.) run on the consumer machine against that of the* zkSaaS *protocol with 128 servers on a 4 Gigabit link. The bar graph indicates the* fraction *of time spent computing Field Operations (T_{field}) vs Group Operations (T_{crypto}). Missing data points on the local curve indicates memory exhaustion.*

Also, observe in Figure 7, that the fraction of time spent computing the field operations (refered to as R1CS to QAP mapping in the case of Groth16) increases with the number of constraints. This is because the FFT operation is asymptotically more expensive ($O(m \log m)$) than the MSM ($O(m/\log m)$).

Varying Parties and Network Speeds. For our next experiment, we show how zkSaaS scales as the number of parties increase, at varying network speeds. The dotted black line denotes the time taken by a local prover. We simulate slower connections by scaling up the time spent on communication by the network slowdown factor, comparing this to an implementation of Groth16/Plonk on a single consumer machine and present our findings in Figure 8. Even at lower network speeds (64 Mbps), we observe that the performance degradation is $\approx 2\times$.

Discussion on Financial Costs. We now estimate the costs of providing zk-SNARKs as a service. The powerful VM costs has a spot pricing of \$0.79/hr[20] and cross continent egress traffic pricing is \$0.08/GB[21]. Being very conservative, our estimates show that with 128 parties, generating a Groth16 proof for an R1CS instance of size 2^{19} takes under 1 minute on a 4-Gbps link and under 5 minutes on a 64 Mbps link, with the total outgoing communication from the server < 1.85GB. Hence, creating this proof would cost < 18 cents with a 4 Gbps link and < 23 cents on a 64 Mbps link.

Acknowledgements

Sanjam Garg, Guru-Vamsi Policharla and Sruthi Sekar are supported in part by DARPA under Agreement No. HR00112020026, AFOSR Award FA9550-19-1-0200, NSF CNS Award 1936826, and research grants by the Sloan Foundation, and Visa Inc. Guru-Vamsi Policharla is also supported by the UC Berkeley Center for Long-Term Cybersecurity. Abhishek Jain is supported in part by NSF CNS-1814919, NSF CAREER 1942789, Johns Hopkins University Catalyst award, AFOSR Award FA9550-19-1-0200, and research gifts from Ethereum, Stellar and Cisco.

[20] https://cloud.google.com/compute/vm-instance-pricing
[21] https://cloud.google.com/compute/network-pricing

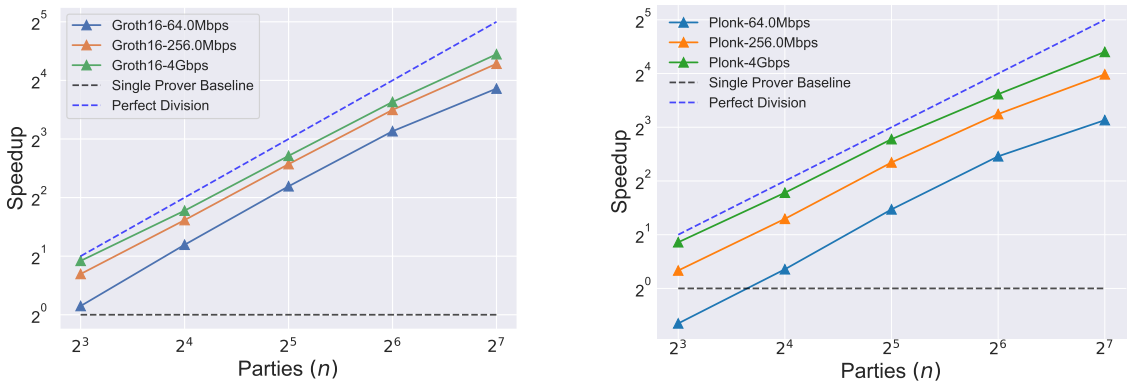

Figure 8: *Proving time versus number of parties, normalized by a single-prover time for* 2^{19} *constraints (gates) denoted by the dotted black line.*

References

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. Cryptology ePrint Archive, Report 2021/576, 2021. https://eprint.iacr.org/2021/576.

[2] alex ozdemir. collaborative-zksnark. https://github.com/alex-ozdemir/collaborative-zksnark, 2022.

[3] arkworks contributors. `arkworks` zksnark ecosystem. https://arkworks.rs, 2022.

[4] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. In *31st FOCS*, pages 16–25. IEEE Computer Society Press, October 1990.

[5] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989*, pages 201–209. ACM, 1989.

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

[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[8] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, 2015.

[9] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[10] Dario Bini and Victor Y. Pan. Polynomial division and its computational complexity. *J. Complex.*, 2(3):179–203, 1986.

[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *J. Cryptol.*, 30(4):989–1066, 2017.

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 111–120. ACM Press, June 2013.

[13] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 255–272. Springer, Heidelberg, August 2012.

[14] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 168–197. Springer, Heidelberg, November 2020.

[15] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 123–152, Virtual Event, August 2021. Springer, Heidelberg.

[16] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. Cryptology ePrint Archive, Report 2014/846, 2014. https://eprint.iacr.org/2014/846.

[17] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, August 2019.

[18] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776. IEEE Computer Society Press, May 2021.

[19] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K. Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 336–365. Springer, Heidelberg, December 2017.

[20] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 595–626. Springer, Heidelberg, December 2018.

[21] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 19–46. Springer, Heidelberg, November 2020.

[22] Jonathan Bootle, Alessandro Chiesa, and Siqi Liu. Zero-knowledge IOPs with linear-time prover and polylogarithmic-time verifier. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 275–304. Springer, Heidelberg, May / June 2022.

[23] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.

[24] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, volume 10958 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2018.

[25] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[26] Vitalik Buterin. Some ways to use zk-snarks for privacy.

[27] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei 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.

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

[29] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zksnark provers. In *USENIX Security Symposium*. USENIX Association, 2023.

[30] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.

[31] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 94–123, Virtual Event, August 2021. Springer, Heidelberg.

[32] William R Claycomb and Alex Nicoll. Insider threats to cloud computing: Directions for new research challenges. In *2012 IEEE 36th annual computer software and applications conference*, pages 387–394. IEEE, 2012.

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

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

[35] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261. Springer, Heidelberg, August 2008.

[36] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[37] Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. Zero knowledge static program analysis. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2951–2967. ACM Press, November 2021.

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

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

[40] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015.

[41] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[42] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

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

[44] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/1043, 2021. https://eprint.iacr.org/2021/1043.

[45] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 694–723. Springer, Heidelberg, October 2021.

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

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

[48] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. Cryptology ePrint Archive, Report 2022/831, 2022. https://eprint.iacr.org/2022/831.

[49] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Proc. Priv. Enhancing Technol.*, 2023(1):627–640, 2023.

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

[51] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017.

[52] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In *USENIX Security Symposium*, pages 4255–4272. USENIX Association, 2022.

[53] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In Mikkel Thorup, editor, *59th FOCS*, pages 124–135. IEEE Computer Society Press, October 2018.

[54] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018.

[55] Sanket Kanjalkar, Ye Zhang, Shreyas Gandlur, and Andrew Miller. Publicly auditable mpc-as-a-service with succinct verification and universal setup. In *IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 386–411. IEEE, 2021.

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

[57] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. Cryptology ePrint Archive, Report 2021/219, 2021. https://eprint.iacr.org/2021/219.

[58] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. A direct construction for asymptotically optimal zksnarks. *IACR Cryptol. ePrint Arch.*, page 1318, 2020.

[59] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part II*, volume 13043 of *LNCS*, pages 1–34. Springer, Heidelberg, November 2021.

[60] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vCNN: Verifiable convolutional neural network. Cryptology ePrint Archive, Report 2020/584, 2020. https://eprint.iacr.org/2020/584.

[61] Fu-Rong Lin, Wai-Ki Ching, and Michael K. Ng. Fast inversion of triangular toeplitz matrices. *Theor. Comput. Sci.*, 315(2-3):511–523, 2004.

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

[63] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 255–271. IEEE Computer Society, 2016.

[64] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4291–4308, Boston, MA, August 2022. USENIX Association.

[65] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252. IEEE Computer Society, 2013.

[66] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4. 2019.

[67] Nicholas Pippenger. On the evaluation of powers and monomials. *SIAM J. Comput.*, 9(2):230–250, 1980.

[68] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi. Cryptology ePrint Archive, Paper 2022/1286, 2022. https://eprint.iacr.org/2022/1286.

[69] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. `zk-creds`: Flexible anonymous credentials from zkSNARKs and existing identity infrastructure. Cryptology ePrint Archive, Report 2022/878, 2022. https://eprint.iacr.org/2022/878.

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

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

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

[73] Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In Martin Albrecht, editor, *Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings*, volume 11929 of *Lecture Notes in Computer Science*, pages 342–366. Springer, 2019.

