DFS: Delegation-friendly zkSNARK and Private Delegation of Provers

Yuncong Hu [*] Pratyush Mishra(✉) [†] Xiao Wang [‡] Jie Xie [§] Kang Yang(✉) [¶] Yu Yu [‖]
Yuwen Zhang [**]

Abstract

Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARKs) lead to proofs that can be succinctly verified but require huge computational resources to generate. Prior systems outsource proof generation either through public delegation, which reveals the witness to the third party, or, more preferably, private delegation that keeps the witness hidden using multiparty computation (MPC). However, current private delegation schemes struggle with scalability and efficiency due to MPC inefficiencies, poor resource utilization, and suboptimal design of zkSNARK protocols.

In this paper, we introduce DFS, a new zkSNARK that is delegation-friendly for both public and private scenarios. Prior work focused on optimizing the MPC protocols for existing zkSNARKs, while DFS uses co-design between MPC and zkSNARK so that the protocol is efficient for both distributed computing and MPC. In particular, DFS achieves linear prover time and logarithmic verification cost in the non-delegated setting. For private delegation, DFS introduces a scheme with zero communication overhead in MPC and achieves malicious security for free, which results in logarithmic overall communication; while prior work required linear communication. Our evaluation shows that DFS is as efficient as state-of-the-art zkSNARKs in public delegation; when used for private delegation, it scales better than previous work. In particular, for 2^{24} constraints, the total communication of DFS is less than 500KB, while prior work incurs 300GB, which is linear to the circuit size. Additionally, we identify and address a security flaw in prior work, EOS (USENIX'23).

1 Introduction

Zero Knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARKs) allow a prover to attest the knowledge of a witness that satisfies any given NP relation. The prover only needs to send a short proof to the verifier, and without any further communication, the verifier will be convinced that the prover has a satisfying witness, without learning anything about the prover's secret witness. Their flexibility has led to numerous academic [4, 10, 90, 46] and industrial applications [89, 68, 82]. However, for many zkSNARK protocols, proof generation for complex statements incurs high compute and memory overheads. In applications where the prover might be resource-bounded (e.g., a mobile phone), these overheads can make it impossible to efficiently generate zkSNARK proofs, limiting the scope of applications.

To address this issue, prior work [83, 84, 58, 25, 39] has proposed **proof delegation** schemes where the prover outsources the proof generation process to third parties with access to more computational capacity. These schemes come in two flavors: *public* delegation, where the prover's witness is revealed to the third party, and *private* delegation, where the prover's witness remains hidden. In this work, we will be primarily concerned with constructing scalable private delegation schemes.

Why hide the witness? Private delegation is motivated by applications that touch sensitive user data. For example, in systems for private financial transactions [4] or private personal identity verification [33, 71], the witness contains sensitive information about the user's financial habits and identity, and revealing this data to the third party weakens the security guarantees of the system.

Prior work on private delegation. To prevent this leakage, recent work [25, 59, 39, 64] has constructed private delegation schemes in which the prover leverages secure multi-party computation (MPC) techniques [70, 43, 2, 30] to outsource proof generation to multiple (non-colluding) third *parties*, who jointly but *privately* compute the proof. Unfortunately, as we discuss in Section 7, while these works are able to provide strong privacy guarantees, they are unable to scale to proving large computations effectively. This is due to a number of reasons:

- **Inefficiencies due to MPC:** Despite efforts to specialize MPC techniques for the specific zkSNARKs, the resulting private delegation schemes still incur significant communication among multiple parties and computation overheads due to inherent inefficiencies of the choice of MPC protocol. For example, zkSAAS [39] uses packed secret sharing, which is not compatible with Pippenger's algorithm [6], a common method for optimizing expensive multi-scalar multiplications.

[*] Shanghai Jiao Tong University, huyuncong@sjtu.edu.cn
[†] University of Pennsylvania, prat@seas.upenn.edu
[‡] Northwestern University, wangxiao@northwestern.edu
[§] Shanghai Jiao Tong University, xiejie1006@gmail.com
[¶] State Key Laboratory of Cryptology, yangk@sklc.org
[‖] Shanghai Jiao Tong University & Shanghai Qi Zhi Institute, yyuu@sjtu.edu.cn
[**] UC Berkeley, yuwen01@berkeley.edu

- **Inefficiencies due to poor resource utilization:** Existing schemes [64, 25, 39] cannot effectively scale: they are bottlenecked by the computational resources of a single machine, and cannot take advantage of distributed computing techniques that would allow proof generation to be parallelized across multiple machines.
- **Inefficiencies from choice of zkSNARK:** Existing protocols target zkSNARKs that, for various reasons, are not well-suited for delegation. For example, zkSNARKs like Plonk [38], which have been the target of prior delegation schemes [39, 64], rely extensively on product check that requires expensive communication in MPC settings.

Our goal is to address all these concerns and construct *horizontally-scalable private delegation schemes* for zkSNARKs by combining distributed computations and MPC. We say that a delegation scheme is scalable if the proving time can be reduced by endowing each party with more nodes. We scale delegation to larger instances by adding nodes within each party, not by adding parties. We believe this is reasonable because we envision that each party will be a non-colluding cloud provider, and it is easier to add nodes within providers than to find new providers. Moreover, only two to three non-colluding clouds are typically available in practice.

1.1 Our contribution

We make progress towards this goal via several contributions.

First, we observed that previous work [64, 39] are unable to avoid linear communication cost in the private delegation setting, primarily due to two drawbacks of the underlying zkSNARKs: 1. random memory access operations, such as FFT, which create bottlenecks in distributed computing communication; 2. the presence of multi-layer multiplication gates hinders the efficiency of MPC communication, making it difficult to reduce the overall overhead in MPC protocols.

Second, we observe that when MPC protocols can achieve security up to additive attacks [40], applying them to zkSNARKs with a *single non-linear* layer simplifies the design of the delegation protocol. This simplification arises because we can eliminate the computation-extensive check for multiplications, as zkSNARKs are publicly verifiable and inherently prevent integrity attacks. As a result, we can achieve malicious security for free in this setting, making the protocol both more efficient and secure without the overhead of extra check. EOS [25] claims to achieve a similar property, but their protocols do not satisfy this condition, leading to vulnerabilities that could be exploited by adversaries.

Finally, we provide a systematic analysis of common zkSNARK building blocks that identifies how suitable they are for distributed computing and different MPC techniques. Based on this analysis, we identified two key techniques as being particularly friendly to distributed MPC computation: holography [24, 73] and multilinear sumchecks [61].

We leveraged these insights to construct DFS, a new zkSNARK for Rank-1 Constraint Systems (R1CS) with the universal setup that achieves several attractive properties:

- In the non-delegated (single machine) setting, DFS achieves linear prover time and logarithmic verification costs, which matches the state-of-the-art linear-time prover SNARKs that are scalable to large statements.
- In the private delegation setting, we construct a new private delegation scheme for DFS that achieves *zero communication overhead* in MPC and malicious security for free. This is a significant improvement over existing schemes [25, 64, 39, 59], which incur linear communication overhead. Moreover, privately delegated DFS leverages distributed computation to accelerate the local computation of each party. As a result, the overall communication cost is logarithmic, and with the increase in the number of computation nodes for each party, the total latency decreases linearly.
- We also apply DFS to the public delegation setting, which efficiently leverages distributed computing resources to optimize proof generation with logarithmic communications.

We implement DFS and its delegation schemes in a modular Rust library, and evaluate its scalability and performance. Our implementation supports two types of secret sharing schemes: Additive Secret Sharing (AddSS) and Replicated Secret Sharing (RSS). We assume three parties for RSS, and two parties for AddSS. These parties could naturally model the prominent cloud computing platforms: AWS, GCP, and Microsoft Azure. As discussed in our scalability goal, we do not expect the need for more parties, as each party represents a distinct non-colluding cloud platform, and such platforms are limited in practice. Instead, we focus on deploying more nodes within each platform to accelerate proof generation.

Our experiments in Section 6 show that (a) DFS's prover time scales linearly with the circuit size, and verification is less than 50ms; (b) our private delegation scheme can scale to larger circuits than prior work with lower communication and latency overheads. For 2^{24} constraints, DFS achieves less than 500KB of communication, while prior work [39] incurs 300GB. (c) our public delegation scheme achieves similar scalability to the prior state-of-the-art Pianist [58], while Pianist is not suitable for private delegation.

Finally as an additional minor contribution, we identify a security flaw in prior work [25] and show that it does not achieve the claimed malicious-security guarantees.

Private Delegation with Different Schemes. DFS's private delegation can be instantiated with different secret-sharing schemes, each offering distinct trade-offs. Due to DFS's single multiplication layer in the private phase, using RSS eliminates the need for inter-party MPC communication and does not require authenticated shares for malicious security. However, RSS is limited to honest-majority settings with three parties. To support more parties, Shamir Secret Sharing (SSS) can be used, which also benefits from DFS's properties, providing similar advantages to RSS while scaling to more parties. AddSS supports dishonest-majority settings

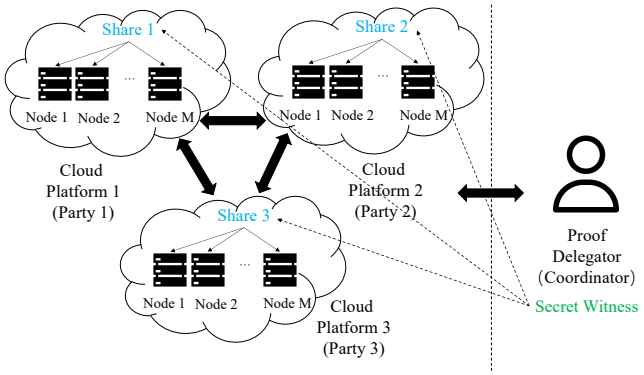

Figure 1: System overview of DFS.

but requires authenticated shares for malicious security, and inter-party communication scales linearly with the instance size. Therefore, we mainly use AddSS in two-party settings, though it can be extended to more parties if required. All of these secret-sharing schemes can be accelerated by our distributed computation designs. Lastly, we did not consider Packed Secret Sharing (PSS) due to its inefficiency in multi-scalar multiplication (MSM) computations and because DFS optimizes proof generation by increasing the number of nodes within parties, rather than increasing the number of parties.

While in this paper we focused on delegating our new SNARK, our techniques enable efficient delegation of all SNARKs whose private (witness-dependent) phase requires only a single layer of multiplications. This class includes popular SNARKs like Spartan [73], Marlin [24] and Lunar [21].

2 Technical overview

System architecture. As shown in Fig. 1, we consider a scenario where a delegator $\mathcal{D}$ wants to outsource the proof generation to the parties $\mathcal{P}_1, \ldots \mathcal{P}_{N_p}$. We consider each party as an independent trust domain, such as different cloud service platforms, geographically distributed computation nodes, or computing resources managed by different organizations. Each party has multiple nodes, enabling it to parallelize its local computations and accelerate the proof generation process.

The delegator $\mathcal{D}$ shares its witness $\boldsymbol{w}$ into N_p shares $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{N_p}$, and sending each share to the respective party. The N_p parties then run a MPC protocol to compute the proof. For efficiency, each party internally parallelizes its computation across a cluster of nodes. For example, the i-th share $\boldsymbol{w}_i$ is split into N_w chunks $w_i^{(1)}, \ldots, w_i^{(N_w)}$, which are distributed across N_w nodes within the party $\mathcal{P}_i$. The delegator also coordinates the proof generation among different parties, but its workload is designed to be logarithmic; thus, even users with limited resources can effectively delegate the proof.

In the public-delegation setting, all nodes are controlled by a single party, which can see the witness, and $\mathcal{D}$ can ask any node to serve as the coordinator.

Threat model. We focus on designing private-delegation protocols in the presence of a malicious adversary that can deviate from the protocol in an arbitrary way. We assume that all the nodes controlled by the same party are part of the same "trust" domain: if this party is corrupted, then all the nodes owned by this party are malicious. We envision that the parties in our protocol will be instantiated with different cloud computing platforms. An attacker who corrupts a cloud platform is more likely to corrupt all the nodes on that platform, but not the nodes on the other platforms.

Depending on the kind of underlying MPC protocol, we target two different settings: the two-party setting and the three-party setting, where at most one party is allowed to be corrupted in both settings. Note that our protocol can be extended to support more parties.

To construct such a scheme, we begin by analyzing whether the cryptographic building blocks we rely on (MPC protocols and components of zkSNARKs) achieve these goals.

2.1 Scalability of MPC

We analyze three linear secret-sharing schemes that have been commonly used in prior MPC protocols for their efficiency properties. We first note that all linear secret-sharing schemes do not require communication for linear operations, and thus eliminate communication overhead for these operations.

Additive secret sharing (AddSS) is an efficient linear secret-sharing scheme used by many MPC protocols [30]. AddSS supports all-but-one dishonest majority security. Multiplication for AddSS relies on the "Beaver's triples", which requires cryptographic operations to generate some preprocessing material for the parties, and requires each party to communicate $n-1$ field elements to evaluate the multiplication gate. This means that if the zkSNARK prover requires a number of multiplications that is linear in the size of the circuit, the communication overhead from AddSS will be linear as well.

Packed secret sharing (PSS) [35] is a generalization of Shamir secret sharing (SSS) [76] that packs multiple secrets into a single share. The corruption threshold supported by PSS depends on the number of packed secrets; if k secrets are packed into a single share, then the corruption threshold can be at most $n-k$. PSS supports (a bounded number of) cheap multiplication operations: simply multiply the shares. However, not all linear operations are cheap in PSS: operations on secrets within the same packed share require unpacking, which involves reconstructing and redistributing the individual secrets, leading to additional communication. Moreover, this also inhibits important algorithmic optimizations like using the Pippenger algorithm [67] for multi-scalar multiplication, thus incurring high computational overheads as well.

Replicated secret sharing (RSS) [62] offers different trade-offs compared to AddSS and PSS. RSS only provides security in an honest-majority setting, and furthermore incurs a $2\times$ overhead for computing linear gates whose output is used

in further secret computations. However, when the output is made *public* (i.e., not used in further secret computations), RSS can compute the linear gate with no overhead. Furthermore, unlike AddSS and PSS, the same benefit applies to *multiplication* gates as well: they incur neither communication nor computation overhead when the gate output is public. To the best extent of our knowledge, the latter observation is novel, and might be of independent interest in other MPC applications.[1] Finally, we also observe that RSS achieves malicious security for *free* in our setting.

Summary. The inefficiency of PSS for common and expensive operations like MSMs makes RSS and AddSS the most efficient choices for our private delegation scheme.

Security up to additive attacks. MPC protocols, which guarantee security up to additive attacks in the presence of malicious adversaries, imply that the only effective attack is to add adversarial-known errors at the output of multiplication gates. Previous work [40] has identified that a class of MPC protocols satisfy this property. We can apply such MPC protocols to realize private delegation of zkSNARKs without the need of checking multiplication gates that is computationally expensive, since zkSNARKs are publicly verifiable and guarantee integrity.

If we remove the multiplication check, we need to require that there is only one layer of multiplication gates. If there are multiple layers of multiplication gates, additive errors at the outputs of multiplication gates may be introduced to the evaluation of subsequent multiplication gates, which brings about the cross terms between errors and secrets. This allows the adversary to perform selective-failure attacks (i.e., it can guess whether the equation involving the errors and secrets is zero), which leak one-bit information for each protocol execution. Conversely, if there is only one layer of multiplication gates, the errors only affect the integrity but not the privacy.

We find that this property can simplify the design of private-delegation protocols. Specifically, when a zkSNARK has only one layer of multiplication gates, we can employ an MPC protocol that guarantees security up to additive attacks against malicious adversaries to realize private delegation. At the end of protocol execution, we can use the public verifiability of zkSNARKs and run the verification algorithm to guarantee integrity, which eliminates the computation-expensive check of multiplication gates. In this way, we can achieve the performance similar to semi-honest protocols, i.e., guarantee malicious security for free.

Previous work, such as EOS [25], attempted to remove the check of multiplication gates, but failed because their MPC protocol did not satisfy this property, resulting in privacy leakage. We will discuss this issue in more detail in Section 2.4.

[1] In fact, the same benefit applies also to Shamir secret sharing [76], but for concreteness we focus on RSS as it achieves better efficiency.

2.2 Scalability of zkSNARKs

We now analyze common building blocks used in zkSNARK constructions, focusing on SNARKs constructed via the PIOP + PC scheme methodology [24, 19]. Chiesa et al. [24] introduced a methodology for constructing zkSNARKs from two components: polynomial interactive oracle proofs (PIOPs) [19] and polynomial commitment (PC) schemes [51]. The methodology works as follows: the PC scheme is used to commit to all (indexer and prover) oracles as they are computed, and then the PC scheme's opening and verification algorithms are used to prove the correctness of the evaluations. It is straightforward to see that the properties of the compiled zkSNARK is determined by the underlying PIOP and PC schemes, so we now focus on analyzing the efficiency of these components.

We first analyze the scalability of PIOPs. There are a myriad of PIOP constructions in the literature, and it is infeasible to analyze all of them individually. Instead, we identify core building blocks subPIOPs that underlie most of these PIOPs, and analyze their communication and computation overhead.

Product-check PIOPs. Many popular zkSNARKs [38, 23] rely on a subprotocol that checks that the product of the entries of a vector v equals a claimed value c, i.e., that $\prod_i v_i = c$. Observe that proving this claim requires computing the product $\prod_i v_i$. This is straightforward in the single-prover setting, and in the public delegation, the computation can be distributed across nodes effectively: give to the nodes equal-sized partitions of v, have them compute the product of their partition, and then have all nodes send their partial products to the delegator, who then computes the final product.

In the private delegation setting, however, these strategies do not work. When the vector v consists of secret-shared values (e.g., when the vector is the witness), computing the product requires computing a product of n secret-shared values. Despite optimizations, the best known protocol for this task requires $O(n)$ inter-party communication. Not only does this violate our target of sublinear communication, but prior work [64] observes that this additional communication leads to bottlenecks in performance. Moreover, the product introduces multiple layers of multiplication gates, making it impossible to achieve malicious security for free. Note that prior works [23, 75] use sumcheck to prove product relations, avoiding direct product-checks. However, computing intermediate polynomials still involves multiple multiplication layers, leading to unavoidable MPC overhead.

Hence, we deem product-check subPIOPs and the PIOPs that rely on them unsuitable for private delegation.

Sumcheck PIOPs. A second class of popular zkSNARKs [24, 73, 21, 9] relies instead on the sumcheck protocol [61], which checks that the *sum* of the entries of a vector v (represented as polynomials) equals a claimed value c, i.e., $\sum_i v_i = c$. There are two kinds of sumcheck protocols in the literature: univariate sumcheck and multilinear sumcheck.

Univariate sumcheck requires quasilinear proving time due its use of FFTs for polynomial multiplications and divisions, and distributed computation of FFTs is known to require linear communication [83], and furthermore, provides diminishing returns as the number of nodes increases. Therefore, univariate sumcheck is unsuitable for delegation.

For multilinear sumcheck, on the other hand, there are efficient linear-time prover algorithms for sumcheck [80, 78]. Moreover, the operations performed by the prover in these algorithms are embarrassingly parallel, and thus can be distributed across nodes effectively in delegation settings.

Finally, in the private delegation setting, when performing sumcheck over a multilinear polynomial, these same operations are linear, and thus do not require communication in MPC. However, when performing sumcheck over a *product* of multilinear polynomials, this does require communication in MPC. However, we observe that if the sumcheck only involves a single-layer of inner-product, then the protocol can be performed in RSS without communication. Thus, we conclude that multilinear sumcheck-based PIOPs are ideal candidates for private delegation.

PIOPs for table lookups have emerged as a key tool for improving the efficiency of several recent zkSNARKs [37, 66, 36, 34, 74, 88, 48]. These protocols assert that the entries of a vector are contained in predefined tables, and have been used to reduce the overhead of circuit operations that have traditionally been regarded as expensive in zkSNARKs, such as bitwise operations, integer operations, comparisons, and more. Recently, they have also been used to directly improve the efficiency of PIOP constructions [73, 74].

Unfortunately, not all lookup protocols are suitable for delegation. For example, the widely-used plookup [37] protocol requires product checks, which incur high costs. Yet another class of lookup protocols, such as Lasso [74, 73], based on time-stamping and offline memory-checking [7] has shown excellent performance in the single-prover setting. However, previous work [9] has indicated that offline memory-checking requires random memory access, which is not suitable for distributed computing environments.

We observe that a recently proposed lookup protocol, LogUp [48] is able to avoid these drawbacks by relying on multilinear sumcheck. LogUp allows distributed computation to avoid additional communication between nodes, as its operations are inherently parallelizable and can be executed locally within each node. In Section 5 we show how to leverage this fact to distribute the prover's computation across multiple nodes while minimizing communication overhead.

It is important to note that all lookup protocols, including LogUp, cannot avoid multiple layers of multiplication gates when computing intermediate polynomials. As a result, applying lookup protocols in private delegation presents challenges. However, as we designed in Section 2.3, we can separate the witness-dependent and witness-independent phases of the zkSNARK. By applying the lookup protocol only to the witness-independent portion, we ensure that multiple multiplication layers do not appear in the MPC phase, enabling more efficient private delegation.

Scalability of polynomial commitment schemes. There have been numerous constructions of PC schemes in recent work, spanning a variety of assumptions and efficiency. Following prior work on MPC-friendly zkSNARKs [64, 25], we focus on PC schemes based on pairing-friendly curves [51, 65], as the core component of their commitment and opening algorithms, namely elliptic-curve multi-scalar multiplications (MSMs), are linear operators. This means that committing to and opening secret-shared polynomials does not require communication under MPC. Furthermore, because the MSM is easily partitioned and parallelized [6, 92, 60], it is well-suited for distributed computing. Another popular class of PC schemes consists of those based on error-correcting codes [3, 86, 44]. These achieve attractive properties such as post-quantum security, but their algorithms require constructing and opening Merkle trees, which are highly nonlinear operations, and are hence expensive to perform in MPC.

2.3 Delegation-friendly zkSNARKs

Our goal is to design a zkSNARK protocol that is friendly to both distributed computing and multi-party computation, enabling it to work effectively in both public and private delegation scenarios.

Starting point: an observation about Marlin. We start by recalling an observation made in prior work on delegating SNARKs [64, 25]: the Marlin zkSNARK [24] proved to be more efficiently delegable than other common zkSNARKs such as Groth16 [45] and Plonk [38]. These works attribute this to the fact that Marlin's protocol can be divided into two parts: a witness-dependent portion (non-holography), and a witness-independent portion (holography). While the witness-dependent portion clearly requires MPC to protect witness privacy, the witness-independent portion only involves public data, and hence can be computed *without MPC*. Because in practice the witness-independent portion is the most time-consuming part of the protocol, eliminating the need to use MPC allowed prior work to derive significant improvements.

Not all protocols satisfy the holography property, and thus the whole protocol is witness-dependent. As a result, these protocols [23, 38] unavoidably involve multiple layers of multiplication gates for computing intermediate polynomials in private delegation, leading to increased MPC overhead.

Our approach. Marlin is a good starting point for constructing DFS. Unfortunately, as noted in Section 2.2, Marlin relies on univariate sumcheck and is hence not suitable for scalable delegation. So instead we take as our starting point the multilinear adaptation of Marlin, namely Spartan [73]. Like Marlin, Spartan's protocol can also be divided into witness-dependent and witness-independent portions. However, Spartan's pro-

tocol uses as a crucial component a lookup PIOP that relies on offline memory-checking, and as noted in Section 2.2, this protocol is not scalable. We show how to fix this issue and construct a delegation-friendly zkSNARK. DFS's PIOP for R1CS follows the lead of the holographic PIOPs and proves R1CS by breaking it up into three subrelations: a "rowcheck", a "lincheck", and a matrix-evaluation check, shown in Fig. 2.

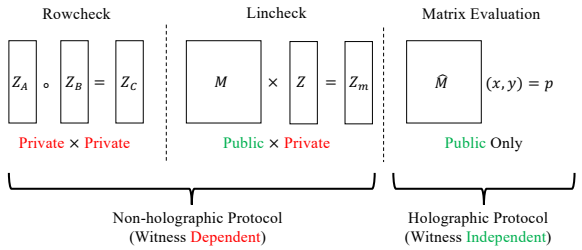

Figure 2: Protocol overview of zkSNARKs for R1CS. $M \in \{A,B,C\}$ indicates the R1CS matrices, $\boldsymbol{z}$ indicates the witness vector, $\boldsymbol{z}_M$ indicates the result of multiplying M by $\boldsymbol{z}$, and $\hat{M}$ indicates the polynomial encoding of M. $\boldsymbol{z}, \boldsymbol{z}_M$ are private, and $M, \hat{M}, x, y, p$ are public.

The protocol starts with the indexer computing polynomial encodings of the R1CS matrices. Then, the prover computes polynomials $\boldsymbol{z}_A, \boldsymbol{z}_B, \boldsymbol{z}_C$ claiming to equal the matrix vector products $A\boldsymbol{z}$, $B\boldsymbol{z}$, and $C\boldsymbol{z}$, and proves that these satisfy the Hadamard product relation $A\boldsymbol{z} \circ B\boldsymbol{z} = C\boldsymbol{z}$ (this is the rowcheck step). DFS reduces the rowcheck to a sumcheck over products of polynomials, which requires only a *single layer of multiplications* between secret-shared values.

After this, the prover is then left to show that for each $M \in \{A,B,C\}$, $\boldsymbol{z}_M$ indeed equals $M\boldsymbol{z}$. This is the lincheck step, and, as illustrated in Fig. 2, it involves linear checks between the public matrix M and the private vectors $\boldsymbol{z}_M$ and $\boldsymbol{z}$. To prove this step, DFS uses sumcheck (this time only over a *single private polynomial*) to reduce the check to a claim about evaluations of polynomials encoding the matrices M.

Spartan uses offline memory-checking technique to express this claim as a specialized circuit and then uses an external proof system to prove this circuit. However, this technique has two drawbacks. First, as we discussed in Section 2.2, offline memory-checking may require random memory accesses, which are not scalable. Second, the external proof system is required to be delegation-friendly, which leads to a circular dependency. We instead express the matrix evaluations as a lookup relations, and then use the sumcheck-based lookup algorithm LogUp [48] to prove the claim directly. LogUp not only avoids the random memory accesses required by offline memory-checking, but also performs simple and parallelizable operations over the involved polynomials before producing a sumcheck claim. We observe that this matrix-evaluation check is witness-independent and involve *only public data*; thus, it can be computed without MPC.

We instantiate DFS with the PST13 polynomial commitment scheme [65], which is scalable and MPC-friendly.

Distributed computation. DFS is highly efficient for distributed computing because the protocol design deliberately avoids operations such as FFT and memory-checking, which require random memory access. To construct a public delegation scheme, we designed distributed computation algorithms for the PIOP and polynomial commitment that require only logarithmic communication overhead, shown in Appendix E.

2.4 Private delegation for DFS

We first identify a security issue in the recent private-delegation protocol EOS [25] based on additive secret sharing (AddSS). Then, we show how to fix it, and present our improved approach in the two-party setting. In the three-party setting with at most one malicious corruption, we present a simpler and more efficient approach to delegate the proof generation of DFS using replicated secret sharing (RSS). The RSS-based protocol can achieve *zero communication overhead* and malicious security *for free*. When the number of parties is two or three, and every party has a cluster of nodes, our private-delegation protocols are more efficient than the recent protocols [64, 39, 59] and the improved EOS, due to the MPC-friendly design of DFS and the optimized techniques exploiting the specific private-delegation application.

A security issue in EOS. The recent work EOS [25] adopts AddSS to realize private delegation of zkSNARKs in the dishonest-majority *malicious* setting. It uses the technique of Beaver triples [1] to compute multiplications. In particular, given two additive secret sharings $\langle x \rangle$ and $\langle y \rangle$, the parties need to compute an additive secret sharing $\langle z \rangle$ with $z = x \cdot y$. Given a Beaver triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ with two random elements a, b and $c = a \cdot b$ from a delegator $\mathcal{D}$, all parties run the standard protocol to open $\langle x \rangle - \langle a \rangle$ and $\langle y \rangle - \langle b \rangle$ to obtain two public values $\mu = x - a$ and $\nu = y - b$. Then, the parties compute $\langle z \rangle = \mu \cdot \nu + \nu \cdot \langle a \rangle + \mu \cdot \langle b \rangle + [c]$ based on the linear property of AddSS. The proof is verified by $\mathcal{D}$ to guarantee the correctness of multiplication computation.

EOS claims that the opened values do not require authentication, thereby reducing the cost. We observe that a malicious adversary can introduce two errors e_0, e_1 during the open procedure, so that $\mu' = \mu + e_0$ and $\nu' = \nu + e_1$ are computed. When z is reconstructed by $\mathcal{D}$ using the shares sent by the parties, an additional error e_2 can be introduced by the adversary. In this case, the parties actually obtain a value $z' = z + (e_0 \cdot b + e_1 \cdot a + e_2)$. If the resulting proof passes the verification, then $e_0 \cdot b + e_1 \cdot a + e_2 = 0$, else it is not zero. This leaks one-bit information on a, b, which in turn reveals the information on secret values x, y, due to the fact that $\mu = x - a$ and $\nu = y - b$ are publicly known. The above attack is referred to as a selective-failure attack in the MPC setting, and can be repeated to leak more information. We have reported this problem to the authors of EOS, and they will fix it.

MPC for DFS in the two-party setting. The crucial security issue of EOS is that the values to be opened are *not* authenticated. To solve the issue, we can equip additive secret sharings with SPDZ-like information-theoretic message authentication codes (IT-MACs) [30, 29], so that the values to be opened are authenticated with IT-MACs. Furthermore, we can adopt the recent VOLE protocols (e.g., [11, 13, 72, 14, 81, 12, 47, 69]) to compress random IT-MACs into short seeds such that the communication between $\mathcal{D}$ and all parties is sublinear in the number of IT-MACs. We can use the batch-check technique [30, 29] to realize authenticated opening, which brings a negligibly small communication overhead.

If we transform a Beaver triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ into an authenticated triple denoted by $([\![a]\!], [\![b]\!], [\![c]\!])$, this will incur an expensive computation cost for all parties, according to the state-of-the-art protocols [15, 31, 12, 8] for generating authenticated triples. We overcome the efficiency problem by letting D generate partially authenticated triples $(([\![a]\!], [\![b]\!], \langle c\rangle)$, i.e., $c = a \cdot b$ is not authenticated with IT-MACs), where the input shares are authenticated but the output shares are not. This allows the parties to compute an unauthenticated sharing $\langle z\rangle$, which is not a problem when there is only one layer of multiplications in zkSNARKs such as DFS. Therefore, it does not allow the adversary to introduce an error, leading to a cross term between the error and secret, and thus prevents selective-failure attacks. Our zkSNARK scheme DFS uses the inner product instead of Hadamard product as the unique non-linear operations. In this case, we are able to generalize the above approach to handle inner products by generating a semi-authenticated inner-product tuple, i.e., $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], \langle c\rangle)$, where c is the inner product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.

MPC for DFS in the three-party setting. When at most one of three parties is corrupted, we use replicated secret sharings to give a simpler and more efficient solution. Specifically, due to the linear property, all linear operations are communication-free. In DFS, the only non-linear operations are a single layer of inner products. We use the multiplication property of RSS to let all parties locally compute the inner-product operation. In this case, RSS will be converted into AddSS. Different from the general MPC setting, we do not choose to convert AddSS back into RSS. The reason behind this is that all inner-product operations only occur in **a single** layer, no non-linear operation between two additive secret sharings is required. Therefore, we achieve **zero communication** for inner-product operations. This idea can be applied to any non-linear operations, as long as there is a single layer. To obtain the simulation-based security, we let all parties randomize the resulting sharings with fresh zero sharings, so that the shares of honest parties are still uniform under the condition that the shares could be reconstructed as the secret.

Since the inner product of two RSSs results in an AddSS, and AddSS does not provide the authentication property (even in the honest-majority setting), we cannot guarantee the correctness of reconstruction of the inner-product result. Nevertheless, the reconstruction procedure does not allow the adversary to introduce an error leading to a cross term between the error and secret. Therefore, it is not possible to perform selective-failure attacks. We note that the above idea can also be applied to Shamir secret sharings, which support more parties. In conclusion, our approach only allows a malicious adversary to mount additive attacks, and guarantees privacy in the presence of malicious adversaries. Through the public verifiability of zkSNARKs, we can guarantee the integrity, and achieve malicious security for free.

Distributed computation of private delegation. We employ distributed computing to accelerate the MPC operations performed by each party, thereby achieving the overall speedup. This approach is highly effective because, in DFS, most operations are linear and can be computed locally without communication between parties. The only operation that requires the communication is the inner product. However, we have accounted for this in the design of DFS, i.e., there is only one layer of inner-product operations during the sumcheck for the product of multiple polynomials. In the three-party setting, we can achieve zero communication overhead for this step, allowing all computations to be executed locally within each party. Each party can then leverage its local cluster of nodes to parallelize local computations, thus ensuring scalability without inter-party communications.

Another key advantage of DFS is the holography property, as illustrated in Fig. 2, which ensures that the final phase involves purely public computation. This means that we do not need to rely on MPC to protect the witness during this phase, allowing for public delegation instead. Since secret sharings are no longer required, we can switch from private delegation to public delegation, enabling the use of all nodes from all parties for distributed computation. This maximizes the utilization of computational resources. Our experiments show that, in the non-delegation setting, the holographic portion accounts for 70% of the proof generation time. This indicates that even in the private-delegation setting, we can accelerate the majority of computations purely through distributed computing, without the additional overhead of MPC.

3 Preliminaries

Notation. We use λ to denote the security parameter, and $\mathbb{F}$ to denote the prime field. We denote by $[a \dots b]$ the set $\{a, \dots, b\}$ for $a, b \in \mathbb{N}$; particularly, we use $[1 \dots n]$ to denote the set $\{1, \dots, n\}$ for some $n \in \mathbb{N}$. We use $\mathbb{F}$ to denote a finite field $\mathbb{Z}_q$ of prime order q. We will use bold letters like $\boldsymbol{x}$ for vectors, and denote by $\boldsymbol{x}_i$ the i-th component of $\boldsymbol{x}$ with $\boldsymbol{x}_1$ the first entry. We also use colon notation to denote slices of vectors. For example, $\boldsymbol{x}[i : j]$ denotes a vector constituting of $\boldsymbol{x}_i, \dots, \boldsymbol{x}_j$. We will occasionally explicitly denote the multilinear extension of a vector $\boldsymbol{a} \in \mathbb{F}^N$ using the notation $\boldsymbol{a}(\boldsymbol{x}) \in \mathbb{F}[\boldsymbol{x}]$ where $\boldsymbol{x} \in \{0,1\}^{\log N}$, and for example, $\boldsymbol{a}(0) = \boldsymbol{a}_0$.

We use upper-case letters A, B, C, M to denote matrices. For the matrix $M \in \mathbb{F}^{n \times m}$, we denote by $\boldsymbol{M}(x, y)$ the multilinear extension of M to $\mathbb{F}[\boldsymbol{x}, \boldsymbol{y}]$, where $\boldsymbol{x} \in \mathbb{F}^{\log n}, \boldsymbol{y} \in \mathbb{F}^{\log m}$.

Algebraic background. Throughout this paper, we will work with the n-dimensional Boolean hypercube $\{0,1\}^n$. The polynomial $\mathrm{eq}(\boldsymbol{x}, \boldsymbol{y}) := \prod_{i=1}^{n}(\boldsymbol{x}_i \boldsymbol{y}_i + (1-\boldsymbol{x}_i)(1-\boldsymbol{y}_i))$ checks that $\boldsymbol{x} = \boldsymbol{y}$.

Definition 3.1 (Rank-1 Constraint Systems)**.** *The indexed relation $\mathscr{R}_{\mathrm{R1CS}}$ is the set of all triples*

$$(\mathbb{i}, \mathbb{x}, \mathbb{w}) = \big((\mathbb{F}, n, m, A, B, C), x, w\big)$$

where $\mathbb{F}$ is a finite field, n and m are natural numbers, $w \in \mathbb{F}^{m-|x|-1}$ is a vector over $\mathbb{F}$, A, B, C are $m \times m$ matrices over $\mathbb{F}$ with at most n nonzero entries, such that $A\boldsymbol{z} \circ B\boldsymbol{z} = C\boldsymbol{z}$, where $\boldsymbol{z} := (x \mid\mid 1 \mid\mid w)$. Let $s = \log m$ and $d = \log n$.

Sparse matrix encodings. Prior work [24, 26, 21] has shown how to represent (or arithmetize) a *square* matrix $M \in \mathbb{F}^{N \times N}$ with N non-zero entries via three multilinear polynomials $\boldsymbol{r}_M$, $\boldsymbol{c}_M$, $\boldsymbol{v}_M$, such that $\boldsymbol{v}_M(\boldsymbol{x}) = M_{\boldsymbol{r}_M(\boldsymbol{x}), \boldsymbol{c}_M(\boldsymbol{x})}$, where $\boldsymbol{x} \in \{0,1\}^{\log N}$. If we map the matrix index (i, j) to the vector $(\boldsymbol{i}, \boldsymbol{j})$, we can define the function $\hat{\boldsymbol{r}_M}(\boldsymbol{x}) = \boldsymbol{i}$ and $\hat{\boldsymbol{c}_M}(\boldsymbol{x}) = \boldsymbol{j}$. Then the matrix encoding of M is the multilinear polynomial $\boldsymbol{M}(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{x}} \boldsymbol{v}_M(\boldsymbol{x}) \mathrm{eq}(\hat{\boldsymbol{r}_M}(\boldsymbol{x}), \boldsymbol{i}), \mathrm{eq}(\hat{\boldsymbol{c}_M}(\boldsymbol{x}), \boldsymbol{j})$.

3.1 Background for zkSNARKs

A *succinct preprocessing non-interactive argument of knowledge* in the random oracle model (ROM) for an indexed relation $\mathscr{R}$ is a tuple of algorithms $\mathsf{ARG} = (\mathcal{G}, I, \mathcal{P}, \mathcal{V})$ satisfying completeness, knowledge soundness, succinctness, and zero-knowledge. The indexer I preprocesses the NP index $\mathbb{i}$ into index-specific proving (ipk) and verification (ivk) keys. The prover $\mathcal{P}$, on input ipk, an instance $\mathbb{x}$, and a witness $\mathbb{w}$ such that $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathscr{R}$, outputs a proof π which can be checked by the verifier $\mathcal{V}$ when given ivk and $\mathbb{x}$.

Polynomial interactive oracle proofs. A polynomial interactive oracle proof (PIOP) for an indexed relation $\mathscr{R}$ is an interactive protocol specified by a tuple $\mathsf{PIOP} = (\mathbb{F}, \mathsf{k}, \mathsf{s}, I, \mathcal{P}, \mathcal{V})$ where $\mathbb{F}$ is a finite field, k is the number of rounds, $\mathsf{s}(j)$ is the number of prover polynomials in the j-th round, and I, $\mathcal{P}$, $\mathcal{V}$ are algorithms described next.

Polynomial commitments. A polynomial commitment scheme enables a sender to commit to a polynomial p and then later prove the correct evaluation of p at a desired point. Formally, it is a tuple of algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Trim}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Check})$ satisfying completeness, extractability, and hiding properties.

3.2 Background for MPC

We will use linear secret sharing schemes (LSSSs) to realize private delegation of DFS. A t-out-of-n LSSS enables a secret x to be shared among n parties, such that no subset of t parties is able to learn any information on x, while any subset of $t+1$ parties can reconstruct x.

Definition 3.2. *A linear secret sharing scheme has the following algorithms and protocols:*

- $[x] \leftarrow \mathsf{Share}(x)$*: A dealer $\mathcal{D}$ runs this algorithm to share a secret x among the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, such that $\mathcal{P}_i$ gets a share $[x]_i$ for $i \in [1 \ldots n]$. The sharing on x is denoted by $[x] = ([x]_1, \ldots, [x]_n)$.*
- $x \leftarrow \mathsf{Rec}([x], \mathcal{B})$*: Given at least $t+1$ shares in $[x]$, any party $\mathcal{B}$ (e.g., one of $\mathcal{P}_1, \ldots, \mathcal{P}_n$) can reconstruct the secret x. Although $t+1$ shares instead of n shares are sufficient to reconstruct the secret x, we still write $[x]$ as the input of* Rec *for simplifying the description.*
- LINEAR COMBINATION*: Given the public coefficients $c_0, c_1, \ldots, c_\ell$ and secret sharings $[x_1], \ldots, [x_\ell]$, parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ can* locally *compute $[y] = \sum_{i=1}^{\ell} c_i \cdot [x_i] + c_0$, such that $y = \sum_{i=1}^{\ell} c_i \cdot x_i + c_0$.*

We can also define a $\mathsf{Open}([x])$ protocol to open x to all parties, and $\mathsf{Open}([x])$ can be constructed by running $\mathsf{Rec}([x], \mathcal{P}_i)$ for each $i \in [1, n]$. For a vector $\boldsymbol{x} = (x_1, \ldots, x_m)$, we will use $[\boldsymbol{x}]$ to denote $([x_1], \ldots, [x_m])$. We use $[x]_i$ to denote the share held by a party $\mathcal{P}_i$. We present three common LSSS instantiations in Appendix F. We will use $[\![x]\!]$ to denote a secret sharing with the authentication property, i.e., the reconstruction of secrets can be verified (RSS and AddSS with authentication check), and $\langle x \rangle$ to denote an unauthenticated secret sharing.

4 DFS: a delegation-friendly zkSNARK

Before we start generating our proof, we first need to efficiently encode the R1CS matrices A, B, C. Each matrix $M \in \{A, B, C\}$ is expressed as three multilinear polynomials $\boldsymbol{r}_M$, $\boldsymbol{c}_M$, and $\boldsymbol{v}_M$, such that $\boldsymbol{v}_M(\boldsymbol{x}) = M_{\boldsymbol{r}_M(\boldsymbol{x}), \boldsymbol{c}_M(\boldsymbol{x})}$. Here, we focus on the non-zero elements of the matrix, ensuring that the polynomials are defined only over the positions corresponding to these non-zero entries. This encoding is handled by the indexer before the proof generation and can be reused as long as the R1CS matrices remain unchanged. The verifier is given oracle access to $\boldsymbol{r}_M$, $\boldsymbol{c}_M$, and $\boldsymbol{v}_M$.