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

[75] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013.

[76] Riad S. Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. Verifiable asics. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 759–778. IEEE Computer Society, 2016.

[77] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 926–943. IEEE Computer Society, 2018.

[78] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

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

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

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

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

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

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

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

[86] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vram: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 908–925. IEEE Computer Society, 2018.

[87] ZkRollups. An incomplete guide to rollups. https://vitalik.ca/general/2021/01/ 05/rollup.html, 2021.

A Groth16 zkSaaS

In [50], Groth introduced the smallest SNARKs based on pairing assumptions, where the proof size only consists of 3 group elements. These SNARKs rely on an honestly generated structured common reference string (CRS). In this section, we recall the construction of these SNARKs and then demonstrate how our techniques can be used to efficiently compute them in our zkSaaS framework.

Quadratic Arithmetic Programs (QAP). Groth16 SNARKs use a QAP representation of the relation. We start by recalling this representation and then proceed to give an overview of how the proof is computed.

Let C_R be an arithmetic circuit representation (over field $\mathbb{F}$) of the relation function that takes the statement and witness as input and outputs either 1 or 0 depending on whether or not the relation is satisfied. Such a relation can also be described by a set of equations over the statement $(a_1, \ldots, a_\ell)$ and witness $(a_{\ell+1}, \ldots, a_m)$. In particular, let $a_1, \ldots, a_m \in \mathbb{F}$ be variables used to denote the statement and witness and let $a_0 = 1$ be an extra variable. Then there exist variables $\{u_{i,q}, v_{i,q}, w_{i,q}\}_{i\in[0,m],q\in[Q]} \in \mathbb{F}$ such that each of the following equations are satisfied if and only if the output of C_R on input $a_1, \ldots, a_m$ is 1:

$$\sum_{i\in[0,m]} a_i \cdot u_{i,q} \cdot \sum_{i\in[0,m]} a_i \cdot v_{i,q} = \sum_{i\in[0,m]} a_i \cdot w_{i,q}$$

Here m is the size of C_R and Q are the number of multipication gates in the circuit. Gennaro, Gentry, Parno and Raykova [42] showed that for a large enough $\mathbb{F}$, this set of arithmetic constraints can be reformulated as a quadratic arithmetic program. Let $r_1, \ldots, r_Q \in \mathbb{F}$ be arbitrary distinct field elements. Let $t(x) = \prod_{q\in[Q]}(x - r_q)$ and for each $i \in [0, m]$, let $u_i(x), v_i(x), w_i(x)$ be degree $Q-1$ polynomials such that

$$\forall q \in [Q], \quad u_i(r_q) = u_{i,q} \quad v_i(r_q) = v_{i,q} \quad w_i(r_q) = w_{i,q}$$

The relation is satisfied if the following condition is satisfied for some degree $n-2$ quotient polynomial $h(X)$:

$$\sum_{i\in[0,m]} a_i \cdot u_i(X) \cdot \sum_{i\in[0,m]} a_i \cdot v_i(X) = \sum_{i\in[0,m]} a_i \cdot w_i(X) + h(X)t(X)$$

In order to compute the proof, the prover must first compute the polynomial $h(X)$. We assume that the client packed secret shares $a_1, \ldots, a_m$ with the servers. The client also packed secret shares Q "evaluations" of the polynomials $\sum_{i\in[0,m]} a_i u_i(X)$, $\sum_{i\in[0,m]} a_i v_i(X)$ and $\sum_{i\in[0,m]} a_i w_i(X)$. We also assume that the servers have a packed secret sharing of the evaluation representation of polynomial $t(X)$. Given this, the servers compute the following:

— Using our distributed protocols for polynomial multiplication and division they compute packed shares of the evaluation representation of $h(X)$. Finally they use the distributed protocol for inverse FFT to compute packed shares of the coefficients of $h(X)$. Let $h_0, \ldots, h_{Q-2}$ denote the coefficients of polynomial $h(X)$.

Structured Random String. As discussed earlier, Groth SNARKs are based on pairing assumption. Let the map be $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Let g_1 be the generator for $\mathbb{G}_1$ and g_2 be the generator for $\mathbb{G}_2$. The CRS consists of $3+m+2Q$ elements in $\mathbb{G}_1$ and $3+Q$ elements in $\mathbb{G}_2$. The actual values contained in this CRS are immaterial for the purpose of this example. Hence, we use random variables to denote the group elements in the CRS.

— $\mathbb{G}_1$ *Elements:* We use variables $(L, M, N, \{S_i\}_{i\in[0,m]}, \{H_i\}_{i\in[0,m]}, \{T_i\}_{i\in[0,\ell]}, \{W_i\}_{i\in[\ell+1,m]}, \{U_i\}_{i\in[0,Q-2]})$ to denote elements in $\mathbb{G}_1$. We assume that all the servers computing the proof get L, M, N in the clear and only receive packed shares of the remaining elements $\{S_i\}_{i\in[0,Q-1]}, \{H_i\}_{i\in[0,Q-1]}, \{T_i\}_{i\in[0,\ell]}, \{W_i\}_{i\in[\ell+1,m]}, \{U_i\}_{i\in[0,Q-2]}$.

— $\mathbb{G}_2$ *Elements:* Similarly, we use variables $(Z, O, K, \{V_i\}_{i\in[0,m]})$ to denote the elements in $\mathbb{G}_2$. As before, we assume that all the servers computing the proof get Z, O, K in the clear and only receive packed shares of the remaining elements $\{V_i\}_{i\in[0,Q-1]}$.

Proof. The proof consists of three group elements $A, C \in \mathbb{G}_1$ and $B \in \mathbb{G}_2$. The prover samples random field elements $r, s \in \mathbb{F}$. The parties can run f_{rand} to collectively sample and compute (regular) secret shares of r and s.

1. The prover computes the following:

$$A = L \cdot (N)^r \cdot \prod_{i\in[0,m]} (S_i)^{a_i}$$

Given packed shares of S_i and au_i terms, the servers can use π_{MSM} to compute $\prod_{i\in[0,Q-1]}(S_i)^{a_i}$. Since the output of MSM are regular shares, they can then be combined with L, N and regular shares of r to get regular shares of A.

2. Next, the prover computes:

$$B = Z \cdot (K)^s \cdot \prod_{i \in [0,m]} (V_i)^{a_i}$$

In our setting, this can be computed exactly as before. Given packed shares of V_i and av_i terms, the servers can use π_{MSM} to compute $\prod_{i \in [0,Q-1]} (V_i)^{a_i}$. The output of MSM can then be combined with Z, K and regular shares of s to get regular shares of B.

3. Finally, the prover computes:

$$C = \left(\prod_{i \in [\ell+1,m]} (W_i)^{a_i}\right)\left(\prod_{i \in [0,Q-2]} (U_i)^{h_i}\right) \cdot A^s \cdot (M)^r \cdot \left(\prod_{i \in [0,m]} (H_i)^{a_i}\right)^r$$

In our setting, the servers invoke 3 instances of π_{MSM} to compute regular shares of $\prod_{i \in [\ell+1,m]} (W_i)^{a_i}$, $\prod_{i \in [0,Q-2]} (U_i)^{h_i}$ and $\prod_{i \in [0,Q-1]} (H_i)^{a_i}$. The outputs of these are then combined with A, N, M and shares of r and s to obtain shares of C.

B Plonk zkSaaS

In Plonk [39], the authors build SNARKs based on pairing assumptions, using a universal setup, which as opposed to the CRS-model of Groth16 (which is circuit-specific), uses the same structured random string for statements about all circuits of a certain bounded size, and can be updated. The proof size of Plonk consists of 9-group elements and 6 field elements. In this section, we only give an overview of the key steps of the Plonk construction and describe how our techniques can be used to efficiently compute them in our zkSaaS framework. For the detailed description of Plonk, we refer the reader to [39, Section 8.3]. We only discuss the version of Plonk without zero-knowledge, for ease of exposition, however we can easily extend our technique to the zero-knowledge version.

Plonk Arithmetization. This differs from the QAP model used by Groth16. We begin by describing the constraint system used in Plonk [39] before proceeding to give an overview of how the proof is computed. For fixed integers m, Q, the Plonk constraint system $\mathcal{C} = (\mathcal{V}, \mathcal{Q})$ defined below captures fan-in two arithmetic circuits of unlimited fan-out with Q gates and m wires.

- $\mathcal{V}$ is of the form $\mathcal{V} = (\mathbf{a}, \mathbf{b}, \mathbf{c}) \in [m]^{3Q}$, where $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$ are the left, right and output sequence of $\mathcal{C}$.
- $\mathcal{Q} = (\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}) \in (\mathbb{F}^Q)^5$, where $\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}$ are "selector vectors".