To generate the proof, we start by defining the following,

$$\begin{pmatrix} \hat{A}(\boldsymbol{x}) := \sum_{\boldsymbol{y} \in \{0,1\}^s} \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{z}(\boldsymbol{y}) \\ \hat{B}(\boldsymbol{x}) := \sum_{\boldsymbol{y} \in \{0,1\}^s} \boldsymbol{B}(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{z}(\boldsymbol{y}) \\ \hat{C}(\boldsymbol{x}) := \sum_{\boldsymbol{y} \in \{0,1\}^s} \boldsymbol{C}(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{z}(\boldsymbol{y}) \end{pmatrix} \tag{1}$$

where $\boldsymbol{x} \in \{0,1\}^s$, and for $M \in \{A, B, C\}$, $\boldsymbol{M}$ is the sparse matrix encoding defined in Section 3. Notice that $\hat{A}$, $\hat{B}$, and $\hat{C}$ are respectively the multilinear extensions of $A\boldsymbol{z}$, $B\boldsymbol{z}$, and $C\boldsymbol{z}$. This means that $A\boldsymbol{z} \circ B\boldsymbol{z} = C\boldsymbol{z}$ only if the evaluations of the polynomial $\boldsymbol{F}(\boldsymbol{x}) := \hat{A}(\boldsymbol{x}) \cdot \hat{B}(\boldsymbol{x}) - \hat{C}(\boldsymbol{x})$ over the Boolean hypercube are all zero. This is a classical zerocheck claim, and so to check it, the verifier samples a randomness $\boldsymbol{r}$, which

reduces to proving the sumcheck $\sum_{\boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x})\text{eq}(\boldsymbol{x},\boldsymbol{r}) = 0$. At the end of the sumcheck protocol, the verifier checks that $e_x = \boldsymbol{F}(\boldsymbol{\rho_x})\text{eq}(\boldsymbol{\rho_x},\boldsymbol{r})$, where e_x is the final claim of the sumcheck protocol, and $\boldsymbol{\rho_x}$ is the sumcheck challenge. The verifier can evaluate the polynomial $\text{eq}(\boldsymbol{x},\boldsymbol{y})$ in logarithmic time. However, the evaluation of the polynomial $\boldsymbol{F}(\boldsymbol{x})$ needs the proof.

The prover will provide claimed evaluations $v_M := \hat{M}(\boldsymbol{\rho_x})$, where $M \in \{A,B,C\}$, and the verifier will check if $(v_A \cdot v_B - v_C)\text{eq}(\boldsymbol{\rho_x}, r) \stackrel{?}{=} e_x$. Next, in order to verify the authenticity of v_A, v_B, and v_C, we perform a batched sumcheck. The prover receives random challenges r_A, r_B and r_C from the verifier, and we batch $\boldsymbol{A}, \boldsymbol{B}$, and $\boldsymbol{C}$ together as $M_{\boldsymbol{\rho_x}}(\vec{y}) := r_A \cdot \boldsymbol{A}(\boldsymbol{\rho_x},\vec{y}) + r_B \cdot \boldsymbol{B}(\boldsymbol{\rho_x},\vec{y}) + r_C \cdot \boldsymbol{C}(\boldsymbol{\rho_x},\vec{y})$. Then, we perform a sumcheck for:

$$r_A v_A + r_B v_B + r_C v_C \stackrel{?}{=} \sum_{\vec{y}\in\{0,1\}^m} M_{\boldsymbol{\rho_x}}(\vec{y})\boldsymbol{z}(\vec{y}) \tag{2}$$

By the sumcheck protocol, this, too, turns into a claimed evaluation $e_y \stackrel{?}{=} M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})\boldsymbol{z}(\boldsymbol{\rho_y})$, where e_y is the final claim of the sumcheck, and $\boldsymbol{\rho_y}$ is the sumcheck challenge. Again, the verifier needs to check the evaluation of $M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$ and $\boldsymbol{z}(\boldsymbol{\rho_y})$. In order for the verifier to efficiently query $\boldsymbol{z}(\boldsymbol{\rho_y})$, we have the prover send an oracle to witness extension $\boldsymbol{w}$ at the start of the protocol, and the verifier can construct $\boldsymbol{z}(\boldsymbol{\rho_y})$ using $\boldsymbol{w}$.

To evaluate $M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$, the verifier needs a fast, succinct way to evaluate $\boldsymbol{A}, \boldsymbol{B}$, and $\boldsymbol{C}$ at $(\boldsymbol{\rho_x}, \boldsymbol{\rho_y})$. Spartan [73] uses offline memory-checking to express this verification as a circuit, which is then handled by an external proof system. However, as Gemini [9] points out, the timestamp calculations required for offline memory-checking need random memory access, making it unsuitable for distributed computing. Additionally, the external proof system itself must be delegation-friendly, leading to a circular dependency. We instead employ a lookup [48], directly proving the matrix evaluations.

Recall that for all $M \in \{A,B,C\}$, we have $\boldsymbol{M}(\boldsymbol{i},\boldsymbol{j}) = \sum_{\boldsymbol{x}} \boldsymbol{v}_M(\boldsymbol{x})\text{eq}(\hat{\boldsymbol{r}_M}(\boldsymbol{x}),\boldsymbol{i}),\text{eq}(\hat{\boldsymbol{c}_M}(\boldsymbol{x}),\boldsymbol{j})$. Recall that $\hat{\boldsymbol{r}_M}$ and $\hat{\boldsymbol{c}_M}$ output the vector of binary representations of the row and column indices of M, respectively. Then the evaluation claim $M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$ can be reduced to a sumcheck problem for $\sum_{\boldsymbol{x}} \boldsymbol{v}_M(\boldsymbol{x})\text{eq}(\hat{\boldsymbol{r}_M}(\boldsymbol{x}),\boldsymbol{\rho_x}),\text{eq}(\hat{\boldsymbol{c}_M}(\boldsymbol{x}),\boldsymbol{\rho_y})$, which results in evaluating $\boldsymbol{v}_M(r_z)$, $\text{eq}(\hat{\boldsymbol{r}_M}(r_z),\boldsymbol{\rho_x})$, and $\text{eq}(\hat{\boldsymbol{c}_M}(r_z),\boldsymbol{\rho_y})$ at a random point r_z. The verifier can easily check $\boldsymbol{v}_M$ using its oracle. However, the verifier still needs to check $\text{eq}(\hat{\boldsymbol{r}_M}(r_z),\boldsymbol{\rho_x})$ and $\text{eq}(\hat{\boldsymbol{c}_M}(r_z),\boldsymbol{\rho_y})$. The prover computes the multilinear polynomials such that $\text{eq}_{row}(\boldsymbol{x}) := \text{eq}(\hat{\boldsymbol{r}_M}(\boldsymbol{x}),\boldsymbol{\rho_x})$ and $\text{eq}_{col}(\boldsymbol{x}) := \text{eq}(\hat{\boldsymbol{c}_M}(\boldsymbol{x}),\boldsymbol{\rho_y})$ for $\boldsymbol{x} \in \{0,1\}^d$, and send the polynomial oracles to the verifier. And the verifier can simply query the oracles $\text{eq}_{row}(\boldsymbol{x})$ and $\text{eq}_{col}(\boldsymbol{x})$ at r_z.

Now, the prover needs to show that their oracle to $\text{eq}_{row}(x)$ and eq_{col} are well formed. We observe that if we treat $\text{eq}(\boldsymbol{i},\boldsymbol{\rho_x})$ as a table where $\boldsymbol{i}$ is the binary representation of the index, then $\text{eq}(\hat{\boldsymbol{r}_M}(\boldsymbol{x}),\boldsymbol{\rho_x})$ can be viewed as a subset of this table, where each element is indexed by the $\hat{\boldsymbol{r}_M}(\boldsymbol{x})$ values. This interpretation allows us to consider $\text{eq}_{row}(\boldsymbol{x})$ as a sequence of lookups into the table, with each lookup corresponding to the appropriate entry based on the $\boldsymbol{r}_M(\boldsymbol{x})$ indices.

We use the lookup protocol [48] to check the indexed lookup relations: $\{(\boldsymbol{r}_M(\boldsymbol{x}),\text{eq}_{row}(\boldsymbol{x}))\} \subset \{(i,\text{eq}(\boldsymbol{i},\boldsymbol{\rho_x}))\}$ where $i \in \mathbb{F}$, $\boldsymbol{x} \in \{0,1\}^d$ and $\boldsymbol{i} \in \{0,1\}^s$. The verifier can be convinced that the provided oracle to $\text{eq}_{row}(\boldsymbol{x})$ is valid, since this lookup relation ensures that each entry of $\text{eq}_{row}(\boldsymbol{x})$ is obtained from the table based on the index of $\boldsymbol{r}_M(\boldsymbol{x})$. Finally, the lookup protocol require the verifier to check the oracle evaluations of the lookup sequence and the table. Note that the lookup sequence oracle can be obtained from $\boldsymbol{r}_M(\boldsymbol{x})$ and $\text{eq}_{row}(\boldsymbol{x})$, which both are accessible by the verifier. And the table has a structure that allows the verifier to construct its oracle. The same process applies for $\text{eq}_{col}(x)$.

Finally, we instantiate all the polynomial oracles with multilinear polynomial commitments [65].

Formal Construction. We present our formal protocol in Appendix D.

Efficiency. The verifier needs to participate in a constant number of sumchecks over $\log(m)$ and $\log(n)$-variate polynomials and all evaluations can be done in $O(\log(m)+\log(n))$ time; thus, the verifier time and proof size are $O(\log(m)+\log(n))$, where m is the size of inputs and n is the number of non-zero entries in the matrix. The prover work, alongside the aforementioned sumchecks, additionally requires some polynomial operations. All of these take time linear in either the size of input or the number of non-zero entries. The tricky part is the calculation of eq_{row} and eq_{col}. The naive solution might takes $O(n\log m)$ time, as $\text{eq}(\boldsymbol{x},\boldsymbol{y}) := \prod_{i=1}^{\log m}(\boldsymbol{x_i}\boldsymbol{y_i} + (1-\boldsymbol{x_i})(1-\boldsymbol{y_i}))$ and each element could involve $O(\log m)$ multiplications. However, we observe that each element can be derived from the previous element's result. By leveraging dynamic programming, we can efficiently pre-compute and store all the possible values of $\text{eq}(\boldsymbol{x},\boldsymbol{y})$ in $O(m)$ time, and then construct eq_{row} and eq_{col} using the pre-compute results in $O(n)$ time. In summary, the overall prover time complexity is $O(m+n)$.

Zero-knowledge. So far, we have not discussed the zero-knowledge. However, achieving zero-knowledge is straightforward. We can easily incorporate zero-knowledge by adding a masking polynomial to the witness and sumcheck protocol [24, 85]. Additionally, we require the hiding properties of the PST13 [65] polynomial commitment scheme.

Security proof. The security of the non-holographic part follows Spartan's approach. In particular, the proof of Eqs. (1) and (2) follows the Theorem 5.1 in Spartan [73]. For the holographic part, we utilize a lookup algorithm [48] to verify the correctness of the oracle for $\text{eq}_{row}(\boldsymbol{x})$. The key aspect here is that the lookup relation ensures each entry of $\text{eq}_{row}(\boldsymbol{x})$ is accurately retrieved from the structured table based on the index of $\boldsymbol{r}(\boldsymbol{x})$, which fulfills the requirement of the sparse matrix encoding. The table's structure allows the verifier to construct its oracle independently, ensuring that the prover

cannot cheat by manipulating the oracle responses. By requiring soundness from the lookup protocol, we guarantee that any incorrect entries would be detected, thereby ensuring the soundness of the holographic verification.

Public Delegation of DFS. DFS enables efficient public delegation through distributed computing. The protocols primarily leverage the tree-like structure of the multilinear polynomials, so that each node can recursively compute the values in the tree and then the coordinator combines the results to obtain the final output. We present our formal protocol in Appendix E. As a result, each node's computation is $O(N_w + \frac{m+n}{N_w})$, where m is the size of inputs, n is the number of non-zero entries in the matrix, and N_w is the number of nodes, and the communication cost is $O(\log m + \log n)$.

5 Private delegation of DFS

In the private-delegation setting, a delegator $\mathcal{D}$ holds a witness, and $\mathcal{D}$ as well as all parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ know the common reference string (CRS) and preprocessing keys included in a zkSNARK scheme. We denote every party as $\mathcal{P}_j$ for $j \in [1 \ldots N_p]$, which controls N_w nodes, each denoted by $\mathcal{E}_j^{(k)}$ for $k \in [1 \ldots N_w]$. All computations of $\mathcal{P}_j$ could be performed in parallel by these nodes. In this setting, $\mathcal{D}$ is always honest, and at most one party is corrupted for both the two-party and three-party settings. We prove the security of our protocols in the presence of malicious, static adversaries. We use the standard security model in the ideal/real paradigm [22, 41]. In our protocols, for a vector of secret sharings $[\![\boldsymbol{x}]\!]$, each set of shares $[\![\boldsymbol{x}]\!]_j$ held by every party $\mathcal{P}_j$ is split into N_w parts $\{[\![\boldsymbol{x}]\!]_j^{(k)}\}_{k\in[1\ldots N_w]}$, and each node $\mathcal{E}_j^{(k)}$ holds $[\![\boldsymbol{x}]\!]_j^{(k)}$.

5.1 Building blocks

The private-delegation protocol can be built using the following building blocks. For simplicity, we omit the number N_w of nodes controlled by every party.

- **Multi-scalar multiplication:** $\mathsf{MSM}([\![\boldsymbol{y}]\!], \boldsymbol{X}) \to [\![Z]\!]$ takes as input a vector of secret sharings $[\![\boldsymbol{y}]\!]$ with $\boldsymbol{y} \in \mathbb{F}^\ell$, a vector of public group elements $\boldsymbol{X} \in \mathbb{G}^\ell$, and outputs a secret sharing $[\![Z]\!]$ with $Z = \sum_{i\in[1\ldots\ell]} \boldsymbol{y}_i \cdot \boldsymbol{X}_i \in \mathbb{G}$, where "$\cdot$" denotes the scalar multiplication in group $\mathbb{G}$.
- **Linear combination:** $\mathsf{LinearComb}([\![\boldsymbol{x}]\!], \boldsymbol{c}) \to [\![y]\!]$ takes as input a vector of secret sharings $[\![\boldsymbol{x}]\!]$ with $\boldsymbol{x} \in \mathbb{F}^\ell$ and a vector of public elements $\boldsymbol{c} \in \mathbb{F}^{\ell+1}$, and outputs $[\![y]\!]$ with $y = \sum_{i\in[1,\ell]} \boldsymbol{c}_i \cdot \boldsymbol{x}_i + \boldsymbol{c}_0 \in \mathbb{F}$.
- **Inner product:** $\mathsf{InnerProd}([\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]) \to \langle z \rangle$ takes as input two vectors of secret sharings $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$ with $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^\ell$, and outputs a secret sharing $\langle z \rangle$ with $z = \sum_{i\in[1\ldots\ell]} \boldsymbol{x}_i \cdot \boldsymbol{y}_i \in \mathbb{F}$.
- **Folding:** $\mathsf{Fold}([\![\boldsymbol{x}]\!], r) \to [\![\boldsymbol{y}]\!]$ takes as input a vector of secret sharings $[\![\boldsymbol{x}]\!]$ with $\boldsymbol{x} \in \mathbb{F}^\ell$ and a public element $r \in \mathbb{F}$, and outputs a vector of secret sharings $[\![\boldsymbol{y}]\!]$, where $\boldsymbol{y} \in \mathbb{F}^{\ell/2}$ such that $\boldsymbol{y}_i = \boldsymbol{x}_{2i-1} + r \cdot \boldsymbol{x}_{2i} \in \mathbb{F}$ for $i \in [1 \ldots \ell/2]$. Here, we w.l.o.g. assume that ℓ is an even.

In addition, our private-delegation protocol will invoke two building blocks Share and Rec defined in Section 3.2. In particular, a delegator $\mathcal{D}$ could run the Share procedure to share the witness with all parties. The parties run the Rec procedure to let $\mathcal{D}$ obtain a proof. These building blocks may be independent of interest to design private-delegation protocols for other zkSNARK schemes such as Marlin [24].

A private-delegation protocol can invoke the building blocks multiple times in any order. We use an ideal functionality (shown in Appendix J) to define security of the protocols that only consist of these building blocks. Building upon this, we are able to prove the security of our private-delegation protocol with these building blocks.

Next, we show how to instantiate the building blocks using replicated secret sharing (RSS). The AddSS-based instantiation is postponed to Appendix G.

Instantiation from RSS. The following protocol shows how to construct the building blocks in the honest-majority setting from replicated secret sharings. The constructions of building blocks Share and Rec are described in Appendix F.1. We focus on the three-party setting with at most one corrupted party, but the protocol is natural to be extended to more parties. Note that the output by the $\mathsf{InnerProd}$ algorithm need to be randomized with zero sharings. When multiple zero sharings need to be generated, $\mathcal{D}$ can send a set of keys to each party in the setup phase, and then all parties generate random zero sharings with free communication using the keys and a pseudo-random function (PRF). The detailed protocol to generate zero sharings can be found in prior works [57, 55, 49]. The security proof can be found in Appendix J.

Prot. 1: RSS-based Building Blocks

Let $m = \lceil \ell / N_w \rceil$.

- $\mathsf{MSM}([\![\boldsymbol{y}]\!], \boldsymbol{X}) \to [\![Z]\!]$: Given a vector of RSSs $[\![\boldsymbol{y}]\!]$ and public group elements $\boldsymbol{X} \in \mathbb{F}^\ell$, every party $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel: $[\![Z]\!]_j^{(k)} := \sum_{i=(k-1)m+1}^{km} [\![\boldsymbol{y}_i]\!]_j^{(k)} \cdot X_i$. Every party $\mathcal{P}_j$ chooses one node to compute $[\![Z]\!]_j := \sum_{k=1}^{N_w} [\![Z]\!]_j^{(k)}$, and then all parties output a RSS $[\![Z]\!]$.
- $\mathsf{LinearComb}([\![\boldsymbol{x}]\!], \boldsymbol{c}) \to [\![y]\!]$: Given a vector of RSSs $[\![\boldsymbol{x}]\!]$ and public elements $\boldsymbol{c} \in \mathbb{F}^{\ell+1}$, every party $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel: $[\![y]\!]_j^{(k)} := \sum_{i=(k-1)m+1}^{km} \boldsymbol{c}_i \cdot [\![\boldsymbol{x}_i]\!]_j^{(k)}$. Every party $\mathcal{P}_j$ chooses one node to compute $[\![y]\!]_j := \sum_{k=1}^{N_w} [\![y]\!]_j^{(k)} + [\![\boldsymbol{c}_0]\!]_j$, where $[\![\boldsymbol{c}_0]\!]$ is locally computed from the public element $\boldsymbol{c}_0$. Then, all parties output $[\![y]\!]$.
- $\mathsf{InnerProd}([\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]) \to \langle z \rangle$: Given two vectors of RSSs $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$, $\mathcal{D}$ runs the $\mathsf{Share}(0)$ algorithm to let the parties obtain a fresh zero-sharing $\langle 0 \rangle$. Every party $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel: $\langle z \rangle_j^{(k)} :=$

$\sum_{i=(k-1)m+1}^{km} [\![\boldsymbol{x}_i]\!]_j^{(k)} \cdot [\![\boldsymbol{y}_i]\!]_j^{(k)}$. Every party $\mathcal{P}_j$ chooses one node to compute $\langle z \rangle_j := \sum_{k=1}^{N_w} \langle z \rangle_j^{(k)} + \langle 0 \rangle_j$, and then all parties output $\langle z \rangle$.

- $\mathsf{Fold}([\![\boldsymbol{x}]\!], r) \to [\![\boldsymbol{y}]\!]$: Given a vector of RSSs $[\![\boldsymbol{x}]\!]$ and a public element $r \in \mathbb{F}$, let $m' = \lceil \ell / 2N_w \rceil$, every party $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel $[\![\boldsymbol{y}_i]\!]_j^{(k)} := \sum_{i=(k-1)m'+1}^{km'} [\![\boldsymbol{x}_{2i-1}]\!]_j^{(k)} + r \cdot [\![\boldsymbol{x}_{2i}]\!]_j^{(k)}$ for $i \in [1 \dots \ell/2]$. Then, all parties output $[\![\boldsymbol{y}]\!]$ by collecting results from all the nodes.

5.2 Our private-delegation protocol

Delegator Work. In private delegation, the proof delegator acts as the global coordinator, responsible for collecting messages from each party and outputting the final proof. Additionally, the delegator generates the Fiat-Shamir and the zero-knowledge randomnesses. Finally, the delegator also needs to verify proof to ensure that no malicious attacks have compromised the integrity of the computation. Since both the output proof and verification are succinct, the workload and communication for the delegator are logarithmic.

Private delegations for sumcheck. In DFS, all linear operations can be completed locally by each party without additional MPC communication. The only exception is the sumcheck protocol, where the computation of sumcheck messages involves inner-product operations that may require MPC communication. Specifically, DFS includes two types of sumchecks involving secret shares: $\sum f \cdot g \cdot h$, where f and g are private polynomials and h is a public polynomial, and $\sum f \cdot g$, where f is a private polynomial and g is a public polynomial. For the second type, since g is public, the multiplication can be handled via linear combination $\mathsf{linearcomb}$, avoiding the need for inner-product operations on secret values. However, the first type involves the product of two private polynomials, which requires MPC communication to securely compute the inner product. This makes the first type of sumcheck more communication-intensive compared to the second.

Specifically, in the i-th round of sumcheck, the prover needs to compute a degree-4 polynomial $p(X_i)$, defined as $p(X_i) := \sum_{\boldsymbol{b} \in \{0,1\}^{n-i}} (f \cdot g \cdot h)(\boldsymbol{r}, X_i, \boldsymbol{b})$, where $\boldsymbol{r} \in \{0,1\}^{i-1}$ are the verifier randomnesses in the previous rounds, and send it to the verifier. After receiving $p(X_i)$, the verifier samples a new randomness r_i and sends it back to the prover, who then proceeds with the next round of computation. It's important to note that the inner product only occurs during the computation of $p(X_i)$ in each round, and $p(X_i)$ is directly used as part of the proof. So the computation of $p(X_{i+1})$ relies only on f, g, h, and the public randomness r_i, and there is only a single layer of multiplication gates. As we designed in the honest-majority scenario, the inner product will not trigger additional MPC communication, allowing each party to perform the computation locally without needing inter-party communication. However, in the dishonest-majority scenario, each party needs to perform multiplications using Beaver triples. This means that inter-party communication is inevitable and will involve linear communication overhead. Specifically, during the multiplication operations in the sumcheck protocol, each party must engage in MPC protocols to securely compute the inner products using the Beaver triples.

Finally, we also need to use distributed computing to accelerate each party's local computation. This part is similar to the distributed computing protocol used in public delegation. Each party will first distribute the shares of f and g, along with the public polynomial h, across its local computation nodes. For simplicity, we assume the number of nodes is a power of 2, allowing us to partition the computations by variates so that each node receives a multilinear polynomial with fewer variables. In the sumcheck protocol, after determining each variable r_i, the process effectively performs a folding operation on the polynomial. This folding can be done in parallel across nodes, with each node folding its local polynomial. Similarly, polynomial evaluation can also be executed locally at each node, with the results aggregated at the end.

Notably, all operations in this process—both folding and evaluation—are linear and do not require any MPC communication. Thus, the entire procedure can be completed efficiently through distributed computing, ensuring scalability and speed without incurring additional communication overheads.

Mode Switching. As mentioned in Section 2, The final phase of DFS involves the holography verification, which only contains computations on purely public data. At this stage, the protocol can seamlessly switch from private delegation to public delegation. Specifically, the delegator can distribute the public data, such as the R1CS matrices, across all the nodes managed by the different parties before the proof generation. This phase can directly adopt the distributed computing protocol used for public delegation (Appendix E), allowing for the maximization of resource utilization, further enhancing scalability and efficiency. As shown in Section 6.1.1, even with a single node per party, DFS outperforms prior work EOS because of mode switching.

Formal Construction. We present our formal protocols in Appendix H.

Security. The security of our private-delegation protocol is straightforward. All SNARK operations except for inner product are linear, and thus the security is directly guaranteed. Only the inner-product operation allows a malicious adversary to introduce an error at the inner-product result. However, the result is reconstructed to the delegator $\mathcal{D}$, and thus is not used in other operations. Therefore, the error does not reveal the privacy, and will be detected by $\mathcal{D}$ through verifying the final proof. In addition, the adversary can also introduce some errors into the reconstruction secrets. In a similar reason, the errors have no impact on privacy, and will also be detected by $\mathcal{D}$. Overall, the malicious adversary can only perform additive attacks, which will be detected by $\mathcal{D}$. Due to the

single layer of non-linear operations, the privacy of witness is guaranteed. The final proof is verified by $\mathcal{D}$, and thus is valid if $\mathcal{D}$ does not abort. Note that our protocol also involves multi-round of Fiat-Shamir transformations. This is secure because each subcircuit has only one layer of multiplication, and the output is hashed for FS. Therefore, any additive attack would lead to different FS values and, thus, invalid proofs with overwhelming probability. Formal proof is in Appendix J.

Efficiency. Consider an R1CS with m constraints and n non-zero entries, with three parties in the honest-majority setting and two parties in the dishonest-majority setting, where each party has N_w computation nodes. In the honest-majority scenario, there is no additional MPC communication, and each party only needs to distribute and process the computations related to its share. Note that each party may internally coordinate its own nodes. Thus, the computation cost for each node is $O(N_w + \frac{(m+n)}{N_w})$. Communication is minimal due to DFS requiring $O(\log m + \log n)$ rounds, with each round only needing to send a constant-sized message. Therefore, the communication cost per node is $O(\log m + \log n)$. In the dishonest-majority scenario, the computation cost for each node remains $O(N_w + \frac{(m+n)}{N_w})$, but multiplications incur linear communications, meaning the communication cost for each node is also $O(\frac{(m+n)}{N_w})$. In both scenarios, the delegator, responsible for coordinating the parties and verifying the proof, only performs a constant amount of work per round, so the total computation and communication cost is $O(\log m + \log n)$. Finally, the proof size and verification cost remain the same.

6 Implementation and evaluation

We implemented DFS as a library in about 13000 lines of Rust code, building on top of the `arkworks` framework [27]. Our library contains not only a single-machine implementation of DFS's prover, but also public and private delegation protocols. For the zkSNARK, we implemented the LogUp and multilinear sumcheck PIOPs, and PST13 polynomial commitments. For private delegation, our implementation supports both AddSS and RSS protocols. To provide inter-node communication, we relied on `mpi-rs`, a MPI library for Rust. The artifact can be found in DOI 10.5281/zenodo.14677896. We also plan to open-source the library in Arkworks [27].

6.1 Evaluation and comparison

Our goal is to determine whether DFS achieves horizontal scalability, and thus answer the following questions:

- Does the proof generation latency decrease *linearly* as we increase the number of nodes?
- Does the proof generation latency increase *linearly* as we increase the size of the instance?
- Does communication become a bottleneck when we *increase* the number of nodes and the instance size?

Experimental setup. We evaluated our system on the university's computing cluster, with each machine equipped with an Intel Xeon Scalable Cascade Lake 6248 CPU and 192GB of memory. In the delegation scenarios, each node uses 8 vCPUs. All network link has maximum bandwidth of 100Gbps. The nodes communicated directly with the coordinator, with logarithmic communication costs. In all scenarios, the coordinator cost is also logarithmic. In public delegation, the coordinator could be one of the nodes; in private delegation, the coordinator could be the delegator. We ran the delegator on a cluster node, but limited its vCPUs and memory to simulate real-world deployments. Concretely, the delegator was limited to 1MB of memory during the delegation We obtained experimental setups and data of prior work from their papers. Moreover, DFS's setup used fewer resources than prior baselines, and so our comparisons are conservative and favor the baselines. For example, EOS [25]'s experiments used two EC2 c5.24xlarge servers (96 vCPUs at 3.6GHz); the equivalent DFS experiments used two servers with 8 vCPUs at 2.5GHz. Both used BLS12-381 and Arkworks. Pianist [58]'s experiments used m6i.16xlarge instances (64 vCPUs at 3.5GHz); the equivalent DFS experiments had to use more machines (8 vCPUs at 2.5GHz) to reach the same total number of vCPUs. Pianist used a faster curve (BN254) and MSM (Gnark).

We assume three parties for RSS, and two parties for AddSS. We do not enforce authenticated secret share for AddSS-based protocol to enable a fair comparison with prior semi-honest work. In contrast, our RSS-based protocol achieves malicious security for free. We assume that each party receives their share of the instance, witness, and computation trace before the proof generation begins. These shares are then distributed to their respective nodes. Since this distribution can be preprocessed, we do not include this time in the overall proof generation time. In all scenarios, the total per-party input is less than 100 GB. We gives a detailed brakedown in Appendix A.

Proof size, verifier time and coordinator cost. The proof size and verifier time do not depend on the particular implementation of the prover algorithm. For instance sizes ranging from 2^{15} to 2^{27}, the proof size is about 10 - 17 KB and the verification time is about 37 - 45 ms. Both are logarithmic to the instance size. The number of interaction rounds is also logarithmic. In all scenarios, the coordinator's local computation time is less than 50 ms, communication with each node is less than 30 KB, and memory usage is less than 1 MB making our protocol viable for weak devices.

Single machine. We present the single machine proving time in Appendix B. The results show that DFS's prover achieves lower latency than Marlin [24], previously noted for its efficiency in private delegation [64].

Public delegation. We present the full evaluation results in Appendix C. Compared to the state-of-the-art public delegation protocol Pianist [58], both take approximately 20 seconds

for 2^{25} constraints with 512 vCPUs. The communication cost of DFS is about 21 KB, which is larger than Pianist's 2.1 KB. However, it is still very small and acceptable in real-world applications. Additionally, DFS provides better support for private delegation, offering enhanced privacy and efficiency in scenarios where privacy is crucial. Pianist achieves a constant proof size (2.8 KB) and verifier time (3.5 ms) with quasilinear proving time, whereas we achieve logarithmic proof size (10 KB) and verifier time (40 ms) with linear proving time.

6.1.1 Private delegation

Replicated secret sharing. In Fig. 3 and Fig. 4, we report the proving time of our private delegation protocol based on RSS. We observe that as expected, the proof time decreases linearly with the increasing number of nodes per party. Moreover, even when each party has only one node, the time required is still less than that of a single machine. This is a significant advantage of DFS, because the holography part, as the bottleneck, does not require MPC and therefore can utilize all resources for acceleration, achieving better performance.

There are three types of communication: node-node within a party, inter-party, and node-coordinator. Our private delegation protocols avoid intra-party communication; RSS-based protocols further eliminate inter-party communication. Fig. 5 shows the per-node cost for communication. We see that the communication cost grows logarithmically with the number of constraints, as expected. Moreover, since DFS achieves zero communication cost for MPC, the communication here is only related to public operations.

Additive secret sharing. In Fig. 6 and Fig. 7, we report the proving time of DFS based on AddSS. We observe that the performance is close to that of RSS. That is because of several reasons. First, our default bandwidth is 100 Gbps, which is much larger than the communication cost of AddSS. If the bandwidth is 5 Mbps, AddSS may require about 100 extra seconds for 2^{24} constraints. Second, the holographic protocol in both AddSS and RSS is accelerated without the use of MPC. Due to implementation limitations, the holographic protocol can only be distributed among 2^n nodes, even though RSS has three parties and thus more vCPUs. This issue could be addressed in future implementations. Finally, we do not implement authenticated share for AddSS to enforce comparison with other semi-honest work [25]. Fig. 8 reports the per-node communication cost. Unlike RSS, the communication cost scales *linearly* with the number of constraints. Note that this is dominated by inter-node communications for multiplication, and the coordinator communication is still logarithmic and no more than 30 KB. Moreover, each node interacts exclusively with another node, enabling the entire communication to be efficiently partitioned.

Finally, in both the replicated and additive cases, the delegation protocol spends about 55% time in the holographic protocol, compare to 70% in public delegation. This is because the non-holographic part in the private delegation uses MPC and less resources for distributed computation.

Comparison to prior work. Compared to previous private delegation protocols, our approach differs significantly in both the threat model and system model, and we give a more detailed comparison in Section 7. Despite these differences, the distributed private delegation scheme zkSAAS [39] requires about 4000 seconds and 350 GB of communication for 2^{24} constraints with 280 vCPUs. In contrast, DFS using RSS and AddSS spend approximately 50 seconds with only 192 and 128 vCPUs. Specifically, RSS-based DFS requires only logarithmic communication, with each machine needing no more than 30 KB and a total communication overhead of less than 500 KB. While AddSS-based DFS, like zkSAAS, involves linear communication, each node requires no more than 1.3 GB, with a total communication overhead of less than 20 GB. A recent prior work [59] has achieved improvements over zkSAAS. However, this approach is limited to data-parallel circuits and still requires linear communication, which concretely can be up to 100× worse than our method.

Another line of work [25, 64] shares a similar system model with ours but does not consider distributed computing, making them not scalable. In comparison to EOS [25], which uses AddSS with each party operating a single node, our protocol demonstrates improvement under the same setup. For 2^{24} constraints, EOS requires approximately 500 seconds, whereas DFS only needs 400 seconds while using much fewer vCPUs per machine. This improvement is due to DFS being faster than Marlin and the mode switch of our delegation protocol in the holographic phase as described in Section 5.2.

7 Related works

Existing zkSNARKs. Existing zkSNARKs vary widely in terms of their design goals, computational requirements, and application areas. For example, [45, 24, 38] are designed for succinct verification; [73, 44, 86, 23] are designed for linear proving; [5, 26, 3] are designed for post-quantum security. However, none of them are designed for private delegation, making them less friendly for distributed computing or MPC. For instance, Marlin's [24] reliance on FFTs and Spartan's [73] offline-memory checking are not efficient in distributed environments. Plonk's [38, 23] partial product computations and Orion's [86] Merkle-tree hashing are inefficient for MPC. These limitations underscore the need for new protocols that better support both distributed computing and MPC. In contrast, DFS achieves linear proving time, logarithmic verification, and efficient private delegation.

Public delegation. Public delegation protocols have been studied to enhance the efficiency and scalability of proof generation in distributed systems without hiding the witness. DIZK [83] is proposed to support distributed computation for Groth16 [45]. Similarly, deVirgo accelerates the GKR-

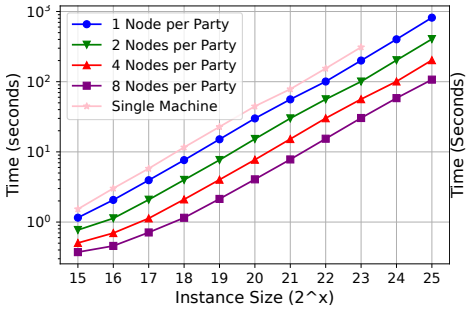

Figure 3: Latency with RSS.

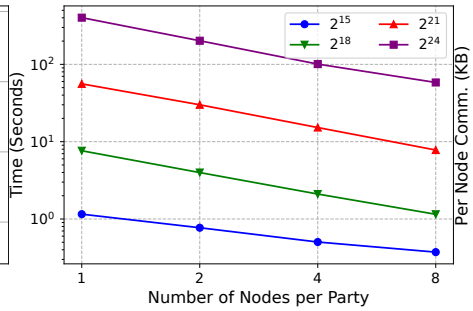

Figure 4: Scalability with RSS.

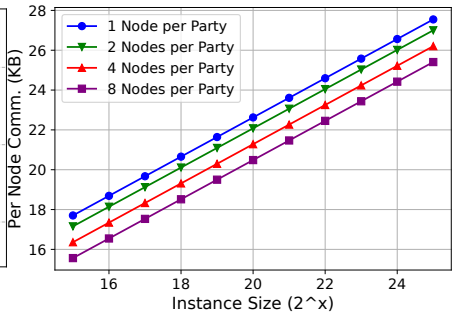

Figure 5: Comm. with RSS.

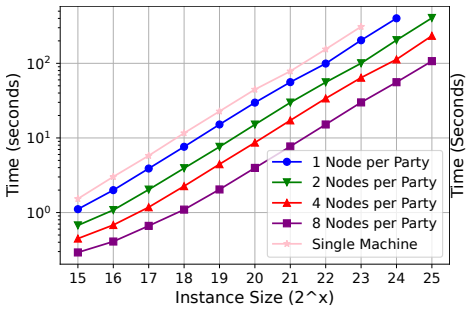

Figure 6: Latency with AddSS.

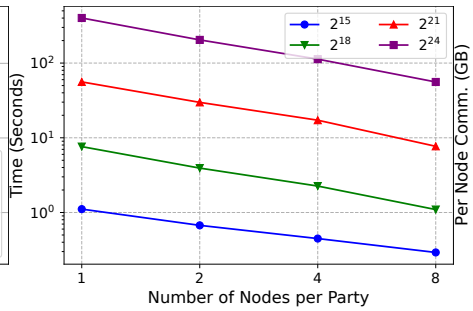

Figure 7: Scalability with AddSS.

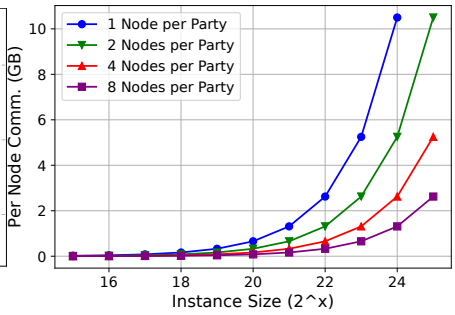

Figure 8: Comm. with AddSS.

based [42] protocol Virgo [91], showing improvements over DIZK. However, Virgo's succinct verification is limited to structured layer-circuits, and deVirgo requires the circuits to be data-parallel for effective parallelization. Both DIZK and deVirgo incur linear communication costs. Pianist [58] solves the issue and proposes a constant-communication solution for arithmetic circuits. However, Pianist's prover time is quasilinear, while DFS is linear. A recent concurrent work HyperPianist [56] achieves the same complexity as public delegated DFS but targets arithmetic circuits instead of R1CS. Another work, Gemini [9], is proposed to achieve a space-efficient prover, and has the potential to be adapted for efficient distributed computation. However, these work [58, 56, 9] require the witness to compute partial products, which introduces significant overheads in MPC. Another line of work uses folding-based SNARKs [63, 18, 54] to reduce resources overheads. However, they rely on recursive proof composition, incurring additional overheads.

Private delegation. Private delegation protocols have been proposed to support efficient private proof generation in distributed systems. Collaborative zkSNARKs [64] tested various MPC schemes on Groth16, Marlin, and Plonk, finding that Marlin performed the best. Another work [32] applies MPC to Ligero. These works were the first to introduce the concept of MPC-friendliness as a criteria for evaluating *existing* zkSNARKs. Building on this insight, EOS [25] explores further optimizations for Marlin's performance using AddSS. As discussed in Section 2.4, EOS fails to achieve malicious security without authenticated shares. Moreover, these works require linear communication and do not support parallel computing. We show that under the same threat model and MPC scheme, DFS outperforms EOS in Section 6.1.1. zkSAAS [39] aims to enhance performance and scalability while preserving privacy by using an honest-majority threat model and leveraging packed secret sharing (PSS) to accelerate Plonk. In zkSAAS, each party operates a single node for parallel computation, supporting a large number of parties. However, Plonk's product check is not friendly for MPC, resulting in linear communication overhead. Moreover, zkSAAS only achieves semi-honest security and, due to its star network topology, is limited by the resources of the central node. Despite the different threat and system model, our experiments show that DFS performs significantly better.

Similar to zkSAAS's setting, a recent work [59] scales a GKR-based zkSNARK, Libra [85]. However, they inherit weaknesses from GKR [42] – Libra works with layered arithmetic circuits. Their proof size and verifier time is linear in the size of the repeated circuit, which may be large for certain computations. In addition, this work only scales for data-parallel circuits, while we achieve high parallelism for general arithmetic circuits. That is because, depending on the circuit structure, GKR may require PSS to handle operations between secrets within the same share. Moreover, [59] still require linear communication, while DFS's cost is logarithmic. We also note that GKR protocol might not be distribution-friendly for non-parallel circuit since the circuit structure affects the memory access patterns.

8 Conclusion

We propose a new delegation-friendly zkSNARK DFS, and provide both public delegation and private delegation for it. The experiments show that DFS achieves logarithmic communication and can scale with large general circuits.

9 Ethics considerations and compliance with the open science policy

9.1 Ethics considerations

We attest that we have thoroughly reviewed the ethics considerations as outlined in the conference call for papers, the detailed submission instructions, and the ethics guidelines document provided by the conference organizers. The research team has carefully evaluated the ethical implications of our work on DFS, ensuring that the research has been conducted in accordance with the highest ethical standards.

Our team has considered all potential ethical issues arising from this research, including the responsible disclosure of findings, the privacy implications of the technologies developed, and the potential for both positive and negative impacts on stakeholders. We have also proactively assessed the possible risks and mitigated them where necessary. We believe that our research was conducted ethically and in a manner that aligns with both the principles of beneficence and respect for persons as described in the Menlo Report.

Additionally, our next steps following publication have been carefully planned with ethical considerations in mind. We commit to following responsible procedures for the further dissemination and application of our findings, particularly in terms of sharing data and code in compliance with the conference's open science policy. We are prepared to engage with the broader community to address any ethical concerns that may arise as the research progresses.

Finally, we have also provided this additional Ethics Considerations and Compliance with the Open Science Policy section to ensure that all relevant ethical issues are transparent and addressed appropriately.

9.2 Compliance with the open science policy

In alignment with the Open Science Policy, we commit to making all research artifacts related to DFS openly accessible to the community. This includes the source code and detailed experimental results used in our study. Our goal is to promote transparency and reproducibility in the field of cryptographic research, enabling others to validate, build upon, and further enhance our work.