$\mathbf{x} \in \mathbb{F}^m$ satisfies $\mathcal{C}$ if for each $i \in [Q]$,

$$(\mathbf{q_L})_i \cdot \mathbf{x_{a_i}} + (\mathbf{q_R})_i \cdot \mathbf{x_{b_i}} + (\mathbf{q_O})_i \cdot \mathbf{x_{c_i}} + (\mathbf{q_M})_i \cdot \mathbf{x_{a_i}} \mathbf{x_{b_i}} + (\mathbf{q_C})_i = 0$$

A relation $\mathcal{R}_\mathcal{C}$ is defined as the set of pairs $(x, w) \in \mathbb{F}^m$, such that (x, w) satisfies $\mathcal{C}$. In the pre-processing step, the polynomials that defined the circuit are computed: $q_M(X), q_L(X), q_R(X), q_O(X), q_C(X)$ are the selector polynomials, $S_{\mathsf{ID}_1}, S_{\mathsf{ID}_2}$ and S_{ID_3} are the polynomials that help in performing the permutation checks (to ensure the correct ordering of the $3Q$ wires across the gates) that are defined by a permutation map σ^* on $3Q$. Given these, to prove that $(x, w) \in \mathbb{F}^{3Q}$ satisfies the constraint $\mathcal{C}$, in addition to the constraint check defined above, the prover also proves that $w_i = w_{\sigma^*(i)}$ and $x_i = x_{\sigma^*(i)}$, $\forall i$.

We assume that the client packed secret shares the values $\mathbf{q_M}_i$, $\mathbf{q_L}_i$, $\mathbf{q_R}_i$, $\mathbf{q_O}_i$, $\mathbf{q_C}_i$ corresponding to all Q gates, and the $\sigma^*(i)$ corresponding to all $3Q$ wires. This corresponds to the packed shares of the evaluation representation of the respective polynomials.

Structured Random String. The Plonk pre-processing outputs an SRS consisting of $3Q + 3(\#(\text{addition gates}))$-$\mathbb{G}_1$ and 1-$\mathbb{G}_2$ elements along with all the polynomials mentioned above. This consists of powers of a random value τ, that will be used to get polynomial evaluations in the main protocol at τ. We assume that all these group elements are given as packed shares to the servers. In addition, as we mentioned above, the packed shares corresponding to the evaluations of all the polynomials are given to the server. Since, the Plonk-protocol uses evaluation representations of all polynomials throughout the protocol, we also assume that the Lagrange polynomial values at τ are packed secret shared and given to the servers. Looking ahead, this would be used in our MSM computations.

Proof. On input $(x, w) \in \mathbb{F}^{3Q}$, the Plonk prover algorithm interacts with the verifier in 5 rounds and sends 9-$\mathbb{G}_1$ elemets and 6-$\mathbb{F}$ elements. We mention the exact calls made to the sub-protocols (KZG commitment and proof generation, MSMs, partial products and FFTs) that are done in each of these five rounds. These can then be adapted to our techniques as well. For a detailed description of each round, refer to [39, Section 8.3].

1. In Round 1, the Plonk prover computes three polynomials $a(X), b(X)$ and $c(X)$, each of degree $Q-1$ that correspond to each of the Q left, right and output wire values given in the input and sends their KZG commitments to the verifier. We assume that the client packed secret shares the evaluations of a, b and c and sends to the server. Given these packed secret shares, computing each KZG commitment involves one call to the MSM functionality of size Q. Thus, use three invocations of our distributed MSM to compute the commitments in this round.

2. In Round 2, the prover sends the commitment of a single polynomial z correponding to the wire permutation check. The evaluations of z correspond to partial products on input of size Q. In our setting, the servers can invoke the distributed partial products to compute the packed shares of z and then invoke the distributed MSM to compute its commitment.

3. In Round 3, the prover computes a quotient polynomial $t(X)$ obtained by diving by a public polynomial $Z_H(X) = X^Q - 1$, and then computes 3 polynomial commitments on three parts of $t(X)$, each of degree $< Q$. Evaluating the polynomial $t(X)$, given the evaluations of all the polynomials from previous rounds, requires a constant number of field multiplications and additions followed by a single call to polynomial division. Splitting of this larger degree polynomial into the three degree $< Q$ polynomials helps in reducing the group elements in the SRS. In our setting, post the field multiplications and divisions on the corresponding packed shares, the servers invoke our distributed polynomial division which outputs the packed shares of $t(X)$ in the evaluation representation, and thus requires 2 calls to the distributed FFT.

4. In Rounds 4 and 5, the Plonk prover computes 6 polynomial openings on random values along with a KZG proof of these openings. Computing each opening involves a call to FFT, while computing the combined KZG proof involves doing one polynomial division followed by an 2 MSMs. In our setting, the servers can invoke the distributed FFT for each opening. For the opening proof, the servers can invoke distributed polynomial division once, followed by two calls to the distributed MSM.

C Sub-Protocols for Standard Functionalities

We now give a description of some standard MPC sub-protocols that are used in the design of our protocols in Section 5.

C.1 Secret Sharing Random Values

In this section, we describe a protocol π_{rand} (Figure 10) for computing shares of a batch of random and independently chosen values in $\mathbb{F}$. It makes use of the regular Shamir secret sharing scheme along with an $n \times (n-t)$ Vandermonde matrix $\mathbf{V}_{n,(n-t)}$. First, each party samples a random value and (regular) secret shares it amongst the other parties. The parties then compute $n-t$ linear combinations of these shares using the Vandermonde matrix to obtain shares of $n-t$ uniformly random values in $\mathbb{F}$.

This protocol realizes $n-t$ instantiations of the functionality f_{rand} (Figure 9) with abort. The total communication and computation complexity of this protocol is $O(n^2)$. However, since each protocol yields shares of $n-t$ random values, the amortized cost of sharing a single random value using this protocol is $O(n)$.

The functionality f_{rand}

The n-party functionality f_{rand}, running with parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{rand} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{rand} receives shares $\{u_i\}_{i \in \mathsf{Corr}}$ from Sim.
- f_{rand} chooses a random value $r \in \mathbb{F}$, and samples a random (degree t) sharing of r, such that the shares of the corrupted parties are identical to those received from the Sim, i.e., $\{u_i\}_{i \in \mathsf{Corr}}$.
- f_{rand} distributed the shares of r to the honest parties.

Figure 9: Functionality for Generating Shares of a Random Value in $\mathbb{F}$

Lemma 4. *The protocol π_{rand} described in Figure 10 securely computes $n-t$ instantiations of f_{rand} against a semi-honest adversary who controls upto t parties.*

The proof of this lemma follows from [36].

C.2 Double Sharing of Random Values

In this section, we describe a protocol $\pi_{\mathsf{double-rand}}$ (Figure 12) for computing doubles shares for a batch of random and independently chosen values in $\mathbb{F}$. This protocol is very similar to π_{rand}, except that here we generate two sharings of the same random value – one w.r.t. to a degree t sharing and another w.r.t. a degree $n-1$ sharing. The complexity of this protocol is same as that of π_{rand}.

Lemma 5. *The protocol $\pi_{\mathsf{double-rand}}$ described in Figure 12 securely computes $n-t$ instantiations of $f_{\mathsf{double-rand}}$ against a semi-honest adversary who controls upto t parties.*

The proof of this lemma follows from [36].

C.3 Packed Secret Sharing of Random Vectors

We now describe a protocol π_{prand} (in Figure 14) for computing a batch of packed secret shares of random vectors and do not reconstruct the vectors. This protocol is similar also to f_{rand}, except that now we compute packed shares of random vectors. The functionality f_{prand} realized by this protocol is presented in Figure 13. As before, the total communication and computation complexity of this protocol is

The protocol π_{rand}

Auxiliary Input: Vandermonde matrix $\mathbf{V}_{n,n-t}$ over $\mathbb{F}$.
Inputs: The parties do not have any inputs.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Each party P_i for $i \in [n]$ samples $u_i \leftarrow\$ \mathbb{F}$ uniformly at random and evaluates $\mathsf{share}(\mathbb{F}, u_i, t)$ to receive $[u_i]$.
- For each $i, j \in [n]$, party P_i sends $[u_i]_j$ to P_j.
- The parties compute

$$([r_1], \ldots, [r_{n-t}]) = \mathbf{V}^T_{n,n-t} \cdot ([u_1], \ldots, [u_n]).$$

Output: Each party P_i outputs shares $[r_1]_i, \ldots, [r_{n-t}]_i$.