We will provide all relevant resources in a public repository, ensuring that proper documentation and instructions are included for easy replication and understanding. Additionally, we will release the DFS implementation under an appropriate open-source license, allowing others to freely use, modify, and distribute the work while maintaining responsible practices. By adhering to these principles, we not only comply with the Open Science Policy but also contribute to the broader academic and research community in a manner that fosters collaboration and innovation.

The artifact can be found in DOI 10.5281/zenodo.14677896.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their invaluable feedback.

Yu Yu is supported by the National Natural Science Foundation of China (Grant Nos. 92270201 and 62125204). Yu Yu's work has also been supported by the New Corner-stone Science Foundation through the XPLORER PRIZE. Work of Kang Yang is supported by the National Natural Science Foundation of China (Grant No. 62102037). Work of Xiao Wang is supported by NSF awards #2236819 and #2318975. Yuncong Hu is supported by the National Natural Science Foundation of China, the Science and Technology Commission of Shanghai Municipality, the Shanghai Science and Technology Program (Grant Nos. 24BC3200200), the ExploreX project of SJTU, and gifts/awards from BIANJIE.AI, Polyhedra, and Xiaomi.

References

[1] D. Beaver. "Efficient Multiparty Protocols Using Circuit Randomization". In: CRYPTO '91.

[2] A. Ben-David, N. Nisan, and B. Pinkas. "FairplayMP: a system for secure multi-party computation". In: CCS '08.

[3] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: ICALP '18.

[4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: S&P '14.

[5] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. "Aurora: Transparent Succinct Arguments for R1CS". In: EUROCRYPT '19.

[6] D. J. Bernstein. "Pippenger's Exponentiation Algorithm". http://cr.yp.to/papers/pippenger.pdf.

[7] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. "Checking the correctness of memories". In: FOCS '91.

[8] M. Bombar, G. Couteau, A. Couvreur, and C. Ducros. "Correlated Pseudorandomness from the Hardness of Quasi-Abelian Decoding". In: ed. by H. Handschuh and A. Lysyanskaya. Springer Nature Switzerland.

[9] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. "Gemini: Elastic SNARKs for Diverse Environments". In: EUROCRYPT '22.

[10] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. “ZEXE: Enabling Decentralized Private Computation”. In: S&P ’20.

[11] E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai. “Compressing Vector OLE”. In: CCS ’18.

[12] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl. “Correlated Pseudorandomness from Expand-Accumulate Codes”. In: CRYPTO ’22.

[13] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl. “Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation”. In: CCS ’19.

[14] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Correlated Pseudorandom Functions from Variable-Density LPN”. In: FOCS ’20.

[15] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Efficient Pseudorandom Correlation Generators from Ring-LPN”. In: CRYPTO ’20.

[16] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Efficient Pseudorandom Correlation Generators: Silent OT Extension and More”. In: CRYPTO ’19.

[17] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. “Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs”. In: ASIACRYPT ’20.

[18] B. Bünz and B. Chen. “Protostar: Generic Efficient Accumulation/Folding for Special-Sound Protocols”. In: ASIACRYPT’2023.

[19] B. Bünz, B. Fisch, and A. Szepieniec. “Transparent SNARKs from DARK Compilers”. In: EUROCRYPT ’20.

[20] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. “The Wonderful World of Global Random Oracles”. In: EUROCRYPT ’18.

[21] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodriguez. “Lunar: a Toolbox for More Efficient Universal and Updatable zkSNARKs and Commit-and-Prove Extensions”. In: ASIACRYPT ’21.

[22] R. Canetti. “Security and Composition of Multiparty Cryptographic Protocols”. In: *Journal of Cryptology* (2000).

[23] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. “HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates”. In: EUROCRYPT ’23.

[24] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: EUROCRYPT ’20.

[25] A. Chiesa, R. Lehmkuhl, P. Mishra, and Y. Zhang. “Eos: Efficient Private Delegation of zkSNARK Provers”. In: USENIX Security ’23.

[26] A. Chiesa, D. Ojha, and N. Spooner. “Fractal: Post-Quantum and Transparent Recursive Proofs from Holography”. In: EUROCRYPT ’20.

[27] arkworks contributors. *arkworks zkSNARK ecosystem*. 2022. URL: https://arkworks.rs.

[28] R. Cramer, I. Damgård, and Y. Ishai. “Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation”. In: TCC ’05.

[29] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. “Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits”. In: ESORICS ’13.

[30] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: CRYPTO ’12.

[31] A. Damiano and P. Scholl. “Low-Communication Multiparty Triple Generation for SPDZ from Ring-LPN”. In: PKC ’22.

[32] P. Dayama, A. Patra, P. Paul, N. Singh, and D. Vinayagamurthy. “How to prove any NP statement jointly? Efficient Distributed-prover Zero-Knowledge Protocols”. In: PETS ’22.

[33] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. “Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation”. In: S&P ’16.

[34] L. Eagen, D. Fiore, and A. Gabizon. “cq: Cached quotients for fast lookups”. In: *IACR Cryptol. ePrint Arch.* (2022).

[35] M. K. Franklin and M. Yung. “Communication Complexity of Secure Computation (Extended Abstract)”. In: ACM Press.

[36] A. Gabizon and D. Khovratovich. “flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size”. In: *IACR Cryptol. ePrint Arch.* (2022).

[37] A. Gabizon and Z. J. Williamson. “plookup: A simplified polynomial protocol for lookup tables”. In: *IACR Cryptol. ePrint Arch.* (2020).

[38] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. “PLONK: Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge”. IACR ePrint Report 2019/953.

[39] S. Garg, A. Goel, A. Jain, G.-V. Policharla, and S. Sekar. “zkSaaS: Zero-Knowledge SNARKs as a Service”. In: USENIX Security ’23.

[40] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. “Circuits resilient to additive attacks with applications to secure computation”. In: STOC ’14.

[41] O. Goldreich. *Foundations of Cryptography: Basic Applications*. Vol. 2. Cambridge, UK: Cambridge University Press, 2004.

[42] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. “Delegating Computation: Interactive Proofs for Muggles”. In: STOC ’08.

[43] S. Goldwasser and Y. Lindell. “Secure Multi-Party Computation without Agreement”. In: *Journal of Cryptology* (2005).

[44] A. Golovnev, J. Lee, S. T. V. Setty, J. Thaler, and R. S. Wahby. “Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS”. In: CRYPTO ’23.

[45] J. Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: EUROCRYPT ’16.

[46] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish. “Zero-Knowledge Middleboxes”. In: USENIX Security ’22.

[47] X. Guo, K. Yang, X. Wang, W. Zhang, X. Xie, J. Zhang, and Z. Liu. “Half-Tree: Halving the Cost of Tree Expansion in COT and DPF”. In: EUROCRYPT ’23.

[48] U. Haböck. “Multivariate lookups based on logarithmic derivatives”. In: *IACR Cryptol. ePrint Arch.* (2022).

[49] C. Hazay, E. Orsini, P. Scholl, and E. Soria-Vazquez. “TinyKeys: A New Approach to Efficient Multi-Party Computation”. In: (2022).

[50] C. Hazay, P. Scholl, and E. Soria-Vazquez. “Low Cost Constant Round MPC Combining BMR and Oblivious Transfer”. In: vol. 10624. LNCS. Springer.

[51] A. Kate, G. M. Zaverucha, and I. Goldberg. “Constant-Size Commitments to Polynomials and Their Applications”. In: ASIACRYPT ’10.

[52] M. Keller, E. Orsini, and P. Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: CCS ’16.

[53] M. Keller, V. Pastro, and D. Rotaru. “Overdrive: Making SPDZ Great Again”. In: EUROCRYPT ’18.

[54] A. Kothapalli, S. T. V. Setty, and I. Tzialla. “Nova: Recursive Zero-Knowledge Arguments from Folding Schemes”. In:

[55] N. Koti, S. Patil, A. Patra, and A. Suresh. “MPClan: Protocol Suite for Privacy-Conscious Computations”. In: *Journal of Cryptology* (2023).

[56] C. Li, Y. Li, P. Zhu, W. Qu, and J. Zhang. “HyperPianist: Pianist with Linear-Time Prover via Fully Distributed HyperPlonk”. In: *Cryptology ePrint Archive* (2024).

[57] Y. Lindell and A. Nof. “A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority”. In: CCS ’17.

[58] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang. “Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs”. In: S&P’24.

[59] X. Liu, Z. Zhou, Y. Wang, B. Zhang, and X. Yang. “Scalable Collaborative zk-SNARK: Fully Distributed Proof Generation and Malicious Security”. Cryptology ePrint Archive, Paper 2024/143.

[60] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen. “cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2023).

[61] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *JACM* (1992).

[62] I. Mitsuru, S. Akira, and N. Takao. “Secret sharing scheme realizing general access structure”. In: *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* (1989).

[63] W. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh. “Mangrove: A scalable framework for folding-based SNARKs”. In: *Cryptology ePrint Archive* (2024).

[64] A. Ozdemir and D. Boneh. “Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets”. In: USENIX Security ’22.

[65] C. Papamanthou, E. Shi, and R. Tamassia. “Signatures of Correct Computation”. In: TCC ’13.

[66] L. Pearson, J. B. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia. “PlonKup: Reconciling PlonK with plookup”. In: *IACR Cryptol. ePrint Arch.* (2022).

[67] N. Pippenger. “On the Evaluation of Powers and Monomials”. In: *SIAM J. Comp.* (1980).

[68] A. Pruden. “zkCloud: Decentralized Private Computing”. https://aleo.org/post/zkcloud.

[69] S. Raghuraman, P. Rindal, and T. Tanguy. “Expand-Convolute Codes for Pseudorandom Correlation Generators from LPN”. In: CRYPTO ’23.

[70] C. Ran, Y. Lindell, R. Ostrovsky, and A. Sahai. “Universally composable two-party and multi-party secure computation”. In: STOC ’02.

[71] M. Rosenberg, J. D. White, C. Garman, and I. Miers. “zkcreds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure”. In: IEEE S&P ’23.

[72] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova. “Distributed Vector-OLE: Improved Constructions and Implementation”. In: CCS ’19.

[73] S. Setty. “Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup”. In: CRYPTO ’20.

[74] S. T. V. Setty, J. Thaler, and R. S. Wahby. “Unlocking the Lookup Singularity with Lasso”. In: EUROCRYPT ’24.

[75] S. Setty and J. Lee. “Quarks: Quadruple-efficient transparent zkSNARKs”. In: *ePrint* (2020).

[76] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* (1979).

[77] J. Thaler. “Proofs, Arguments, and Zero-Knowledge”. In: *Found. Trends Priv. Secur.* (2022).

[78] J. Thaler. “Time-Optimal Interactive Proofs for Circuit Evaluation”. In: CRYPTO ’13.

[79] A. Toshinori, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. “High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority”. In: CCS ’16.

[80] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. “A hybrid architecture for interactive verifiable computation”. In: S&P ’13.

[81] C. Weng, K. Yang, J. Katz, and X. Wang. “Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits”. In: IEEE S&P ’21.

[82] Z. J. Williamson. “The Aztec protocol”. URL: https://github.com/AztecProtocol/AZTEC.

[83] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. “DIZK: A Distributed Zero Knowledge Proof System”. In: USENIX Security ’18.

[84] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song. “zkBridge: Trustless Cross-chain Bridges Made Practical”. In: CCS’22.

[85] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: CRYPTO '19.

[86] T. Xie, Y. Zhang, and D. Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In:

[87] K. Yang, X. Wang, and J. Zhang. "More Efficient MPC from Improved Triple Generation and Authenticated Garbling". In: CCS '20.

[88] A. Zapico, A. Gabizon, D. Khovratovich, M. Maller, and C. Ràfols. "Baloo: Nearly Optimal Lookup Arguments". In: *IACR Cryptol. ePrint Arch.* (2022).

[89] "Zcash". https://z.cash/.

[90] J. Zhang, Z. Fang, Y. Zhang, and D. Song. "Zero knowledge proofs for decision tree predictions and accuracy". In: CCS '20.

[91] J. Zhang, T. Xie, Y. Zhang, and D. Song. "Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof". In: IEEE S&P '20.

[92] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun. "PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture". In: ISCA'21.

A Computation traces

In all scenarios, the total per-party input is less than 100 GB, which includes the SRS, the R1CS matrices, and the Beaver triples for the largest measured instance size of 2^{25}. Some of these are one-off costs (SRS, matrices), while others are incurred per-delegation (Beaver triples, witness shares). The per-delegation cost is at most 7.5 GiB, which does not include optimizations from EOS which would further reduce this cost by over half. One can also reduce Beaver triple costs entirely via techniques like pseudorandom correlation generators [16].

We further emphasize that DFS, like prior work, focuses on the online performance of proof generation after the witnesses have been secret-shared, and indeed all prior work incurs a similar per-delegation communication cost. In the private delegation setting, the computation trace of zkSaaS includes Az, Bz, Cz due to the reliance on packed secret sharing, while EOS's trace only includes z because their protocol computes shares of Az, Bz, Cz on the servers directly. DFS can do the same in EOS's setting (no distributed proving within each party), but distributed matrix-vector multiplication seems more challenging. In the public delegation setting, Pianist omits witness generation and distribution costs, and in fact generates the entire witness on each machine.

B Single machine evaluation

In Fig. 9, we evaluate the latency of DFS's prover on a single machine, comparing it against Marlin [24], which has been highlighted in prior work [64] to be more efficient in the private delegation. Fig. 9 illustrates that DFS's prover incurs significantly lower latency than Marlin. This is because DFS avoids FFTs and also requires fewer MSM operations.

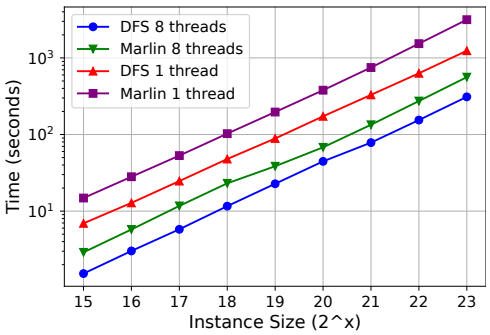

Figure 9: Proving time of single machine setting.

C Public delegation evaluation

We focus on two metrics: proof generation time and communication cost. In Fig. 10 and Fig. 11, we present the proof generation time. Note that some experiments stops earlier due to the memory constraints. We first observe that as the number of nodes increases, the proving latency decreases approximately linearly. Additionally, when the number of nodes is fixed, increasing the number of constraints results in a corresponding linear increase in proving latency. This demonstrates the scalability of our protocol with respect to both the number of nodes and the size of the constraints. Furthermore, our experiments show that for all measured instance sizes, the delegation protocol spends 70% of the time in the holographic protocol, which is consistent with the single machine scenario. It indicates that the holography part is the bottleneck of the whole protocol, which can be accelerated without the cost of MPC.

Fig. 12 reports the per-node communication cost during the proof generation. The cost grows logarithmically with the number of constraints. Furthermore, the per-node cost (slowly) *decreases* as we increase the number of nodes, since each node handles a smaller portion of the instance.

D Formal protocol of DFS PIOP

D.1 Common PIOPs

We now recall some common PIOPs that we will use in our construction of DFS. We omit their completeness, soundness, and zero-knowledge proofs as these can be found in prior work [78, 77, 23].

Sumcheck PIOP. Throughout this paper, we will be tasked with checking that an n-variate polynomial p sums to a claimed value σ over an n-dimensional Boolean hypercube $\{0,1\}^n$. This is formalized via the following relation:

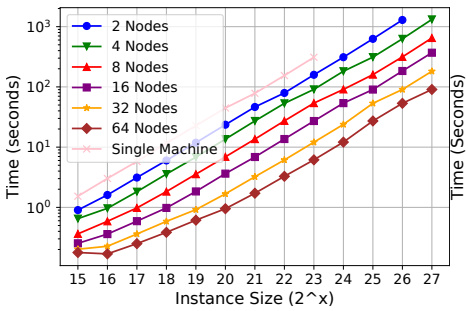

Figure 10: Latency of public delegation.

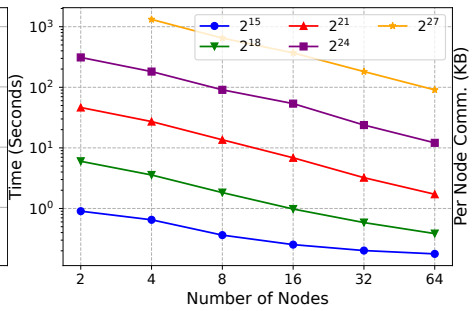

Figure 11: Public delegation scalability.

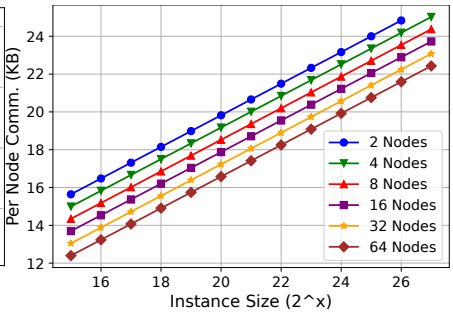

Figure 12: Comm. of public delegation.

Definition D.1. *The* **Sumcheck** *relation* $\mathscr{R}_{\text{SUM}}$ *is the set of tuples*

$$(\mathbb{i},\mathbb{x},\mathbb{w}) = (\perp,(\mathbb{F},n,\sigma),p(\boldsymbol{x}))$$

where $\sigma \in \mathbb{F}$*, and* $\sum_{\boldsymbol{x}\in\{0,1\}^n} p(\boldsymbol{x}) = \sigma$.

The PIOP below illustrates a standard way of proving this relation.

PIOP 1: PIOP for SUMCHECK

For each i in $1,\ldots,n$:

1. If $i = 1$, $\mathcal{V}$ sets $\sigma_i := \sigma$; otherwise, it sets $\sigma_i := p_{i-1}(r_{i-1})$.
2. $\mathcal{P}$ computes the sumcheck message $p_i(X_i) := \sum_{b_{i+1},\ldots,b_n\in\{0,1\}^{n-i}} p(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)$ and sends it to $\mathcal{V}$.
3. $\mathcal{V}$ checks that $p_i(0)+p_i(1) = \sigma_i$.
4. $\mathcal{V}$ samples a random point $r_i \in \mathbb{F}$ and sends it to $\mathcal{P}$.

Finally, $\mathcal{V}$ needs to check the polynomial oracle p at the evaluation point $(r_1,\ldots,r_n)$.

Zerocheck PIOP. Throughout this paper, we will be tasked with checking that an n-variate polynomial p is zero at all points of an n-dimensional Boolean hypercube $\{0,1\}^n$. This is formalized via the following relation:

Definition D.2. *The* **Zerocheck** *relation* $\mathscr{R}_{\text{ZERO}}$ *is the set of tuples*

$$(\mathbb{i},\mathbb{x},\mathbb{w}) = (\perp,(\mathbb{F},n),p(\boldsymbol{x}))$$

where $\forall \boldsymbol{x} \in \{0,1\}^n, p(\boldsymbol{x}) = 0$.

The PIOP below illustrates a standard way of proving this relation.

PIOP 2: PIOP for ZEROCHECK

$\mathcal{P}$ has input p, while $\mathcal{V}$ has oracle access to p.

1. $\mathcal{V}$ samples a random point $r \in \mathbb{F}^n$ and sends it to $\mathcal{P}$.
2. $\mathcal{P}$ and $\mathcal{V}$ invoke the sumcheck PIOP for the claim "$\sum_{X\in\{0,1\}^n} p(X)\cdot \text{eq}(X,r) = 0$".

Lookup PIOP. Another important building block is the lookup PIOP, where $\mathcal{P}$ has a *query* vector $\vec{q} \in \mathbb{F}^{2^n}$ and and a pre-shared *table* vector $\vec{t} \in \mathbb{F}^{2^m}$. The prover's goal is to assert that all elements of the query vector are contained in the table vector. In practice, the query and table vectors are represented as the evaluations of polynomials over the boolean hypercube.

This problem is formalized via the following relation:

Definition D.3. *The* **Lookup** *relation* $\mathscr{R}_{\text{LU}}$ *is the set of tuples*

$$(\mathbb{i},\mathbb{x},\mathbb{w}) = (\perp,(\mathbb{F},n),(p_0(\boldsymbol{x}),p_1(\boldsymbol{x}),p_2(\boldsymbol{x}),p_3(\boldsymbol{x})))$$

where $\{(p_0(\boldsymbol{x}),p_1(\boldsymbol{x}))\} \subset \{(p_2(\boldsymbol{x}),p_3(\boldsymbol{x}))\}$ *for* $\boldsymbol{x} \in \{0,1\}^n$.

The PIOP below illustrates a standard way of proving this relation that is adapted from [48].

PIOP 3: PIOP for LOOKUP

$\mathcal{P}$ gets as input (q,t), while $\mathcal{V}$ gets as input $(\mathbb{F},n,m,d)$ and oracles for q and t.