Figure 10: Protocol for Generating Shares of a Random Value in $\mathbb{F}$

The functionality $f_{\mathsf{double-rand}}$

The n-party functionality $f_{\mathsf{double-rand}}$, running with parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- $f_{\mathsf{double-rand}}$ receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- $f_{\mathsf{double-rand}}$ receives shares $\{u_i, v_i\}_{i \in \mathsf{Corr}}$ from Sim.
- $f_{\mathsf{double-rand}}$ chooses a random value $r \in \mathbb{F}$, and samples random degree t and $n-1$ sharing of r, such that the shares of the corrupted parties are identical to those received from the Sim, i.e., $\{u_i, v_i\}_{i \in \mathsf{Corr}}$.
- $f_{\mathsf{double-rand}}$ distributed the two sharings of r to the honest parties.

Figure 11: Functionality for Generating Double Sharings of a Random Value in $\mathbb{F}$

$O(n^2)$. However, since each protocol yields packed sharings of $O(n-t)$ vectors, each of length $O(n)$, the amortized cost of sharing a random value using this protocol is $O(1)$.

Lemma 6. *The protocol π_{prand} described in Figure 14 securely computes $n-t$ instantiations of f_{prand} against a semi-honest adversary who controls upto t parties.*

The proof of this Lemma follows from [34].

C.4 Double Packed Secret Sharing of Random Vectors

We now describe a protocol $\pi_{\mathsf{double-prand}}$ (in Figure 16) for generating double packed sharings of a batch of random vectors. This is essentially a packed sharing based counterpart of $\pi_{\mathsf{double-prand}}$. The complexity of this protocol is similar to π_{prand}.

Lemma 7. *The protocol $\pi_{\mathsf{double-prand}}$ described in Figure 16 securely computes $n-t$ instantiations of $f_{\mathsf{double-prand}}$ against a semi-honest adversary who controls upto t parties.*

The protocol π_{rand}

Auxiliary Input: Vandermonde matrix $\mathbf{V}_{n,n-t}$ over $\mathbb{F}$.
Inputs: The parties do not have any inputs.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Each party P_i for $i \in [n]$ samples $u_i \leftarrow\!\$\, \mathbb{F}$ uniformly at random and evaluates $\mathsf{share}(\mathbb{F}, u_i, t)$ and $\mathsf{share}(\mathbb{F}, u_i, n-1)$ to receive $[u_i]$ and $\langle u_i \rangle$.
- For each $i, j \in [n]$, party P_i sends $[u_i]_j$ and $\langle u_i \rangle_j$ to P_j.
- The parties compute

$$([r_1], \ldots, [r_{n-t}]) = \mathbf{V}^T_{n,n-t} \cdot ([u_1], \ldots, [u_n])$$

$$(\langle r_1 \rangle, \ldots, \langle r_{n-t} \rangle) = \mathbf{V}^T_{n,n-t} \cdot (\langle u_1 \rangle, \ldots, \langle u_n \rangle).$$

Output: Each party P_i outputs shares $[r_1]_i, \ldots, [r_{n-t}]_i$ and $\langle r_1 \rangle_i, \ldots, \langle r_{n-t} \rangle_i$.

Figure 12: Protocol for Generating Double Sharings of a Random Value in $\mathbb{F}$

The functionality f_{prand}

The n-party functionality f_{prand}, running with parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{prand} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{prand} receives shares $\{u_i\}_{i \in \mathsf{Corr}}$ from Sim.
- f_{prand} chooses a random vector $\mathbf{r} \in \mathbb{F}^\ell$, and samples a random (degree D) sharing of $\mathbf{r}$, such that the shares of the corrupted parties are identical to those received from the Sim, i.e., $\{u_i\}_{i \in \mathsf{Corr}}$.
- f_{prand} distributed the shares of $\mathbf{r}$ to the honest parties.

Figure 13: Functionality for Generating Packed Sharing of a Random Vector over $\mathbb{F}$

The proof of this Lemma follows from [34].

C.5 Multiplying Packed Secret Shared Vectors

In this section, we describe a sub-protocol for multiplication. This subprotocol is identical to the one used in [34]. As proved in [34], this protocol is secure against a semi-honest adversary. This protocol $\pi_{\mathsf{pack-mult}}$ is described in Figure 20. We note that we describe this protocol assuming that it takes as input two pack secret shared inputs, where both are secret shared w.r.t. a degree D polynomial. However, it is easy to see that it still works as is, if any of those values are secret shared using a smaller degree polynomial. The protocol in Figure 20 incurs a total communication cost of $O(n)$ in order to multiply packed secret shares, containing $O(n)$ elements. As such, the amortized cost of multiplication is $O(1)$.

Lemma 8. *The protocol $\pi_{\mathsf{pack-mult}}$ described in Figure 18 securely computes $f_{\mathsf{pack-mult}}$ against a semi-honest adversary who controls upto t parties.*

The protocol π_{prand}

Auxiliary Input: Vandermonde matrix $\mathbf{V}_{n,n-t}$ over $\mathbb{F}$.
Inputs: The parties do not have any inputs.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Each party P_i for $i \in [n]$ samples $\mathbf{u}_i \leftarrow\$ \mathbb{F}^\ell$ uniformly at random and evaluates $\mathsf{pshare}(\mathbb{F}, \mathbf{u}_i, D)$ to receive $[\![\mathbf{u}_i]\!]$.
- For each $i, j \in [n]$, party P_i sends $[\![\mathbf{u}_i]\!]_j$ to P_j.
- The parties compute

$$([\![\mathbf{r}_1]\!]_i, \ldots, [\![\mathbf{r}_{n-t}]\!]_i) = \mathbf{V}^T_{n,n-t} \cdot ([\![\mathbf{u}_1]\!]_i, \ldots, [\![\mathbf{u}_n]\!]_i).$$

Output: Each party P_i outputs $[\![\mathbf{r}_1]\!]_i, \ldots, [\![\mathbf{r}_{n-t}]\!]_i$.

Figure 14: Protocol for Generating Packed Sharing of Random Vectors over $\mathbb{F}$

The functionality $f_{\mathsf{double-prand}}$

The n-party functionality $f_{\mathsf{double-prand}}$, running with parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- $f_{\mathsf{double-prand}}$ receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- $f_{\mathsf{double-prand}}$ receives shares $\{u_i, v_i\}_{i \in \mathsf{Corr}}$ from Sim.
- $f_{\mathsf{double-prand}}$ chooses a random vector $\mathbf{r} \in \mathbb{F}^\ell$, and samples random degree D and $n-1$ sharing of $\mathbf{r}$, such that the shares of the corrupted parties are identical to those received from the Sim, i.e., $\{u_i, v_i\}_{i \in \mathsf{Corr}}$.
- $f_{\mathsf{double-prand}}$ distributed the two sharings of $\mathbf{r}$ to the honest parties.

Figure 15: Functionality for Generating Double Packed Sharings of a Random Vector in $\mathbb{F}$

The proof of this Lemma follows from [34].

C.6 Multiplying Secret Shared Values

We now describe protocol π_{mult} for securely multiplying two secret shared values. This is a regular secret sharing variant of $\pi_{\mathsf{pack-mult}}$ and works exactly like that protocol, albeit on regular shares. This protocol incurs a total communication cost of $\mathcal{O}(n)$ for multiplying 2 secret shared values.

Lemma 9. *The protocol π_{mult} described in Figure 20 securely computes f_{mult} against a semi-honest adversary who controls upto t parties.*

The proof of this lemma follows from [36].

The protocol $\pi_{\mathsf{double-prand}}$

Auxiliary Input: Vandermonde matrix $\mathbf{V}_{n,n-t}$ over $\mathbb{F}$.
Inputs: The parties do not have any inputs.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Each party P_i for $i \in [n]$ samples $\mathbf{u}_i \leftarrow\$ \mathbb{F}^\ell$ uniformly at random and evaluates $\mathsf{pshare}(\mathbb{F}, \mathbf{u}_i, D)$ and $\mathsf{pshare}(\mathbb{F}, \mathbf{u}_i, n-1)$ to receive $[\![\mathbf{u}_i]\!]$ and $\langle\!\langle \mathbf{u}_i \rangle\!\rangle$.
- For each $i, j \in [n]$, party P_i sends $[\![\mathbf{u}_i]\!]_j$ and $\langle\!\langle \mathbf{u}_i \rangle\!\rangle_j$ to P_j.
- The parties compute

$$([\![\mathbf{r}_1]\!]_i, \ldots, [\![\mathbf{r}_{n-t}]\!]_i) = \mathbf{V}^T_{n,n-t} \cdot ([\![\mathbf{u}_1]\!]_i, \ldots, [\![\mathbf{u}_n]\!]_i).$$

$$(\langle\!\langle \mathbf{r}_1 \rangle\!\rangle_i, \ldots, \langle\!\langle \mathbf{r}_{n-t} \rangle\!\rangle_i) = \mathbf{V}^T_{n,n-t} \cdot (\langle\!\langle \mathbf{u}_1 \rangle\!\rangle_i, \ldots, \langle\!\langle \mathbf{u}_n \rangle\!\rangle_i).$$

Output: Each party P_i outputs $[\![\mathbf{r}_1]\!]_i, \ldots, [\![\mathbf{r}_{n-t}]\!]_i$ and $\langle\!\langle \mathbf{r}_1 \rangle\!\rangle_i, \ldots, \langle\!\langle \mathbf{r}_{n-t} \rangle\!\rangle_i$.