1. $\mathcal{P}$ receives a random challenge $r \in \mathbb{F}$ from $\mathcal{V}$.
2. $\mathcal{P}$ computes polynomials $h_1(X), h_2(X)$ such that for each $x \in \{0,1\}^n$, $h_1(x) = (r+q(x))^{-1}$, and for each $x \in \{0,1\}^m$, $h_2(x) = (r+t(x))^{-1}$. That is, h_1 and h_2 are multilinear extensions of $(r+q(x))^{-1}$ and $(r+t(x))^{-1}$, respectively.
3. $\mathcal{P}$ sends $\boldsymbol{h_1}(\boldsymbol{x})$ and $\boldsymbol{h_2}(\boldsymbol{x})$ to $\mathcal{V}$.
4. $\mathcal{P}$ evaluates $k := \sum_x h_1(x)$. Then, $\mathcal{P}$ and $\mathcal{V}$ invoke two Sumcheck PIOPs: one for the claim "$\sum_x h_1(x) = k$", and another for the claim "$\sum_x h_2(x) = k$".
5. $\mathcal{P}$ and $\mathcal{V}$ invoke a Zerocheck PIOP for the claim "$(r+q(X))h_1(X) - 1 = 0$".
6. $\mathcal{P}$ and $\mathcal{V}$ invoke a Zerocheck PIOP for the claim "$(r+t(X))h_2(X) - 1 = 0$".

This PIOP shows how to perform lookups over scalars, where each element of the query or table vectors is a single field element. Prior work [37, 48] has shown how to batch check multiple lookup relations with a single lookup proof.

D.2 Polynomial commitments

DFS uses the PST13 polynomial commitment scheme [65]. This construction requires a one-time, universal trusted setup, distributing the committer and verifier keys to the appropriate parties. This setup determines the maximum degree of the polynomials we can commit to. For the PC.Commit and PC.Open algorithms, we omit the hiding randomness $\bar{p}$ for simplicity; see [24] for details.

Prot. 2: MULTILINEAR POLYNOMIAL COMMITMENT

$\mathsf{PST.Setup}(1^\lambda, n) \to (\mathsf{ck}, \mathsf{rk})$:
1. Obtain $\langle \mathsf{group} \rangle = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G, H) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. Sample random $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \leftarrow \mathbb{F}^n$.
3. Set $\boldsymbol{\Sigma} := [\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}$.
4. Set $\mathsf{ck} := (\boldsymbol{\Sigma}, \langle \mathsf{group} \rangle)$.
5. Set $\mathsf{rk} := ([\alpha_i \cdot H]_{i \in [1..n]}, \langle \mathsf{group} \rangle)$.
6. Output $(\mathsf{ck}, \mathsf{rk})$.

$\mathsf{PST.Commit}(\mathsf{ck}, p) \to \mathsf{cm}$:
1. Parse ck as $([\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}, G, H)$.
2. Output $\mathsf{cm} := \sum_{i \in \{0,1\}^n} p_i \cdot \mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G$.

$\mathsf{PST.Open}(\mathsf{ck}, p, \boldsymbol{z}) \to \pi_{\mathsf{PC}}$:
Parse: $\mathsf{ck} = ([\mathrm{eq}(\boldsymbol{\alpha}, i) \cdot G]_{i \in \{0,1\}^n}, G, H)$.
1. Let $y := p(\boldsymbol{z})$
2. For each i in $[1, \ldots, n]$:
   (a) Compute i-th witness polynomial $q_i(X)$ such that $p(X) - y = \sum_{i=1}^n q_i(X) \cdot (X_i - z_i)$
   (b) Compute $\pi_i := q_i(\boldsymbol{\alpha}) \cdot G$.
3. Output evaluation proof $\pi_{\mathsf{PC}} := (\pi_1, \ldots, \pi_n)$.

$\mathsf{PST.Check}(\mathsf{rk}, \mathsf{cm}, \boldsymbol{z}, v, \pi_{\mathsf{PC}}) \to \{0, 1\}$:
Parse: $\mathsf{rk} = ([\alpha_i \cdot H]_{i \in [1..n]}, G, H)$ and $\pi_{\mathsf{PC}} = (\pi_1, \ldots, \pi_n)$.
1. Accept if $e(\mathsf{cm} - vG, H) = \sum_{i=1}^n e(\pi_i, (\alpha_i - z_i) \cdot H)$.

D.3 DFS full PIOP

We present our formal protocol description as follows. For simplicity, we assume that there is only private witness and no public input.

PIOP 4: PIOP for R1CS

Indexer I: on input $(\mathbb{F}, n, m, A, B, C)$, proceeds as follows:
1. For each $M \in \{A, B, C\}$:
   (a) Derive polynomials $\boldsymbol{r_M}(\boldsymbol{x})$, $\boldsymbol{c_M}(\boldsymbol{x})$, and $\boldsymbol{v_M}(\boldsymbol{x})$ from M. Output these polynomials.

Initialization: $\mathcal{P}$ gets as input the witness $\boldsymbol{z} = (w)$, as well as A, B, and C, while $\mathcal{V}$ gets the polynomial oracles of $\boldsymbol{r_M}(\boldsymbol{x}), \boldsymbol{c_M}(\boldsymbol{x}), \boldsymbol{v_M}(\boldsymbol{x})$ for $M \in \{A, B, C\}$.

Protocol:
1. $\mathcal{P}$ sends the oracle $\boldsymbol{z}(\boldsymbol{x})$ to the $\mathcal{V}$.
2. Let $\hat{M}(x) := \sum_{\boldsymbol{y} \in \{0,1\}^s} \boldsymbol{M}(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{z}(\boldsymbol{y})$, for $M \in \{A, B, C\}$, and $\boldsymbol{F}(\boldsymbol{x}) = \hat{A}(\boldsymbol{x}) \cdot \hat{B}(\boldsymbol{x}) - \hat{C}(\boldsymbol{x})$. $\mathcal{P}$ and $\mathcal{V}$ invoke the Zerocheck PIOP (PIOP 2) on the polynomial $\boldsymbol{F}$. This leads to an evaluation claim of the form $e_x = \boldsymbol{F}(\boldsymbol{\rho_x}) \cdot \mathrm{eq}(\boldsymbol{r}, \boldsymbol{\rho_x})$ for a zerocheck challenge $\boldsymbol{r}$ and random point $\boldsymbol{\rho_x} \in \mathbb{F}^s$.
3. To answer this claim, $\mathcal{P}$ computes $v_M := \hat{M}(\boldsymbol{\rho_x})$ for each $M \in \{A, B, C\}$, and sends v_A, v_B, v_C to $\mathcal{V}$.
4. $\mathcal{V}$ asserts that $e_x \stackrel{?}{=} (v_A \cdot v_B - v_C) \cdot \mathrm{eq}(\boldsymbol{r}, \boldsymbol{\rho_x})$.
5. $\mathcal{V}$ randomly samples $r_A, r_B, r_C \in \mathbb{F}$, and sends them to $\mathcal{P}$.
6. $\mathcal{P}$ computes $\hat{M}_{\boldsymbol{\rho_x}}(\boldsymbol{y}) := (r_A \cdot \boldsymbol{A}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_B \cdot \boldsymbol{B}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_C \cdot \boldsymbol{C}(\boldsymbol{\rho_x}, \boldsymbol{y}))\boldsymbol{z}(\boldsymbol{y})$.
7. $\mathcal{P}$ and $\mathcal{V}$ engage in a Sumcheck PIOP for the claim "$\sum_{\boldsymbol{y} \in \{0,1\}^s} \hat{M}_{\boldsymbol{\rho_x}}(\boldsymbol{y}) = (r_A v_A + r_B v_B + r_C v_C)$". This leads to an evaluation claim of the form $e_y \stackrel{?}{=} \hat{M}_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$, where $\boldsymbol{\rho_y} \in \mathbb{F}^s$ is a random evaluation point.
8. $\mathcal{V}$ queries the oracle $v_Z := \boldsymbol{z}(\boldsymbol{\rho_y})$. Then, the verifier asserts that $e_y \stackrel{?}{=} (r_A \cdot \boldsymbol{A}(\boldsymbol{\rho_x}, \boldsymbol{\rho_y}) + r_B \cdot \boldsymbol{B}(\boldsymbol{\rho_x}, \boldsymbol{\rho_y}) + r_C \cdot \boldsymbol{C}(\boldsymbol{\rho_x}, \boldsymbol{\rho_y})) \cdot v_Z$
9. For each $M \in \{A, B, C\}$:
   (a) $\mathcal{P}$ sends oracles for $\mathrm{eq}_{row}(\boldsymbol{x})$ and $\mathrm{eq}_{col}(\boldsymbol{x})$.
   (b) $\mathcal{P}$ and $\mathcal{V}$ invoke a Sumcheck PIOP for the claim $\sum \boldsymbol{v}(\boldsymbol{x}) \mathrm{eq}_{row}(\boldsymbol{x}) \mathrm{eq}_{col}(\boldsymbol{x}) = M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$, resulting in a random challenge $\boldsymbol{\rho_z}$ and claimed evaluation e_z.
   (c) $\mathcal{V}$ uses oracles to assert $\boldsymbol{v}_M(\boldsymbol{\rho_z}) \mathrm{eq}_{row}(\boldsymbol{\rho_z}) \mathrm{eq}_{col}(\boldsymbol{\rho_z}) \stackrel{?}{=} e_z$
   (d) $\mathcal{P}$ and $\mathcal{V}$ invoke the batched lookup PIOP where $q_1(\boldsymbol{x}) := \boldsymbol{r}_M(\boldsymbol{x})$, $q_2(\boldsymbol{x}) := \mathrm{eq}_{row}(\boldsymbol{x})$, t_1 is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(\boldsymbol{x}) := \mathrm{eq}(\boldsymbol{x}, \boldsymbol{\rho_x})$.
   (e) $\mathcal{P}$ and $\mathcal{V}$ invoke the batched lookup PIOP for $q_1(\boldsymbol{x}) := \boldsymbol{c}_M(\boldsymbol{x})$, $q_2(\boldsymbol{x}) := \mathrm{eq}_{col}(\boldsymbol{x})$, t_1 is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(\boldsymbol{x}) := \mathrm{eq}(\boldsymbol{x}, \boldsymbol{\rho_y})$.

E Formal protocols of DFS with public delegation

DFS naturally lends itself to the *public delegation* setting, where N_w trusted nodes collaborate to generate a proof with the help of a central coordinator $\mathcal{D}$. Note that in the public setting, we do not protect the privacy of the witness; thus, all nodes are controlled by a single trusted party. The central coordinator is responsible for gathering data from all nodes to produce the final proof. Note that the coordinator's cost is logarithmic, either one of the nodes or the proof delegator can serve as the coordinator. The witness is not hidden from the nodes in this setting. We denote each node as $\mathcal{E}_j$, for $j \in [N_w]$, and the set of all nodes as $\mathcal{E}_{1 \ldots N_w}$. For convenience, we denote $n_w := \log(N_w)$.

E.1 Public delegation protocols

We describe distributed versions of the PIOPs in Section D.1. In general, whenever we have an n-variate polynomial $p(x)$, we can split it into N_w parts $p^{(j)}(x)$, each of which is $n - n_w$-variate. For $p(x)$ in evaluation form, we can simply give each node 2^{n-n_w} consecutive evaluations. Each node essentially has the original n-variate polynomial with the first n_w variables fixed. $\mathcal{E}_1$ has the first n_w variables set to $(0, 0, \cdots, 0)$, $\mathcal{E}_2$ has the first n_w variables set to $(0, 0, \cdots, 1)$, etc.

Distributed sumcheck. For the first $n-n_w$ rounds, each node will locally perform their own sumcheck, and the coordinator will be responsible for producing Fiat-Shamir challenges and computations of the last n_w rounds.

Prot. 3: PUBLICLY DELEGATED SUMCHECK

1. For each i in $1,\ldots,n-n_w$:
   (a) All nodes $\mathcal{E}_j$ compute $p_i^{(j)}(X_i) := \sum_{b_{i+1},\ldots,b_{n-n_w}\in W} p^{(j)}(\ldots,r_{i-1},X_i,b_{i+1},\ldots,)$, where $W=\{0,1\}^{n-n_w-i}$ and sends it to $\mathcal{D}$.
   (b) $\mathcal{D}$ outputs $\sum_j p_i^{(j)}(0)$ and $\sum_j p_i^{(j)}(1)$.
   (c) $\mathcal{D}$ performs Fiat-Shamir and sends a random point $r_i\in\mathbb{F}$ to all $\mathcal{E}_j$.
2. Each node computes $y^{(j)} := p^{(j)}(r_1,r_2,\ldots,r_{n-n_w})$ and sends it to the coordinator.
3. $\mathcal{D}$ interpolates the n_w-variate polynomial $p^*(\boldsymbol{x})$ from the evaluations $y^{(j)}$.
4. $\mathcal{D}$ locally runs the non-distributed sumcheck for $\sum p^*(\boldsymbol{x}) = \sigma_{n-n_w}$, where σ_{n-n_w} is final σ_i from the last loop.

Distributed polynomial evaluation. Each node has a $n-n_w$-variate chunk of the polynomials p, and the evaluation point $\boldsymbol{z}$.

Prot. 4: PUBLICLY DELEGATED POLYNOMIAL EVALUATION

1. Let $\boldsymbol{z}^* := \boldsymbol{z}[:n_w]$, and $\boldsymbol{z}_w := \boldsymbol{z}[n_w+1:n]$.
2. $\mathcal{E}_j$ computes the partial evaluation of $p^{(j)}(\boldsymbol{z}_w)$ according to their id j, and sends to the $\mathcal{D}$.
3. The coordinator $\mathcal{D}$ interpolates the n_w variate polynomial p^* from the evaluations $(p^{(1)}(\boldsymbol{z}_w), p^{(2)}(\boldsymbol{z}_w),\cdots,p^{(N_w)}(\boldsymbol{z}_w))$. Then $\mathcal{D}$ outputs the evaluation $p^*(\boldsymbol{z}^*)$.

Distributed zerocheck. Each node has a $n-n_w$-variate chunk of the polynomials p.

Prot. 5: PUBLICLY DELEGATED ZEROCHECK

1. $\mathcal{D}$ obtains $r\in\mathbb{F}^n$ via the Fiat-Shamir and sends it to $\mathcal{E}_{1\ldots N_w}$.
2. $\mathcal{E}_j$ computes the partial evaluation of $\mathrm{eq}(\boldsymbol{x},\boldsymbol{r})$ according to their id j. $\mathcal{E}_j$ will take evaluations from $j\cdot(n-n_w)$ to $(j+1)\cdot(n-n_w)$ to form their own $\mathrm{eq}^{(j)}(\boldsymbol{x},\boldsymbol{r})$
3. Each node defines $q_j(\boldsymbol{x}) := p^{(j)}(\boldsymbol{x})\cdot\mathrm{eq}^{(j)}(\boldsymbol{x},\boldsymbol{r})$ and do a Distributed Sumcheck (Prot. 3) for the claim $\sum q(\boldsymbol{x})=0$.

Distributed lookup. Each node has a $n-n_w$-variate and $m-n_w$-variate chunk of the polynomials q and t respectively.

Prot. 6: PUBLICLY DELEGATED LOOKUP

1. $\mathcal{D}$ runs the Fiat-Shamir and sends the random challenge $r\in\mathbb{F}$ to each $\mathcal{E}_j$.
2. $\mathcal{E}_j$ computes polynomials $h_1^{(j)}(\boldsymbol{x}), h_2^{(j)}(\boldsymbol{x})$ such that for $\boldsymbol{x}\in\{0,1\}^{n-n_w}$, $h_1^{(j)}(\boldsymbol{x}) = (r+q^{(j)}(\boldsymbol{x}))^{-1}$, and for each $\boldsymbol{x}\in\{0,1\}^{m-n_w}$, $h_2^{(j)}(\boldsymbol{x}) = (r+t^{(j)}(\boldsymbol{x}))^{-1}$.
3. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed polynomial commitment (Prot. 7) to output the commitment of h_1 and h_2.
4. $\mathcal{E}_j$ evaluates $k_j := \sum h_1^{(j)}(\boldsymbol{x})$, and sends it to $\mathcal{D}$, who sums up every $k := \sum k_j$.
5. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke two Distributed Sumcheck (Prot. 3): one for the claim "$\sum_{\boldsymbol{x}} h_1(\boldsymbol{x}) = k$", and "$\sum_{\boldsymbol{x}} h_2(\boldsymbol{x}) = k$".
6. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke a Distributed Zerocheck (Prot. 5) for the claim "$(r+q(\boldsymbol{x}))h_1(\boldsymbol{x})-1=0$", result in the evaluation point $\boldsymbol{\rho_x}$.
7. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke a Distributed Zerocheck (Prot. 5) for the claim "$(r+p_2(\boldsymbol{x}))h_2(\boldsymbol{x})-1=0$", result in the evaluation point $\boldsymbol{\rho_y}$..
8. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed polynomial commitment (Prot. 7) to output the opening proof for $h_1(\boldsymbol{\rho_x}), q(\boldsymbol{\rho_x}), h_2(\boldsymbol{\rho_y}), t(\boldsymbol{\rho_y})$.

Distributed polynomial commitment scheme. Each node will receive a smaller committer key, as well a $n-n_w$-variate chunk of the secret polynomial.

Prot. 7: PUBLICLY DELEGATED POLYNOMIAL COMMITMENT

$\mathsf{PST.Commit}\to\mathsf{cm}$:

1. Each node computes $\mathsf{cm}^{(j)} = \mathsf{PST.Commit}(\mathsf{ck}^{(j)}, p^{(j)})$.
2. The coordinator gathers and outputs $\mathsf{cm} = \sum_{j\in 1\ldots N_w}\mathsf{cm}^{(j)}$

$\mathsf{PST.Open}\to\pi_{\mathsf{PC}}$:

1. Let $\boldsymbol{z}^* := \boldsymbol{z}[:n_w]$, and $\boldsymbol{z}_w := \boldsymbol{z}[n_w+1:n]$.
2. For each $\mathcal{E}_j$,
   (a) Compute $y^{(j)} := p^{(j)}(\boldsymbol{z}_w)$
   (b) Compute i-th witness polynomial $q_i^{(j)}(\boldsymbol{x})$ such that $p^{(j)}(\boldsymbol{x}) - y^{(j)} = \sum_{i=1}^{n-n_w} q_i^{(j)}(\boldsymbol{x})\cdot(\boldsymbol{x}_i - \boldsymbol{z}_w[i])$.
   (c) For each i in $1\ldots n-n_w$: compute $\pi_i^{(j)} := q_i^{(j)}(\boldsymbol{\alpha})\cdot G$, and sends to the coordinator $\mathcal{D}$.
3. For each i in $1\ldots n-n_w$: $\mathcal{D}$ gathers $\pi_{\mathsf{PC}}^w[i] = \sum\pi_i^{(j)}$
4. Each node sends $r^{(j)} := p^{(j)}(\boldsymbol{z}_w)$ to the coordinator $\mathcal{D}$, which is a byproduct of the $\pi_{\mathsf{PC}}^{(j)}$ computation.
5. The coordinator $\mathcal{D}$ interpolates the n_w variate polynomial p^* from the evaluations $(r^{(1)}, r^{(2)},\cdots,r^{(N_w)})$.
6. $\mathcal{D}$ obtains $\pi_{\mathsf{PC}}^* := \mathsf{PST.Open}(\mathsf{ck}^*, p^*, \boldsymbol{z}^*)$.
7. $\mathcal{D}$ outputs $\pi_{\mathsf{PC}} := \pi_{\mathsf{PC}}^w \,||\, \pi_{\mathsf{PC}}^*$.

Publicly delegated DFS. Using these building blocks, we can build DFS for the public delegation setting.

Prot. 8: PUBLICLY DELEGATED DFS

Initialization: The sparse matrix encodings $\boldsymbol{r}_M$, $\boldsymbol{c}_M$, and

$\boldsymbol{v}_M$ are splited into N_w parts for each $M \in \{A,B,C\}$. We denote the jth node's part of a certain polynomial p as $p^{(j)}$. In addition, the computation trace Az, Bz, Cz and the witness z are also distributed to each node as the $s-n_w$-variate polynomials $\hat{A}^{(j)}(\boldsymbol{x}), \hat{B}^{(j)}(\boldsymbol{x}), \hat{C}^{(j)}(\boldsymbol{x}), z^{(j)}(\boldsymbol{x})$.