Figure 16: Protocol for Generating Double Packed Sharings of a Random Vector in $\mathbb{F}$

The Functionality $f_{\mathsf{pack-mult}}$

The functionality $f_{\mathsf{pack-mult}}$ running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- $f_{\mathsf{pack-mult}}$ receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- $f_{\mathsf{pack-mult}}$ receives shares $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$ from all parties.
- $f_{\mathsf{pack-mult}}$ receives from the adversary a set of shares $\{u_i\}_{i \in \mathsf{Corr}}$ from Sim.
- $f_{\mathsf{pack-mult}}$ runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{a}]\!], D)$ and $\mathsf{open}(\mathbb{F}, [\![\mathbf{b}]\!], D)$ to obtain $(a_1, \ldots, a_\ell)$ and $(b_1, \ldots, b_\ell)$.
- For each $j \in [\ell]$, $f_{\mathsf{pack-mult}}$ computes $c_j = a_j \cdot b_j$. Set $\mathbf{c} = (c_1, \ldots, c_\ell)$.
- $f_{\mathsf{pack-mult}}$ samples a random sharing of $\mathbf{c}$, such that the shares of the corrupted parties are identical to those received from Sim, i.e., $\{u_i\}_{i \in \mathsf{Corr}}$.
- $f_{\mathsf{pack-mult}}$ distributes the shares of $\mathbf{c}$ to the honest parties.

Figure 17: Functionality for Multiplying Two Packed Shared Vectors in $\mathbb{F}$

C.7 A Protocol for Converting Regular Shares to Packed Shares

In this section, we describe the non-interactive share conversion protocol presented in [6]. We slightly change notation to integrate it into the rest of our presentation. Additionally, because we work in $\mathbb{Z}_p$, all described operations are in $\mathbb{Z}_p$.

Let $f_1, \ldots, f_\ell$ be the degree $t + \ell$ polynomials that were used for secret sharing secrets $s_1 \ldots, s_\ell$ respectively such that each $f_i(z)$ (for $i \in [\ell]$) is of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial. Then each party P_j (for $j \in [n]$) holds shares $[s_1]_{\mathsf{P}_j}, \ldots, [s_1]_{\mathsf{P}_j} = f_1(\alpha_j), \ldots, f_\ell(\alpha_j)$.

The protocol $\pi_{\mathsf{pack-mult}}$

Inputs: The parties hold shares $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Parties invoke $f_{\mathsf{double-prand}}$ to receive $([\![\mathbf{r}]\!], \langle\!\langle \mathbf{r} \rangle\!\rangle)$.
- Parties compute $\langle\!\langle \mathbf{c} \rangle\!\rangle = [\![\mathbf{a}]\!] \cdot [\![\mathbf{b}]\!]$.
- Parties compute $\langle\!\langle \mathbf{d} \rangle\!\rangle = \langle\!\langle \mathbf{c} \rangle\!\rangle + \langle\!\langle \mathbf{r} \rangle\!\rangle$ and send $\langle\!\langle \mathbf{d} \rangle\!\rangle$ to P_1.
- P_1 runs $\mathsf{open}(\mathbb{F}, \langle\!\langle \mathbf{d} \rangle\!\rangle, n-1)$ to receive $\mathbf{d}$.
- P_1 computes $[\![\mathbf{d}]\!] \leftarrow \mathsf{pshare}(\mathbb{F}, \mathbf{d}, D)$ and sends $[\![\mathbf{d}]\!]_i$ to party P_i (for each $i \in [n]$).
- Parties computes $[\![\mathbf{c}]\!] = [\![\mathbf{d}]\!] - [\![\mathbf{r}]\!]$

Output: Each party P_i outputs $[\![\mathbf{c}]\!]_i$.

Figure 18: Protocol for Multiplying Two Packed Shared Vectors in $\mathbb{F}$

The Functionality f_{mult}

The functionality f_{mult} running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{mult} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{mult} receives shares $[a]$ and $[b]$ from all parties.
- f_{mult} receives from the adversary a set of shares $\{u_i\}_{i \in \mathsf{Corr}}$ from Sim.
- f_{mult} runs $\mathsf{open}(\mathbb{F}, [a], t)$ and $\mathsf{open}(\mathbb{F}, [b], t)$ to obtain a and b.
- f_{mult} computes $c = a \cdot b$.
- f_{mult} samples a random sharing of c, such that the shares of the corrupted parties are identical to those received from Sim, i.e., $\{u_i\}_{i \in \mathsf{Corr}}$.
- f_{mult} distributes the shares of c to the honest parties.

Figure 19: Functionality for Multiplying Two Secret Shared Values in $\mathbb{F}$

Given these shares, each party P_j (locally) computes a packed secret share $[\![s_1, \ldots, s_\ell]\!]$ as follows:

$$\mathsf{ssToPss}(\mathbb{Z}_p, \{f_i(\alpha_j)\}_{i \in [\ell]}) = \sum_{i=1}^{\ell} f_i(\alpha_j) L_i(\alpha_j) = f(\alpha_j)$$

where $L_i(\alpha_j) = \prod_{j=1, j \neq i}^{\ell} \frac{(\alpha_i - e_j)}{(e_i - e_j)}$ is the Lagrange interpolation constant and f corresponds to a new degree $D = t + 2\ell - 1$ polynomial for the packed secret sharing $[\![s_1, \ldots, s_\ell]\!]$.

Lemma 10. *For each $i \in [\ell]$, let $s_a \in \mathbb{Z}_p$ be secret shared using a degree $t + \ell$ polynomial f_i of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial and $e_1, \ldots, e_\ell$ are some pre-determined elements in*

The protocol π_{mult}

Inputs: The parties hold shares $[a]$ and $[b]$.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Parties invoke $f_{\mathsf{double-rand}}$ to receive $([r], \langle r \rangle)$.
- Parties compute $\langle c \rangle = [a] \cdot [b]$.
- Parties compute $\langle d \rangle = \langle c \rangle + \langle r \rangle$ and send $\langle d \rangle$ to P_1.
- P_1 runs $\mathsf{open}(\mathbb{F}, \langle d \rangle, n-1)$ to receive d.
- P_1 computes $[d] \leftarrow \mathsf{share}(\mathbb{F}, d, t)$ and sends $[d]_i$ to party P_i (for each $i \in [n]$).
- Parties computes $[c] = [d] - [r]$

Output: Each party P_i outputs $[c]_i$.

Figure 20: Protocol for Multiplying Two Secret Shared Values in $\mathbb{F}$

$\mathbb{Z}_p$. Then for each $j \in [n]$, $\mathsf{ssToPss}(\mathbb{Z}_p, \{f_i(\alpha_j)\}_{i \in [\ell]})$ outputs the j^{th} share corresponding to a valid packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$, w.r.t. a degree-$D = t + 2\ell - 1$ polynomial.

The proof of Lemma 10 can be found in [6].

C.8 Converting Packed Shares to Regular Shares

In this section we describe a sub-protocol for converting packed shares to regular shares. The protocol begins with the parties generating masks in regular and packed secret shared form. This can be done by slightly modifying the randomness generation protocols, where each party sends a packed sharing and regular shares of a vector of random values. The shares received from all parties are then multiplied with the Vandermonde matrix to obtain the required shares of a batch of $n - t$ sharings. Next, the parties apply these pack secret shared masks to the pack secret sharing that we want to convert and publicly reconstruct the values. Finally, the parties can locally remove the masks using the regular shares of the masks. The total computation complexity for generating the batch of random sharings here is $O(n^3)$. But this randomness generation allows us to generate $O(n)$ sets of randomness that can be used to convert $O(n)$ packed sharings to regular sharings. Therefore the amortized cost of this step is $O(n^2)$. The total computational complexity of the remaining protocol is $O(n^2)$, to convert a vector of $O(n)$ shares. Thus, the amortized cost of converting packed shares to regular shares is $O(n)$.

Lemma 11. *The protocol π_{psstoss} described in Figure 22 securely computes f_{psstoss} against a semi-honest adversary who controls upto t parties.*

Proof of this lemma following similarly to that of Lemma 5.

C.9 Permutation

In this section, we present a protocol for permuting a block of pack secret shared vectors. This protocol is similar to the one presented in [34]. For $x_1, \ldots, x_m \in \mathbb{F}$ and for each $i \in [k]$ (where $k = m/\ell$), let $\mathbf{x}_i = (x_{(i-1)\ell+1}, \ldots, x_{i\ell})$. Let p_{perm} be a permutation function that permutes $x_1, \ldots, x_{k\ell}$ to give $z_1, \ldots, z_{k\ell}$. Set

The Functionality f_{psstoss}

The n-party functionality f_{psstoss}, running with parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{psstoss} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{psstoss} receives shares $[\![\mathbf{x}]\!]$ from all parties.
- For each $j \in [\ell]$, f_{psstoss} receives from the adversary a set of shares $\{u_{j,i}\}_{i \in \mathsf{Corr}}$ from Sim.
- f_{psstoss} runs $\mathsf{open}(\mathbb{F}, [\mathbf{x}], D)$ to obtain $\mathbf{x} = (x_1, \ldots, x_\ell)$
- For each $j \in [\ell]$, f_{psstoss} computes a random sharing of x_j such that the shares of the corrupted parties are identical to those received from Sim, i.e., $\{u_{j,i}\}_{i \in \mathsf{Corr}}$.
- For each $j \in [\ell]$, f_{psstoss} distributes the shares of x_j to the honest parties.

Figure 21: Functionality for Transforming Packed Secret Sharing to Regular Secret Sharing

The protocol π_{psstoss}

Inputs: The parties hold the packed sharing $[\![\mathbf{x}]\!]$ to be converted.
Protocol: The parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ proceed as follows:

- Each party P_i for $i \in [n]$ samples $\mathbf{u}_i = (\mathbf{u}_i[1], \ldots, \mathbf{u}_i[\ell]) \leftarrow\!\$\, \mathbb{F}^\ell$ uniformly at random and evaluates $\mathsf{pshare}(\mathbb{F}, \mathbf{u}_i, D)$ and $\mathsf{share}(\mathbb{F}, \mathbf{u}_i[j], t)$ for every $j \in [\ell]$ to receive $[\![\mathbf{u}_i]\!]$ and $[\mathbf{u}_i[1]], \ldots, [\mathbf{u}_i[\ell]]$.
- For each $i, j \in [n]$, party P_i sends $[\![\mathbf{u}_i]\!]_j$ and $[\mathbf{u}_i[1]]j, \ldots, [\mathbf{u}_i[\ell]]j$ to P_j.
- The parties compute

$$([\![\mathbf{r}_1]\!]_i, \ldots, [\![\mathbf{r}_{n-t}]\!]_i) = \mathbf{V}^T_{n,n-t} \cdot ([\![\mathbf{u}_1]\!]_i, \ldots, [\![\mathbf{u}_n]\!]_i).$$

$$\forall j \in [\ell], ([\mathbf{r}_1[j]]_i, \ldots, [\mathbf{r}_{n-t}[j]]_i) = \mathbf{V}^T_{n,n-t} \cdot ([\mathbf{u}_1[j]]_i, \ldots, [\mathbf{u}_{n-t}[j]]_i).$$

- Parties compute $[\![\mathbf{y}]\!] = [\![\mathbf{x}]\!] - [\![\mathbf{r}_1]\!]$ and send $[\![\mathbf{y}]\!]$ to P_1.
- P_1 runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{y}]\!], D)$ to obtain $\mathbf{y} = (y_1, \ldots, y_\ell)$. It sends $\mathbf{y}$ to all other parties.
- For each $j \in [\ell]$, parties compute $[x_j] = y_j + [\mathbf{r}_1[j]]$.

Output: Each party P_i outputs $([x_1]_i, \ldots, [x_\ell]_i)$.

Figure 22: Protocol for Transforming Packed Secret Sharing to Regular Secret Sharing

$\mathbf{z}_i = (z_{(i-1)\ell+1}, \ldots, z_{i\ell})$, for each $i \in [k]$. We want the following permutation functionality $f_{\mathsf{permute}}(p_{\mathsf{perm}})$, that takes the packed shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]$ as input and outputs the packed shares of the permuted values, $[\![\mathbf{z}_1]\!], \ldots, [\![\mathbf{z}_k]\!]$.
Protocol Overview: From the pre-processing step, for each $j \in [k]$ the parties get packed shares $\langle\!\langle \mathsf{mask}_j \rangle\!\rangle$, $[\![\mathsf{unmask}_j]\!]$, where $\mathsf{mask}_j = \{r_{j,1}, \ldots, r_{j,\ell}\}$ (hidden from all parties) are picked at random and $\mathsf{unmask}_j =$

The functionality $f_{\mathsf{permute}}(p_{\mathsf{perm}})$

The functionality f_{permute}, running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- f_{permute} receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- f_{permute} receives shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]$ from all parties.
- For each $j \in [k]$, f_{permute} receives from the adversary a set of shares $\{u_{j,i}\}_{i \in \mathsf{Corr}}$.
- For each $j \in [k]$, f_{permute} runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{x}_j]\!], t)$ to obtain $(x_{(j-1)\ell+1}, \ldots, x_{j\ell})$.
- f_{permute} computes $z_1, \ldots, z_{k\ell} = p_{\mathsf{perm}}(x_1, \ldots, x_{k\ell})$. For each $j \in [k]$, set $\mathbf{z}_j = (z_{(j-1)\ell+1,\ldots,z_{j\ell}})$.
- For each $j \in [k]$, f_{permute} samples a random sharing of $\mathbf{z}_j$, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_{j,i}\}_{i \in \mathsf{Corr}}$.
- f_{permute} distributes the shares of $\mathbf{z}_1, \ldots, \mathbf{z}_k$ to the honest parties.

Figure 23: Ideal Functionality for Permuting a List of Pack Shared Field Elements

$\{s_{j,1}, \ldots, s_{j,\ell}\}$ is computed as: $\{s_{j,1}, \ldots, s_{j,\ell}\}_{j \in [k]} = p_{\mathsf{perm}}(\{r_{j,1}, \ldots, r_{j,\ell}\}_{j \in [k]})$. In Step 1 of the protocol, the parties locally mask their shares using the $\langle\!\langle \mathsf{mask}_j \rangle\!\rangle$'s, and get the $(n-1)$-sharings of $\mathbf{y}_j = \mathbf{x}_j + \mathsf{mask}_j$ for each $j \in [k]$. In Steps 2 and 3, P_1 opens all the packed shares to get $(y_1, \ldots, y_{k\ell})$ and generates the permutation $z_1, \ldots, z_{k\ell} = p_{\mathsf{perm}}(y_1, \ldots, y_{k\ell})$. P_1 then sets $\mathbf{z}_j = (z_{(j-1)\ell+1,\ldots,z_{j\ell}})$ for each $j \in [k]$ and generates the packed shares $[\![\mathbf{z}_1]\!], \ldots, [\![\mathbf{z}_k]\!]$. Once all parties receive this, in the final Step 5, the parties unmask these packed shares using the $[\![\mathsf{unmask}_j]\!]$'s to get the packed shares $[\![\mathbf{out}_1]\!], \ldots, [\![\mathbf{out}_k]\!]$, which are the desired packed shares of $p_{\mathsf{perm}}(x_1, \ldots, x_{k\ell})$. The detailed protocol is as given in Figure 24.

The Protocol $\pi_{\mathsf{permute}}(p_{\mathsf{perm}})$

Inputs: Every party P_i holds packed shares of the values to be multiplied $([\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!])$.
Pre-Processing: For each $j \in [k]$, every party P_i holds packed shares $\langle\!\langle \mathsf{mask}_j \rangle\!\rangle, [\![\mathsf{unmask}_j]\!]$. Here $\mathsf{mask}_j = \{r_{j,1}, \ldots, r_{j,\ell}\}$ are random vectors unknown to any party and $\mathsf{unmask}_j = \{s_{j,1}, \ldots, s_{j,\ell}\}$, such that $\{s_{j,1}, \ldots, s_{j,\ell}\}_{j \in [k]} = p_{\mathsf{perm}}(\{r_{j,1}, \ldots, r_{j,\ell}\}_{j \in [k]})$.
Protocol: The parties $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ proceed as follows:

1. For each $j \in [k]$, compute $[\![\mathbf{y}_j]\!] = [\![\mathbf{x}_j]\!] + \langle\!\langle \mathsf{mask}_j \rangle\!\rangle$ and send $[\![\mathbf{y}_j]\!]$ to P_1.
2. For each $j \in [k]$, P_1 runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{y}_j]\!], n-1)$ to obtain $(y_{(j-1)\ell+1}, \ldots, y_{j\ell})$.
3. P_1 computes $z_1, \ldots, z_{k\ell} = p_{\mathsf{perm}}(y_1, \ldots, y_{k\ell})$ and for each $j \in [k]$, set $\mathbf{z}_j = (z_{(j-1)\ell+1,\ldots,z_{j\ell}})$.
4. For each $j \in [k]$, P_1 runs $[\![\mathbf{z}_j]\!] \leftarrow \mathsf{pshare}(\mathbb{F}, \mathbf{z}_j, D)$ and sends $[\![\mathbf{z}_j]\!]_i$ to party P_i (for each $i \in [1, n]$).
5. For each $j \in [k]$, parties compute $[\![\mathbf{out}_j]\!] = [\![\mathbf{z}_j]\!] + [\![\mathsf{unmask}_j]\!]$.