Protocol:

1. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed polynomial commitment (Prot. 7) to commit $z(\boldsymbol{x})$ and output the commitment.
2. Let $\boldsymbol{F}^{(j)}(\boldsymbol{x}) := \hat{A}^{(j)}(\boldsymbol{x}) \cdot \hat{B}^{(j)}(\boldsymbol{x}) - \hat{C}^{(j)}(\boldsymbol{x})$. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed zerocheck (Prot. 5) on the polynomial $\boldsymbol{F}$. This leads to an evaluation claim of the form $\boldsymbol{F}(\boldsymbol{\rho_x})\mathrm{eq}(\boldsymbol{\rho_x}, \boldsymbol{r}) = e_x$ for a random point $\boldsymbol{\rho_x} \in \mathbb{F}^s$, as well as the zerocheck challenge $\boldsymbol{r} \in \mathbb{F}^s$.
3. For each $M \in \{A,B,C\}$
   (a) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ engage in a distributed polynomial evaluation protocol (Prot. 4) to evaluate and output $v_M := \hat{M}(\boldsymbol{\rho_x})$.
4. $\mathcal{D}$ runs the Fiat-Shamir and randomly samples $r_A, r_B, r_C \in \mathbb{F}$, and sends them to $\mathcal{E}_{1\ldots N_w}$.
5. Each $\mathcal{E}_j$ individually computes $M^{(j)}_{\boldsymbol{\rho_x}}(\boldsymbol{y}) := (r_A \cdot \boldsymbol{A}^{(j)}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_B \cdot \boldsymbol{B}^{(j)}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_C \cdot \boldsymbol{C}^{(j)}(\boldsymbol{\rho_x}, \boldsymbol{y}))z^{(j)}(\boldsymbol{y})$.
6. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ engage in a distributed sumcheck (Prot. 3) for the claim "$\sum_{\boldsymbol{y}\in\{0,1\}^s} M_{\boldsymbol{\rho_x}}(\boldsymbol{y}) = r_A v_A + r_B v_B + r_C v_C$". This leads to an evaluation claim of the form $M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$, where $\boldsymbol{\rho_y} \in \mathbb{F}^s$ is a random evaluation point.
7. $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed polynomial commitment (Prot. 7) to open $z(\boldsymbol{\rho_y})$ and outputs the opening proof.
8. For each $M \in \{A,B,C\}$:
   (a) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ runs the distributed polynomial commitment (Prot. 7) to output the commitment for eq_{row} and eq_{col}.
   (b) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke a distributed sumcheck (Prot. 3) for the claim $\sum \boldsymbol{v}(\boldsymbol{x})\mathrm{eq}_{row}(\boldsymbol{x})\mathrm{eq}_{col}(\boldsymbol{x}) = M_{\boldsymbol{\rho_x}}(\boldsymbol{\rho_y})$, resulting in a random challenge $\boldsymbol{\rho_z}$ and claimed evaluation e_z.
   (c) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ runs distributed polynomial commitment scheme to output the opening proofs for $\boldsymbol{v}_M(\boldsymbol{\rho_z}), \mathrm{eq}_{row}(\boldsymbol{\rho_z}), \mathrm{eq}_{col}(\boldsymbol{\rho_z})$.
   (d) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed batched lookup (Prot. 6) where $q_1(\boldsymbol{x}) := \boldsymbol{r}_M(\boldsymbol{x})$, $q_2(\boldsymbol{x}) := \mathrm{eq}_{row}(\boldsymbol{x})$, t_1 is the polynomial interpolated from $(0, 1, \cdots, n)$, and $t_2(\boldsymbol{x}) := \mathrm{eq}(\boldsymbol{x}, \boldsymbol{\rho_x})$.
   (e) $\mathcal{E}_{1\ldots N_w}$ and $\mathcal{D}$ invoke the distributed batched lookup (Prot. 6) for $q_1(\boldsymbol{x}) := \boldsymbol{c}_M(\boldsymbol{x})$, $q_2(\boldsymbol{x}) := \mathrm{eq}_{col}(\boldsymbol{x})$, t_1 is the same as in the previous step, and $t_2(\boldsymbol{x}) := \mathrm{eq}(\boldsymbol{x}, \boldsymbol{\rho_y})$.

E.2 Performance analysis

Each of these distributed PIOPs only incurs $O(\log(m))$ or $O(\log(n))$ communication cost for each node, where m is the size of input and n is the number of non-zero entries. In terms of the computation cost, note that in the step 8 of Prot. 8, we cannot use use the same method as in the non-delegated scenario to compute the chunk of $\mathrm{eq}_{row}(\boldsymbol{x})$ in each node, because the precomputed results would require $O(m)$ space, while each node may not have enough space to store. If we were to store the precomputed data in a distributed manner, additional communication would be needed to retrieve the results from other nodes during the construction of $\mathrm{eq}_{row}(\boldsymbol{x})$ for each node. This extra communication could introduce significant overhead since it might incur random memory access.

To avoid this issue, we can use a different approach to compute $\mathrm{eq}_{row}(\boldsymbol{x})$. Recall that $\mathrm{eq}(\boldsymbol{x},\boldsymbol{y}) := \prod_{i=1}^{\log m}(\boldsymbol{x}_i\boldsymbol{y}_i + (1-\boldsymbol{x}_i)(1-\boldsymbol{y}_i)) = \prod_{i=1}^{\log m - \log N_w}(\boldsymbol{x}_i\boldsymbol{y}_i + (1-\boldsymbol{x}_i)(1-\boldsymbol{y}_i)) \cdot \prod_{j=\log m - \log N_w + 1}^{\log m}(\boldsymbol{x}_j\boldsymbol{y}_j + (1-\boldsymbol{x}_j)(1-\boldsymbol{y}_j))$. We ask each node to precompute and store both of these two product terms, which takes $O(\frac{m}{N_w} + N_w)$ time. Then each node can compute its chunk of $\mathrm{eq}_{row}(\boldsymbol{x})$ by multiplying the two precompute product terms in $O(\frac{n}{N_w})$ time. In summary, per node cost is $O(N_w + \frac{m+n}{N_w})$ computation and $O(\log(n) + \log(n))$ communication.

The security, verification cost and proof size are not affected by the distributed implementation of the prover.

F Instantiations for linear secret sharings

F.1 Replicated secret sharing scheme

The scheme of replicated secret sharing (RSS) [62, 28] is described as below. Although we focus on the three-party setting, we provide the description of the RSS scheme for $n = 2t+1$ parties.

- $[\![x]\!] \leftarrow \mathsf{Share}(x)$: On input a secret $x \in \mathbb{F}$, a dealer $\mathcal{D}$ samples $x^T \leftarrow \mathbb{F}$ for $T \in \mathcal{T}$ such that $\sum_{T\in\mathcal{T}} x^T = x$, where $\mathcal{T}$ consists of all sets of t parties. Then, for each $T \in \mathcal{T}$, $\mathcal{D}$ sends x^T to $\mathcal{P}_i$ such that $i \notin T$. Every party $\mathcal{P}_i$ holds a set of shares denoted by $[\![x]\!]_i$, which consists of x^T for all $T \in \mathcal{T}$ such that $i \notin T$. The sharing $[\![x]\!]$ is defined as $([\![x]\!]_1, \ldots, [\![x]\!]_n)$, and involves $x^T \in \mathbb{F}$ for each $T \in \mathcal{T}$.
- $x \leftarrow \mathsf{Rec}([\![x]\!], \mathcal{B})$: For each $T \in \mathcal{T}$, any one party $\mathcal{P}_i$ with $i \notin T$ sends $x^T \in \mathbb{F}$ to $\mathcal{B}$, who computes $x := \sum_{T\in\mathcal{T}} x^T$.

The total number of shares is $\binom{n}{t}$ and every party stores $\binom{n-1}{t}$ shares, which would become very large as n and t grow. Thus, we use RSS when only n is small. In this work, we focus on the case where $n = 3$ and $t = 1$, the dealer samples $x^{\{1\}}, x^{\{2\}}, x^{\{3\}} \leftarrow \mathbb{F}$ such that $x^{\{1\}} + x^{\{2\}} + x^{\{3\}} = x$, and sends $(x^{\{2\}}, x^{\{3\}})$ to P_1, $(x^{\{1\}}, x^{\{3\}})$ to $\mathcal{P}_2$ and $(x^{\{1\}}, x^{\{2\}})$ to $\mathcal{P}_3$.

It is well-known that RSS satisfies the authentication property in the honest-majority setting. However, our private-delegation protocol does not adopt the property, and the reconstruction protocol only needs to receive the shares of any $t+1$ parties in order to recover the secret, which is sufficient to guarantee security for our protocol. Following prior works such as [79, 57, 17], RSS satisfies the multiplication property,

i.e., all parties can locally compute an additive secret sharing $\langle z \rangle$ from two replicated secret sharings $[\![x]\!]$ and $[\![y]\!]$.

F.2 Additive secret sharing scheme

For additive secret sharings (AddSS), we consider $t = n-1$. The classic AddSS scheme is described as follows:

- $\langle x \rangle \leftarrow \mathsf{Share}(x)$: On input a secret $x \in \mathbb{F}$, a dealer $\mathcal{D}$ samples a share $\langle x \rangle_i \leftarrow \mathbb{F}$ for $i \in [1..n]$ such that $\sum_{i\in[1..n]} \langle x \rangle_i = x$, and then sends $\langle x \rangle_i$ to $\mathcal{P}_i$ for $i \in [1..n]$. The resulting sharing is defined as $\langle x \rangle = (\langle x \rangle_1, \ldots, \langle x \rangle_n)$.
- $x \leftarrow \mathsf{Rec}(\langle x \rangle, \mathcal{B})$: For $i \in [1..n]$, $\mathcal{P}_i$ sends its share $\langle x \rangle_i$ to $\mathcal{B}$, who computes $x := \sum_{i\in[1..n]} \langle x \rangle_i$.

Additive secret sharing does not guarantee the correctness of values to be reconstructed. This can be solved using standard authenticated sharings, i.e., additive secret sharings equipped with information-theoretic message authentication codes (IT-MACs). For example, SPDZ-like IT-MACs [30, 29] is in the form of $\mathsf{M} = x \cdot \Delta \in \mathbb{F}$, where x is a secret, Δ is a key and M is an MAC tag. Moreover, pseudorandom correlation generator (PCG) [11, 16] can be used to generate vector oblivious linear evaluation (VOLE) correlations, which would in turn be used to preprocess SPDZ-like authenticated sharings with sublinear cost.

Additive secret sharing does not guarantee the correctness of values to be reconstructed. This can be solved using authenticated sharings, i.e., additive secret sharings equipped with information-theoretic message authentication codes (IT-MACs). In this work, we focus on SPDZ-like IT-MACs [30, 29] in the form of $\mathsf{M} = x \cdot \Delta \in \mathbb{F}$, where x is a secret, Δ is a key and M is an MAC tag. In particular, an SPDZ-like authenticated sharing on $x \in \mathbb{F}$ is defined as $[\![x]\!] = (\langle x \rangle, \langle \mathsf{M} \rangle, \langle \Delta \rangle)$, i.e.,

$$x = \sum_{i=1}^{n} \langle x \rangle_i \in \mathbb{F}, \quad \mathsf{M} = \sum_{i=1}^{n} \langle \mathsf{M} \rangle_i \in \mathbb{F}, \quad \Delta = \sum_{i=1}^{n} \langle \Delta \rangle_i \in \mathbb{F}.$$

Due to the linearity of additive secret sharings, SPDZ-like authenticated sharings are additively homomorphic. The Open procedure on $[\![x]\!]$ is easy to be constructed by running $\mathsf{Open}(\langle x \rangle)$. Note that $\mathsf{Open}(\langle x \rangle)$ requires $O(n^2)$ communication and one round. This can be reduced to $O(n)$ communication complexity at the cost of two rounds. In particular, the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ first run $\mathsf{Rec}(\langle x \rangle, \mathcal{P}_1)$, and then $\mathcal{P}_1$ sends x to other parties. To guarantee the correctness of x, a checking procedure, based on IT-MACs, needs to be performed. Following the SPDZ protocol [30, 29], this procedure can be done in a batch, which enables the communication cost to be independent of the number of values opened. Specifically, given authenticated sharings $[\![x_1]\!], \ldots, [\![x_\ell]\!]$ and opened values $x_1, \ldots, x_\ell$, the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ execute the following $\mathsf{BatchCheck}\big((\![x_i]\!])_{i\in[1,n]}, (x_i)_{i\in[1,n]}\big)$ procedure to check the correctness of these opened values.

1. All parties compute $(\chi_1, \ldots, \chi_\ell) := \mathsf{H}(x_1, \ldots, x_\ell)$, where $\mathsf{H} : \mathbb{F}^\ell \to \mathbb{F}^\ell$ is a random oracle.
2. The parties compute $y := \sum_{j=1}^{\ell} \chi_j \cdot x_j \in \mathbb{F}$ and $\langle \mathsf{M}[y] \rangle := \sum_{j=1}^{\ell} \chi_j \cdot \langle \mathsf{M}[x_h] \rangle$, and then set $\langle \sigma \rangle := \langle \mathsf{M}[y] \rangle - y \cdot \langle \Delta \rangle$.
3. Every party $\mathcal{P}_i$ commits to $\langle \sigma \rangle_i$, and then opens it after all shares on $\langle \sigma \rangle$ are committed, where the commitments can be computed with a random oracle (see, e.g., [50, 20, 87]). Then, every party $\mathcal{P}_i$ checks that $\sum_{i=1}^{n} \langle \sigma \rangle_i = 0$, and aborts if the check fails.

Following prior works [30, 29, 52, 53], the probability that there exists some x_i opened incorrectly, is bounded by $(q_{\mathsf{H}} + 2)/|\mathbb{F}|$, where q_{H} is the number of queries to random oracle H and $\mathbb{F}$ is exponentially large.

F.3 Pseudorandom correlation generators

We adopt the pseudorandom correlation generator (PCG) [11, 16] to generate vector oblivious linear evaluation (VOLE) correlations, which would in turn be used to generate SPDZ-like authenticated sharings. While the notion of PCG is general, we focus on PCG for generating random VOLE correlations. A probabilistic polynomial time (PPT) algorithm $\mathcal{G}$ is called a VOLE correlation generator, if $\mathcal{G}$ takes as input 1^λ and outputs a pair $(R_0 = (\boldsymbol{u}, \boldsymbol{v}), R_1 = (\boldsymbol{w}, \Delta))$ such that $\boldsymbol{u} \in \mathbb{F}^\ell$, $\boldsymbol{v} \in \mathbb{F}^\ell$ and $\Delta \in \mathbb{F}$ are uniformly random, and $\boldsymbol{w} = \boldsymbol{v} + \boldsymbol{u} \cdot \Delta \in \mathbb{F}^\ell$ holds. Such generator $\mathcal{G}$ is *reverse sampleable* [16], i.e., there exists an algorithm $\mathsf{RSample}$, in which given R_b for $b \in \{0,1\}$, it sets $R'_b := R_b$ and outputs R'_{1-b} such that (R'_0, R'_1) is computationally indistinguishable from $(R_0, R_1) \leftarrow \mathcal{G}(1^\lambda)$. In the following, we recall the definition of PCG [16] for VOLE.

Definition F.1. *Let $\mathcal{G}$ be a reverse-sampleable VOLE correlation generator and* $\mathsf{RSample}$ *be the reverse sampling algorithm for $\mathcal{G}$. A pseudorandom correlation generator (PCG) for $\mathcal{G}$ is a pair of algorithms* $(\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *with the following syntax:*

- $\mathsf{PCG.Gen}(1^\lambda)$ *is a PPT algorithm that takes as input 1^λ, and outputs a pair of seeds* $(\mathsf{k}_0, \mathsf{k}_1)$.
- $\mathsf{PCG.Expand}(b, \mathsf{k}_b)$ *is a polynomial-time deterministic algorithm that takes as input a bit $b \in \{0,1\}$ and a seed k_b, and outputs R_b such that $R_b = (\boldsymbol{u}, \boldsymbol{v})$ if $b = 0$ and $R_b = (\boldsymbol{w}, \Delta)$ if $b = 1$.*

The pair of algorithms $(\mathsf{PCG.Gen}, \mathsf{PCG.Expand})$ *should satisfy the following properties:*

- **Correctness and pseudorandomness.** *The VOLE correlation obtained via*

$$\{(R_0, R_1) \,|\, (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCG.Gen}(1^\lambda), \\ R_b \leftarrow \mathsf{PCG.Expand}(b, k_b) \text{ for } b \in \{0,1\}\}$$

is computationally indistinguishable from $\mathcal{G}(1^\lambda)$.

- **Security.** *For any $b \in \{0,1\}$, the following two distributions are computationally indistinguishable*

$$\{(\mathsf{k}_b, R_{1-b}) \,|\, (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCG.Gen}(1^\lambda), R_{1-b} \leftarrow \mathsf{PCG.Expand}(1-b, \mathsf{k}_{1-b})\} \text{ and}$$
$$\{(\mathsf{k}_b, R_{1-b}) \,|\, (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCG.Gen}(1^\lambda), R_b \leftarrow \mathsf{PCG.Expand}(b, \mathsf{k}_b), R_{1-b} \leftarrow \mathsf{RSample}(b, R_b)\}$$

Using the recent PCG constructions [13, 72, 81, 12, 47, 69], the PCG scheme for random VOLE correlations can be constructed with the seed size *sublinear* to the length ℓ of vectors, i.e., $|\mathsf{k}_0| = O_\lambda(\log^2 \ell)$ and $|\mathsf{k}_1| = O_\lambda(\log \ell)$. The computational complexity of $\mathsf{PCG.Gen}(1^\lambda)$ is $O_\lambda(\log^2 \ell)$, and that of $\mathsf{PCG.Expand}(b, \mathsf{k}_b)$ is $O_\lambda(\ell)$. According to the known implementations [72, 13, 81, 69], the computation of both $\mathsf{PCG.Gen}(1^\lambda)$ and $\mathsf{PCG.Expand}(b, \mathsf{k}_b)$ is streamable and concretely efficient. The PCG as described above works for the two-party case. In the multi-party setting, every pair of parties $(\mathcal{P}_i, \mathcal{P}_j)$ for $i, j \in [1,n], i \neq j$ would generate a VOLE correlation such that every party $\mathcal{P}_i$ obtains the same $\boldsymbol{u}^i \in \mathbb{F}^\ell$ and $\Delta^i \in \mathbb{F}$ among all VOLE correlations. Note that the existing PCG schemes w.r.t. VOLE satisfy *programmability* defined in [16], i.e., $\mathsf{PCG.Gen}$ takes additional inputs $(\boldsymbol{u}^i, \Delta^j)$ and outputs a pair of seeds that are expanded to a VOLE correlation with fixed $\boldsymbol{u}^i, \Delta^j$. Based on the programmability, we are able to construct PCG for VOLE in the multi-party setting (see [16] for details). Building upon multi-party PCG for VOLE, we show the construction of a multi-party PCG scheme $(\mathsf{PCG.Gen}_{\mathsf{spdz}}, \mathsf{PCG.Expand}_{\mathsf{spdz}})$ for generating a vector of SPDZ-like authenticated sharings, while guaranteeing the security.

- $\mathsf{PCG.Gen}_{\mathsf{spdz}}(1^\lambda)$ runs $n(n-1)$ executions of $\mathsf{PCG.Gen}(1^\lambda)$ to generate $(\mathsf{k}_1, \ldots, \mathsf{k}_n)$ such that the size of every seed k_i is $O_\lambda((n-1)\log^2 \ell)$. In particular, $\mathsf{PCG.Gen}_{\mathsf{spdz}}(1^\lambda)$ executes as follows:

  1. For each $i \in [1,n]$, sample $\boldsymbol{u}^i \leftarrow \mathbb{F}^\ell$ and $\Delta^i \leftarrow \mathbb{F}$. Optionally, compute $\boldsymbol{u} := \sum_{i\in[1,n]} \boldsymbol{u}^i$.
  2. For each $i, j \in [1,n]$ with $i \neq j$, run $\mathsf{PCG.Gen}(1^\lambda, \boldsymbol{u}^i, \Delta^j)$ to generate a pair of seeds $(\mathsf{k}_0^{(i,j)}, \mathsf{k}_1^{(i,j)})$.
  3. For each $i \in [1,n]$, output $\mathsf{k}_i = \left(\{\mathsf{k}_0^{(i,j)}\}_{j\neq i}, \{\mathsf{k}_1^{(j,i)}\}_{j\neq i}\right)$. Optionally, output $\boldsymbol{u}$.

- $\mathsf{PCG.Expand}_{\mathsf{spdz}}(i, \mathsf{k}_i)$ runs $2(n-1)$ executions of $\mathsf{PCG.Expand}$ to generate $\mathcal{P}_i$'shares on a vector of SPDZ-like authenticated sharings $[\![\boldsymbol{u}]\!]$. For each $i \in [1,n]$, $\mathsf{PCG.Expand}_{\mathsf{spdz}}(i, \mathsf{k}_i)$ performs the following:

  1. For each $j \neq i$, run $\mathsf{PCG.Expand}\left(0, \mathsf{k}_0^{(i,j)}\right)$ to generate $R_0^{(i,j)} = \left(\boldsymbol{u}^i, \boldsymbol{v}^{(i,j)}\right)$ such that $\boldsymbol{w}^{(i,j)} = \boldsymbol{v}^{(i,j)} + \boldsymbol{u}^i \cdot \Delta^j \in \mathbb{F}^\ell$, and $\mathsf{PCG.Expand}\left(1, \mathsf{k}_1^{(j,i)}\right)$ to generate $R_1^{(j,i)} = \left(\boldsymbol{w}^{(j,i)}, \Delta^i\right)$ such that $\boldsymbol{w}^{(j,i)} = \boldsymbol{v}^{(j,i)} + \boldsymbol{u}^j \cdot \Delta^i \in \mathbb{F}^\ell$.
  2. Compute $\mathbf{M}^i := \boldsymbol{u}^i \cdot \Delta^i - \sum_{j\neq i} \boldsymbol{v}^{(i,j)} + \sum_{j\neq i} \boldsymbol{w}^{(j,i)} \in \mathbb{F}^\ell$.