Output: Each party P_i outputs $[\![\mathbf{out}_1]\!]_i, \ldots, [\![\mathbf{out}_k]\!]_i$.

Figure 24: Permuting a List of Pack Shared Field Elements

Lemma 12. *Protocol π_{permute} securely computes functionality f_{permute} against a semi-honest adversary who corrupts at most t parties.*

Proof. The correctness of the protocol follows directly from the discussion in the protocol overview and the fact that p_{perm} is a linear function.

Security Proof. Let $\mathsf{Corr} \subset [n]$, with $|\mathsf{Corr}| = t$, be the set of corrupted parties. We show how to simulate the view of Corr in the ideal world, given the input shares $(\llbracket \mathbf{x}_1 \rrbracket_{\mathsf{Corr}}, \ldots, \llbracket \mathbf{x}_k \rrbracket_{\mathsf{Corr}})$ of the corrupted parties. The simulator $\mathcal{S}$ does the following:

— For the pre-processing step, the simulator samples the sets of random values $\mathsf{mask}_1, \cdots, \mathsf{mask}_k$, and computes $\mathsf{unmask}_j = \{s_{j,1}, \ldots, s_{j,\ell}\}$ for each $j \in [k]$, where $\{s_{j,1}, \ldots, s_{j,\ell}\}_{j \in [k]} = p_{\mathsf{perm}}(\bigcup_{j \in [k]} \mathsf{mask}_j)$. Then, $\mathcal{S}$ generates the shares $\langle\!\langle \mathsf{mask}_1 \rangle\!\rangle_{\mathsf{Corr}}, \ldots, \langle\!\langle \mathsf{mask}_k \rangle\!\rangle_{\mathsf{Corr}}, \llbracket \mathsf{unmask}_1 \rrbracket_{\mathsf{Corr}}, \ldots, \llbracket \mathsf{unmask}_k \rrbracket_{\mathsf{Corr}}$, corresponding to the corrupted parties.

— For Step 1, $\mathcal{S}$ can locally compute $\llbracket \mathbf{y}_j \rrbracket_{\mathsf{Corr}} = \llbracket \mathbf{x}_j \rrbracket_{\mathsf{Corr}} + \langle\!\langle \mathsf{mask}_j \rangle\!\rangle_{\mathsf{Corr}}$.

— For Steps 2, 3 and 4, $\mathcal{S}$ first generates $(y_1, \ldots, y_{k\ell})$ at random, computes $z_1, \ldots, z_{k\ell} = p_{\mathsf{perm}}(y_1, \ldots, y_{k\ell})$ and for each $j \in [k]$, sets $\mathbf{z}_j = (z_{(j-1)\ell+1, \ldots, z_{j\ell}})$. Note that, these intermediate values can be used when $P_1 \in \mathsf{Corr}$. Finally $\mathcal{S}$ generates $\llbracket \mathbf{z}_1 \rrbracket_{\mathsf{Corr}}, \ldots, \llbracket \mathbf{z}_k \rrbracket_{\mathsf{Corr}}$. This can be done as the y_j's look random to Corr, by the security of packed secret sharing.

— Finally, $\mathcal{S}$ computes and sets the output shares of Corr as $\llbracket \mathbf{out}_j \rrbracket_{\mathsf{Corr}} := \llbracket \mathbf{z}_j \rrbracket_{\mathsf{Corr}} + \llbracket \mathsf{unmask}_j \rrbracket_{\mathsf{Corr}}$ for each $j \in [k]$.

Note here that for each step, by security of the corresponding packed secret sharing scheme (wherever mentioned) and as the random mask_j's are hidden from all parties, $\mathcal{S}$ generates a distribution that is identical to the views of Corr in the real world. Hence, the ideal world distribution $\mathsf{IDEAL}_{f_{\mathsf{permute}}(p_{\mathsf{perm}}), \mathsf{Corr}, \mathcal{S}}(1^\lambda, \llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_k \rrbracket)$, corresponding the functionality f_{permute} is identical to the real world distribution $\mathsf{REAL}_{\pi_{\mathsf{permute}}, \mathsf{Corr}, \mathcal{A}}(1^\lambda, \llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_k \rrbracket)$. □

D Alternate Protocol for Partial Products with Equal Division of Work

In this section, we present an alternative protocol for distributed computation of partial products, where the work gets equally divided amongst all parties.

For this alternate protocol, the parties are assumed to have packed secret sharing of the following vectors: Let $k = m/\ell$. For each $i \in [k]$, $\mathbf{x}_i = (x_{(i-1)\ell+1}, \ldots, x_{i\ell})$. Since the order in which the vectors are packed is different here, correspondingly, we describe the alternate ideal functionality for this order in Figure 25.

Protocol Overview. In the pre-processing step, for each $j \in [m/\ell]$, the parties get packed shares $\llbracket \mathsf{mask}_j \rrbracket$ and $\llbracket \mathsf{unmask}_j \rrbracket$, where $\mathsf{mask}_j := \mathbf{s}_j \odot \mathbf{s}_j^{\mathsf{inv}}$, for a random vector $\mathbf{s}_j = ((\mathbf{s}_j)_1, \ldots, (\mathbf{s}_j)_\ell)$, unknown to any party and $\mathbf{s}_j^{\mathsf{inv}} = ((\mathbf{s}_j^{-1})_2, \ldots, (\mathbf{s}_j^{-1})_\ell, (\mathbf{s}_{j+1}^{-1})_1)$, and $\mathsf{unmask}_j = ((\mathbf{s}_1^{-1})_1, \ldots, (\mathbf{s}_1^{-1})_1) \odot \mathbf{s}_j$.

In the main protocol, the first step involves masking the input to compute the packed shares of the vectors $\mathbf{y}_j = \mathbf{x}_j \odot \mathsf{mask}_j$, following which party P_1 opens and sends k/n vectors $(\mathbf{y}_j)_{j=(i-1)k/n+1, \ldots, ik/n}$ to party P_i, for each $i = 2, \ldots, n$. In the second step, each party P_i computes the partial product function F_{part} on input of size $k/n \cdot \ell = m/n$, to get $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$. Here, $(\mathbf{z}_i)_j$ represents the j-th value in vector $\mathbf{z}_i$. Then, for each $i \in [n]$, party P_i sends the last value, $(\mathbf{z}_{ik/n})_\ell = (\mathbf{s}_{(i-1)k/n+1})_1 \cdot (\prod_{j=(i-1)k/n+1}^{ik/n} \prod_{i=1}^{\ell} (\mathbf{x}_j)_i) \cdot (\mathbf{s}_{ik/n}^{\mathsf{inv}})_\ell$ to all the parties.

In Step 3, for each $i = 2, \ldots, n$, P_i computes $\prod_{j=1}^{i-1} (\mathbf{z}_{jk/n})_\ell$ and multiplies it with the vector that it computed, i.e., $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$ to get the vectors

The functionality $f_{\mathsf{part-product-2.0}}$

The functionality $f_{\mathsf{part-product-2.0}}$, running with a set of parties $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ and the ideal adversary Sim proceeds as follows:

- $f_{\mathsf{part-product-2.0}}$ receives the set of corrupted parties, denoted $\mathsf{Corr} \subset [n]$.
- $f_{\mathsf{part-product-2.0}}$ receives shares $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_k]\!]$ from all parties.
- $f_{\mathsf{part-product-2.0}}$ receives from the adversary a set of shares $\{u_i\}_{i \in \mathsf{Corr}}$.
- For each $j \in [k]$, $f_{\mathsf{part-product-2.0}}$ runs $\mathsf{open}(\mathbb{F}, [\![\mathbf{x}_j]\!], D)$ to obtain $(x_{(j-1)\ell+1}, \ldots, x_{j\ell})$.
- $f_{\mathsf{part-product-2.0}}$ computes $(z_1, \ldots, z_m) := (\prod_{i \in [j]} x_i)_{j \in [m]}$. For each $j \in [m/\ell]$, let $\mathbf{z}_j = (z_i)_{i \in [(j-1)\ell+1, j\ell]}$.
- $f_{\mathsf{part-product-2.0}}$ samples a random sharing of $\mathbf{z}_j$, such that the shares of the corrupted parties are identical to those received from the adversary, i.e., $\{u_{i,j}\}_{i \in \mathsf{Corr}}$.
- $f_{\mathsf{part-product-2.0}}$ distributes the shares of $\mathbf{z}_1, \ldots, \mathbf{z}_{m/\ell}$ to the honest parties.

Figure 25: Ideal Functionality for Partial Products 2.0

$(\mathbf{h}_{(i-1)k/n+1}, \ldots, \mathbf{h}_{ik/n})$. Finally, for each $j \in [k]$, the parties compute the packed shares of $\mathsf{unmask}_j \odot \mathbf{h}_j$, where $\mathbf{h}_j = \mathbf{z}_j$ for each $j \in [1, k/n]$. Note that from the way that the mask_j's and unmask_j's are setup, this final output exactly contains the partial products desired. The detailed protocol is given in Figure 26.

Complexity. It is easy to see that this protocol runs in constant rounds. Moreover, each party is required to communicate $\mathcal{O}(m/\ell + n)$ field elements. Each server performs $\mathcal{O}(m/\ell)$ field operations and requires $\mathcal{O}(m/\ell)$ space complexity. We note that unlike all our other sub-protocols, where the weak servers only communicate with the large server, this protocol requires every server to talk to every other server.

Lemma 13. *Protocol $\pi_{\mathsf{part-prod-2.0}}$ securely computes functionality $f_{\mathsf{part-product-2.0}}$ (c.f. figure 25) in the $f_{\mathsf{pack-mult}}$-hybrid model against a semi-honest adversary who corrupts at most t parties.*

Proof. Correctness. The correctness of the protocol follows directly from the correctness of the underlying functionality, $f_{\mathsf{pack-mult}}$ and the discussion in the protocol overview.

Security Proof. The security of this protocol only holds when $x_j \neq 0$, for all $j \in [m]$, except with a negligible probability. Let $\mathsf{Corr} \subset [n]$, with $|\mathsf{Corr}| = t$, be the set of corrupted parties. We show how to simulate the view of Corr in the ideal world, given the input shares $([\![\mathbf{x}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathbf{x}_k]\!]_{\mathsf{Corr}})$ of the corrupted parties. The simulator $\mathcal{S}$ does the following:

- For the pre-processing,$\mathcal{S}$ samples the random vectors mask_j and unmask_j as described in the main protocol. Then, $\mathcal{S}$ generates the shares $([\![\mathsf{mask}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathsf{mask}_k]\!]_{\mathsf{Corr}})$, $([\![\mathsf{unmask}_1]\!]_{\mathsf{Corr}}, \ldots, [\![\mathsf{unmask}_k]\!]_{\mathsf{Corr}})$, corresponding to the corrupted parties.
- For Step 1, the parties run the ideal functionality $f_{\mathsf{pack-mult}}$. $\mathcal{S}$ can set the shares of the output of this as shares of some random vectors (by the security of the packed secret sharing), and picks $\mathbf{y}_j$ at random for each $j \in [k]$ (since the mask_j's look random to Corr, the $\mathbf{y}_j$'s also look random to Corr. This is true only because $\mathbf{x}_j$'s are all non-zero!).
- For Step 2, $\mathcal{S}$ can generate $F_{\mathsf{part}}((\mathbf{y}_{(i-1)k/n+1})_1, \ldots, (\mathbf{y}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{y}_{ik/n})_1, \ldots, (\mathbf{y}_{ik/n})_\ell)$ to get $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$, for each $i \in \mathsf{Corr}$.

The protocol $\pi_{\mathsf{part-prod-2.0}}$

Inputs: Every party P_i holds packed shares of the values to be multiplied $(\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_k \rrbracket)$.
Pre-processing: For each $j \in [m/\ell]$, every party P_i holds packed shares $\llbracket \mathsf{mask}_j \rrbracket$, and $\llbracket \mathsf{unmask}_j \rrbracket$. Here, for each $j \in [m/\ell]$, $\mathsf{mask}_j := \mathbf{s}_j \odot \mathbf{s}_j^{\mathsf{inv}}$, where $\mathbf{s}_j = ((\mathbf{s}_j)_1, \ldots, (\mathbf{s}_j)_\ell)$ is a random vector, unknown to any party and $\mathbf{s}_j^{\mathsf{inv}} = ((\mathbf{s}_j^{-1})_2, \ldots, (\mathbf{s}_j^{-1})_\ell, (\mathbf{s}_{j+1}^{-1})_1)$, and $\mathsf{unmask}_j = ((\mathbf{s}_1^{-1})_1, \ldots, (\mathbf{s}_1^{-1})_1) \odot \mathbf{s}_j$.
Protocol: The parties $(\mathsf{P}_1, \ldots, \mathsf{P}_n)$ proceed as follows:

1. Compute $\llbracket \mathbf{y}_j \rrbracket$ where $\mathbf{y}_j = \mathbf{x}_j \odot \mathsf{mask}_j$ by computing the following for all $j \in [k]$
   - Run $f_{\mathsf{pack-mult}}(\mathbb{F}, \llbracket \mathbf{x}_j \rrbracket, \llbracket \mathsf{mask}_j \rrbracket)$ to receive $\llbracket \mathbf{y}_j \rrbracket$ and send $\llbracket \mathbf{y}_j \rrbracket$ to P_1.
   - P_1 runs $\mathsf{open}(\mathbb{F}, \llbracket \mathbf{y}_j \rrbracket, D)$ and sends $(\mathbf{y}_j)_{j=(i-1)k/n+1,\ldots,ik/n}$ to party P_i, for each $i = 2, \ldots, n$.
2. For each $i \in [n]$, party P_i computes $F_{\mathsf{part}}((\mathbf{y}_{(i-1)k/n+1})_1, \ldots, (\mathbf{y}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{y}_{ik/n})_1, \ldots, (\mathbf{y}_{ik/n})_\ell)$ to get $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$. Here, $(\mathbf{z}_i)_j$ represents the j-th value in vector $\mathbf{z}_i$. Then, for each $i \in [n]$, party P_i sends the last value, $(\mathbf{z}_{ik/n})_\ell$ to all the parties.

   Note here that each party is computing the partial product of $k/n \cdot \ell = m/n$ inputs, and sending one field element to every other party.
3. For each $i = 2, \ldots, n$, party P_i first computes $\prod_{j=1}^{i-1} (\mathbf{z}_{jk/n})_\ell$ and then multiplies it with the vector that it computed, i.e., $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$ to get $(\mathbf{h}_{(i-1)k/n+1}, \ldots, \mathbf{h}_{ik/n})$. For $j \in [1, k/n]$, set $\mathbf{h}_j = \mathbf{z}_j$.
4. Finally, for each $j \in [k]$, run $f_{\mathsf{pack-mult}}(\mathbb{F}, \llbracket \mathsf{unmask}_j \rrbracket, \mathbf{h}_j)$ to get $\llbracket \mathbf{out}_j \rrbracket$.

Output: Each party P_i outputs $\{\llbracket \mathbf{out}_j \rrbracket_i\}_{j \in [k]}$.

Figure 26: Distributed Protocol for Computing Partial Products 2.0 with Equal Division of Work

- For Step 3, since for each $i \in \mathsf{Corr}$, $(\mathbf{z}_{(i-1)k/n})_\ell$ received from the party P_{i-1} looks random to P_i, $\mathcal{S}$ can pick it at random and multiply it with $((\mathbf{z}_{(i-1)k/n+1})_1, \ldots, (\mathbf{z}_{(i-1)k/n+1})_\ell, \ldots, (\mathbf{z}_{ik/n})_1, \ldots, (\mathbf{z}_{ik/n})_\ell)$ to get $(\mathbf{h}_{(i-1)k/n+1}, \ldots, \mathbf{h}_{ik/n})$. If $i = 1$, $\mathcal{S}$ just sets $\mathbf{h}_j = \mathbf{z}_j$ for each $j \in [1, k/n]$.
- Finally, in Step 4 the parties run $f_{\mathsf{pack-mult}}$. $\mathcal{S}$ sets the output shares $\mathbf{out}_j$'s corresponding to Corr as shares of random values (by the security of packed secret sharing).

Note here that for each step, by security of the packed secret sharing scheme (wherever mentioned) and since the y_j's look random to Corr (under our assumption that the x_j's are all non-zero w.h.p.), $\mathcal{S}$ generates a distribution that is identical to the views of Corr in the real world, assuming that the x_j's are all non-zero. Hence, the ideal world distribution $\mathsf{IDEAL}_{f_{\mathsf{part-product-2.0}}, \mathsf{Corr}, \mathcal{S}}(1^\lambda, (\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_k \rrbracket))$, corresponding the functionality $f_{\mathsf{part-product-2.0}}$ is statistically close to $\mathsf{REAL}_{\pi_{\mathsf{part-prod-2.0}}, \mathsf{Corr}, \mathcal{A}}(1^\lambda, (\llbracket \mathbf{x}_1 \rrbracket, \ldots, \llbracket \mathbf{x}_k \rrbracket))$, where the parties are all given access to the ideal functionality $f_{\mathsf{pack-mult}}$. □