Through the programmability, we guarantee that $\boldsymbol{u}^i$ and Δ^j are reused among all VOLE correlations associated with i or j, but $\boldsymbol{v}^{(i,j)}$ and $\boldsymbol{w}^{(i,j)}$ are independent for each pair (i,j). For correctness, we have

$$\begin{aligned}\sum_{i\in[1,n]} \mathbf{M}^i &= \sum_{i\in[1,n]} \boldsymbol{u}^i \cdot \Delta^i + \sum_{i\in[1,n]} \sum_{j\in[1,n], j\neq i} \left(\boldsymbol{w}^{(i,j)} - \boldsymbol{v}^{(i,j)}\right), \\ &= \sum_{i\in[1,n]} \boldsymbol{u}^i \cdot \Delta^i + \sum_{i\in[1,n]} \sum_{j\in[1,n], j\neq i} \boldsymbol{u}^i \cdot \Delta^j = \boldsymbol{u} \cdot \Delta \in \mathbb{F}^\ell,\end{aligned}$$

where $\sum_{i\in[1,n]} \Delta^i = \Delta \in \mathbb{F}$.

G Building blocks for private delegation from additive secret sharings

In the following protocol, we describe the private-delegation protocol in the dishonest-majority setting using additive secret sharings equipped with SPDZ-like IT-MACs. In $\Pi^{\mathsf{authss}}_{\mathsf{PrivDeleg}}$, we describe the private-delegation protocol in the dishonest-majority setting using additive secret sharings equipped with SPDZ-like IT-MACs. For two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, we use $\boldsymbol{a} \odot \boldsymbol{b}$ to denote their inner product. This protocol invokes SPDZ algorithms to compress the communication of generating IT-MACs to the sublinear complexity, where the algorithms can be constructed using the recent PCG schemes such as [81, 12]. All BatchCheck procedures can be deferred to the end of protocol execution, but must be performed before verifying the resulting proof. This is similar to the SPDZ-like MPC protocols [30, 29]. We include the formal security proof in Appendix J.

Prot. 9: AddSS-based Building Blocks

Let $m = \lceil \ell / N_w \rceil$.

- $\mathsf{MSM}([\![\boldsymbol{y}]\!], \boldsymbol{X}) \to [\![Z]\!]$: Given a vector of authenticated secret sharings $[\![\boldsymbol{y}]\!]$ and public group elements $\boldsymbol{X} \in \mathbb{F}^\ell$, for each $j \in \{0,1\}$, $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel as follows:

$$[\![Z]\!]_j^{(k)} := \sum_{i=(k-1)m+1}^{km} [\![\boldsymbol{y}_i]\!]_j^{(k)} \cdot X_i.$$

  For each $j \in \{0,1\}$, $\mathcal{P}_j$ chooses one node to compute $[\![Z]\!]_j := \sum_{k=1}^{N_w} [\![Z]\!]_j^{(k)}$, and then all parties output an authenticated secret sharing $[\![Z]\!]$.

- $\mathsf{LinearComb}([\![\boldsymbol{x}]\!], \boldsymbol{c}) \to [\![y]\!]$: Given a vector of authenticated secret sharings $[\![\boldsymbol{x}]\!]$ and public elements $\boldsymbol{c} \in \mathbb{F}^{\ell+1}$, for each $j \in \{0,1\}$, $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in paral-

lel:

$$[\![y]\!]_j^{(k)} := \sum_{i=(k-1)m+1}^{km} \boldsymbol{c}_i \cdot [\![\boldsymbol{x}_i]\!]_j^{(k)}.$$

For each $j \in \{0,1\}$, $\mathcal{P}_j$ chooses one node to compute $[\![y]\!]_j := \sum_{k=1}^{N_w} [\![y]\!]_j^{(k)} + [\![\boldsymbol{c}_0]\!]_j$, where $[\![\boldsymbol{c}_0]\!]$ is locally computed from the public element $\boldsymbol{c}_0$. Then, all parties output $[\![y]\!]$.

- $\mathsf{InnerProd}([\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]) \to \langle z \rangle$: Given two vectors of authenticated secret sharings $[\![\boldsymbol{x}]\!] = (\langle x \rangle, \langle \Delta \rangle, \langle x \cdot \Delta \rangle)$ and $[\![\boldsymbol{y}]\!] = (\langle y \rangle, \langle \Delta \rangle, \langle y \cdot \Delta \rangle)$, the delegator $\mathcal{D}$ and two parties $\mathcal{P}_0, \mathcal{P}_1$ execute as follows:

  1. In the preprocessing phase, $\mathcal{D}$ runs $\mathsf{PCG.Gen}(1^\lambda)$ to generate a pair of short keys $(\mathsf{k}_0, \mathsf{k}_1)$, and then sends k_j to $\mathcal{P}_j$ for each $j \in \{0,1\}$. Then, for each $j \in \{0,1\}$, $\mathcal{P}_j$ runs $\mathsf{PCG.Expand}(j, \mathsf{k}_j)$ to generate $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$ such that $[\![\boldsymbol{a}]\!] = (\langle a \rangle, \langle \Delta \rangle, \langle a \cdot \Delta \rangle)$ and $[\![\boldsymbol{b}]\!] = (\langle b \rangle, \langle \Delta \rangle, \langle b \cdot \Delta \rangle)$. Here, the computation of running $\mathsf{PCG.Expand}$ is distributed among all the N_w nodes of each party $\mathcal{P}_j$.
  2. In the preprocessing phase, both parties run $\mathsf{Rec}(\langle \boldsymbol{a} \rangle, \mathcal{D})$ and $\mathsf{Rec}(\langle \boldsymbol{b} \rangle, \mathcal{D})$ to let $\mathcal{D}$ obtain $\boldsymbol{a} \in \mathbb{F}^\ell$ and $\boldsymbol{b} \in \mathbb{F}^\ell$. Then, $\mathcal{D}$ computes $c := \boldsymbol{a} \odot \boldsymbol{b} \in \mathbb{F}$, and runs $\mathsf{Share}(c)$ to let the parties obtain $\langle c \rangle$.
  3. Both parties run $\mathsf{Open}(\langle \boldsymbol{x} \rangle - \langle \boldsymbol{a} \rangle)$ to obtain $\boldsymbol{g} \in \mathbb{F}^\ell$ and $\mathsf{Open}(\langle \boldsymbol{y} \rangle - \langle \boldsymbol{b} \rangle)$ to get $\boldsymbol{h} \in \mathbb{F}^\ell$. During the Open procedure, the computation of shares on $\langle \boldsymbol{g} \rangle = \langle \boldsymbol{x} \rangle - \langle \boldsymbol{a} \rangle$ and $\langle \boldsymbol{h} \rangle = \langle \boldsymbol{y} \rangle - \langle \boldsymbol{b} \rangle$ is distributed as follows:
     - For each $j \in \{0,1\}$, $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute $\langle \boldsymbol{g} \rangle_j^{(k)} = \langle \boldsymbol{x} \rangle_j^{(k)} - \langle \boldsymbol{a} \rangle_j^{(k)}$.
     - For each $j \in \{0,1\}$, $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute $\langle \boldsymbol{h} \rangle_j^{(k)} = \langle \boldsymbol{y} \rangle_j^{(k)} - \langle \boldsymbol{b} \rangle_j^{(k)}$.

     For each $j \in \{0,1\}$, $\mathcal{P}_j$ divides $\boldsymbol{g}$ and $\boldsymbol{h}$ into N_w parts, and each node $\mathcal{E}_j^{(k)}$ holds $\boldsymbol{g}^{(k)}$ and $\boldsymbol{h}^{(k)}$.
  4. Both parties locally compute $\langle z \rangle := \boldsymbol{g} \odot \boldsymbol{h} + \boldsymbol{g} \odot \langle \boldsymbol{b} \rangle + \boldsymbol{h} \odot \langle \boldsymbol{a} \rangle + \langle c \rangle$ as follows:
     (a) For $j \in \{0,1\}$, $\mathcal{P}_j$ makes each node $\mathcal{E}_j^{(k)}$ compute

     $$\langle z \rangle_j^{(k)} := \boldsymbol{g}^{(k)} \odot \boldsymbol{h}^{(k)} + \boldsymbol{g}^{(k)} \odot \langle \boldsymbol{b} \rangle_j^{(k)} + \boldsymbol{h}^{(k)} \odot \langle \boldsymbol{a} \rangle_j^{(k)}.$$

     (b) For $j \in \{0,1\}$, $\mathcal{P}_j$ chooses one node to compute

     $$\langle z \rangle_j := \sum_{k=1}^{N_w} \langle z \rangle_j^{(k)} + \langle c \rangle_j.$$

  5. $\mathcal{P}_0$ and $\mathcal{P}_1$ execute the procedure $\mathsf{BatchCheck}((([\![\boldsymbol{x}]\!] - [\![\boldsymbol{a}]\!], [\![\boldsymbol{y}]\!] - [\![\boldsymbol{b}]\!]), (\boldsymbol{g}, \boldsymbol{h}))$ to check that $\boldsymbol{g} = \boldsymbol{x} - \boldsymbol{a} \in \mathbb{F}^\ell$ and $\boldsymbol{h} = \boldsymbol{y} - \boldsymbol{b} \in \mathbb{F}^\ell$. Both parties abort if the check fails. Recall that the distributed computation for the $\mathsf{BatchCheck}$ procedure among all nodes of every party has already described in Appendix F.2.

- $\mathsf{Fold}([\![\boldsymbol{x}]\!], r) \to [\![\boldsymbol{y}]\!]$: Given a vector of authenticated secret sharings $[\![\boldsymbol{x}]\!]$ and a public element $r \in \mathbb{F}$, let $m' = \lceil \ell / 2N_w \rceil$, for each $j \in \{0,1\}$, $\mathcal{P}_j$ controls each node $\mathcal{E}_j^{(k)}$ to compute in parallel as follows:

  $$[\![\boldsymbol{y}_i]\!]_j^{(k)} := \sum_{i=(k-1)m'+1}^{km'} [\![\boldsymbol{x}_{2i-1}]\!]_j^{(k)} + r \cdot [\![\boldsymbol{x}_{2i}]\!]_j^{(k)} \text{ for } i \in [1, \ell/2].$$

  Then, both parties output $[\![\boldsymbol{y}]\!]$ by collecting the shares from all the nodes.

H Detailed protocol for DFS with private delegation

In this section, we describe the private-delegation protocol for DFS, where a delegator $\mathcal{D}$ delegates the generation of a zkSNARK proof to N_p untrusted parties, and $\mathcal{D}$ will obtain the proof at the end of protocol execution. Note that the delegator is trusted and will perform the logarithmic number of operations to accelerate the proof generation. Recall that every party controls N_w nodes. For convenience, we denote $n_p := \log(N_p)$ and $n_w := \log(N_w)$.

This private-delegation protocol invokes the algorithms and procedures shown in Section 5.1 to securely compute the operations implied in the proving algorithm. Whenever we have an n-variate polynomial $p(\boldsymbol{x})$, if it is a private polynomial, then all parties hold a secret sharing $[\![p(\boldsymbol{x})]\!]$, where each party $\mathcal{P}_j$ holds a share $[\![p(\boldsymbol{x})]\!]_j$. In particular, the sharing $[\![p(\boldsymbol{x})]\!]$ consists of the secret sharings on the evaluations of polynomial $p(\boldsymbol{x})$ at different points, i.e., $[\![p(\boldsymbol{x})]\!]$ is a vector of linear secret sharings. Therefore, both evaluation and interpolation of private polynomial $p(\boldsymbol{x})$ are corresponding to the linear-combination operation of a private vector and another public vector. Given a vector of linear secret sharings $[\![p(\boldsymbol{x})]\!]$, all parties can run the $\mathsf{LinearComb}$ algorithm defined in Section 5.1 to compute secret sharings on the evaluation or interpolation of polynomial $p(\boldsymbol{x})$ without any communication. The computation of every party can be accelerated using the worker nodes to perform the computation in a distributed and parallel way.

Initialization. In the initialization phase, in addition to processing the index information for the zkSNARK itself, we also need to generate and distribute the witness shares to each party. Each party will then distribute its share of the witness, along with the non-zero elements of the matrix, across its local nodes for distributed computation. In the dishonest-majority scenario, Beaver triples must be prepared for each party to securely handle the multiplications involved in the inner-product computations. However, there is not needed for Beaver triples in the honest-majority scenario, since no MPC communication is required between parties during the computation of the inner-product. We additionally distribute the computation trace Az, Bz and Cz before the proof generation.

Private delegation of sumcheck. In the sub-protocol, the delegator will open the sumcheck messages (see step 2 of

PIOP 1) to all parties. Our zkSNARK scheme DFS needs to handle two cases of sumchecks:

- Case 1: private data is multiplied (step 2 of PIOP 4);
- Case 2: public data is multiplied by private data (step 7 of PIOP 4).

In the case of purely public data (step 9 of PIOP 4), since it does not involve any private information, MPC is not required, and we can simply invoke the public-delegation protocol in Appendix E. Here, we focus on describing the protocols for the first two sumcheck cases that involve private data.

We first present the first case (step 2 of PIOP 4), which involves the product of three polynomials: $f(\boldsymbol{x})$, $g(\boldsymbol{x})$ and $h(\boldsymbol{x})$ such that $\sum_{\boldsymbol{b}\in\{0,1\}^n} f(\boldsymbol{b})\cdot g(\boldsymbol{b})\cdot h(\boldsymbol{b}) = \sigma$. In this case, $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are private polynomials, while $h(\boldsymbol{x})$ and σ are public. For convenience of presentation, we assume that $h(\boldsymbol{x})$ has been distributed among the parties and their nodes in the same way as $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$. All parties hold two vectors of secret sharings $[\![f(\boldsymbol{x})]\!]$ and $[\![g(\boldsymbol{x})]\!]$ along with a public $h(\boldsymbol{x})$.

Prot. 10:
PRIVATE DELEGATION OF SUMCHECK FOR CASE 1

1. $\mathcal{D}$ samples a random masking polynomial $\tau(\boldsymbol{x})$ (which has only n non-zero entries), and outputs its commitment and the sumcheck claim $\sigma_\tau = \sum_{\boldsymbol{b}\in\{0,1\}^n}\tau(\boldsymbol{b})$.
2. $\mathcal{D}$ performs the Fiat-Shamir transform to generate a random challenge $\rho\in\mathbb{F}$, and then sends it to all parties. (Below, the protocol will produce the sumcheck messages for checking $\sum_{\boldsymbol{b}\in\{0,1\}^n} f(\boldsymbol{b})\cdot g(\boldsymbol{b})\cdot h(\boldsymbol{b}) + \rho\cdot\tau(\boldsymbol{b}) = \sigma + \rho\cdot\sigma_\tau$).
3. For each $i\in[1,\ldots,n]$, $\mathcal{D}$ and all parties $\mathcal{P}_1,\ldots,\mathcal{P}_{N_p}$ execute as follows:
   (a) For each $X_i\in\{0,1,2,3\}$, the parties run the LinearComb algorithm to compute the following two vectors of secret sharings:
   $$[\![\boldsymbol{f}(X_i)]\!] := [\![f(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)]\!]$$
   $$[\![\boldsymbol{g}(X_i)]\!] := [\![g(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)]\!]\cdot h(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n),$$
   where $(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}$ are enumerated to form two vectors $\boldsymbol{f}(X_i)\in\mathbb{F}^{2^{n-i}}$ and $\boldsymbol{g}(X_i)\in\mathbb{F}^{2^{n-i}}$.
   (b) For each $X_i\in\{0,1,2,3\}$, all parties execute the inner-product procedure
   $$\mathsf{InnerProd}([\![\boldsymbol{f}(X_i)]\!],[\![\boldsymbol{g}(X_i)]\!])\to\langle z(X_i)\rangle.$$
   (c) For each $X_i\in\{0,1,2,3\}$, the parties run the procedure $\mathsf{Rec}(\langle z(X_i)\rangle,\mathcal{D})$ to let $\mathcal{D}$ obtain $z(X_i)$. It is easy to see that $z(X_i) =$
   $$\sum_{(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}} (f\cdot g\cdot h)(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n).$$
   Then, $\mathcal{D}$ outputs $z(X_i) + \sum_{(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}}\rho\cdot\tau(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)$.
   (d) $\mathcal{D}$ performs the Fiat-Shamir transform to generate a random point $r_i\in\mathbb{F}$, and then sends it to all parties.
   (e) The parties run the folding algorithm $\mathsf{Fold}([\![f(\boldsymbol{x})]\!],r_i)$ to obtain $[\![f(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})]\!]$, and update $[\![f(\boldsymbol{x})]\!]$ as $[\![f(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})]\!]$. Then, they run $\mathsf{Fold}([\![g(\boldsymbol{x})]\!],r_i)$ to update $[\![g(\boldsymbol{x})]\!]$ as $[\![g(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})]\!]$. All parties also perform the folding operation with the inputs of public polynomial $h(\boldsymbol{x})$ and r_i to update $h(\boldsymbol{x})$ as $h(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})$.
4. $\mathcal{D}$ also outputs an opening proof for $\tau(\boldsymbol{r})$ where $\boldsymbol{r} = (r_1,\ldots,r_n)$.

We now present the second case (step 7 of PIOP 4), which involves the product of two polynomials: $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ such that $\sum_{\boldsymbol{b}\in\{0,1\}^n} f(\boldsymbol{b})\cdot g(\boldsymbol{b}) = \sigma$. In this case, $f(\boldsymbol{x})$ is a private polynomial, while $g(\boldsymbol{x})$ and σ are public. To simply the description, we assume that $g(\boldsymbol{x})$ has been distributed among the parties and their nodes (i.e., each party holds a copy of $g(\boldsymbol{x})$). This case is simple, since it does not involve multiplication of private data. All parties hold a vector of secret sharings $[\![f(\boldsymbol{x})]\!]$ as well as a public polynomial $g(\boldsymbol{x})$.

Prot. 11:
PRIVATE DELEGATION OF SUMCHECK FOR CASE 2

1. $\mathcal{D}$ samples a random masking polynomial $\tau(\boldsymbol{x})$ (which has only n non-zero entries), and outputs its commitment and the sumcheck claim $\sigma_\tau = \sum_{\boldsymbol{b}\in\{0,1\}^n}\tau(\boldsymbol{b})$.
2. $\mathcal{D}$ performs the Fiat-Shamir transform to generate a random challenge $\rho\in\mathbb{F}$, and then sends it to all parties. (Below, the protocol will produce the sumcheck messages for checking $\sum_{\boldsymbol{b}\in\{0,1\}^n} f(\boldsymbol{b})\cdot g(\boldsymbol{b}) + \rho\cdot\tau(\boldsymbol{b}) = \sigma + \rho\cdot\sigma_\tau$.)
3. For each $i\in[1,\ldots,n]$, $\mathcal{D}$ and all parties $\mathcal{P}_1,\ldots,\mathcal{P}_{N_p}$ execute as follows:
   (a) For each $X_i\in\{0,1,2\}$, the parties run the LinearComb algorithm to compute the secret sharing $[\![z(X_i)]\!] :=$
   $$\sum_{(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}} [\![f(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)]\!]\cdot g(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)$$
   (b) For each $X_i\in\{0,1,2\}$, all parties run the reconstruction procedure $\mathsf{Rec}([\![z(X_i)]\!],\mathcal{D})$ to let $\mathcal{D}$ obtain $z(X_i)$, such that $z(X_i) =$
   $$\sum_{(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}} (f\cdot g)(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n).$$
   Then, $\mathcal{D}$ outputs $z(X_i) + \sum_{(b_{i+1},\ldots,b_n)\in\{0,1\}^{n-i}}\rho\cdot\tau(r_1,\ldots,r_{i-1},X_i,b_{i+1},\ldots,b_n)$.
   (c) $\mathcal{D}$ performs the Fiat-Shamir transform to generate a random point $r_i\in\mathbb{F}$, and then sends it to all parties.
   (d) All parties run the folding algorithm $\mathsf{Fold}([\![f(\boldsymbol{x})]\!],r_i)$ to obtain $[\![f(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})]\!]$, and update $[\![f(\boldsymbol{x})]\!]$ as $[\![f(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})]\!]$. The parties also perform the folding operation with the inputs of public polynomial $g(\boldsymbol{x})$ and r_i to update $g(\boldsymbol{x})$ as $g(r_1,\ldots,r_{i-1},r_i,\boldsymbol{x})$.

4. $\mathcal{D}$ outputs an opening proof for $\tau(\boldsymbol{r})$ with $\boldsymbol{r} = (r_1, \ldots, r_n)$.

Private delegation of zerocheck. We build a private-delegation protocol for zerocheck (step 2 of PIOP 4), based on the private-delegation protocol of sumcheck (Prot. 10). All parties hold three vectors of linear secret sharings $[\![f(\boldsymbol{x})]\!]$, $[\![g(\boldsymbol{x})]\!]$ and $[\![h(\boldsymbol{x})]\!]$, such that $f(\boldsymbol{b}) \cdot g(\boldsymbol{b}) + h(\boldsymbol{b}) = 0$ for all $\boldsymbol{b} \in \{0,1\}^n$.

Prot. 12: Private Delegation of Zerocheck

1. $\mathcal{D}$ performs the Fiat-Shamir transform to generate a random vector $\boldsymbol{r} \in \mathbb{F}^n$, and then sends it to all parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$.
2. Polynomial $\mathrm{eq}(\boldsymbol{x}, \boldsymbol{r})$ (defined in Section 3) can be divided into N_w parts, where each part includes $2^n/N_w$ terms. Each node $\mathcal{E}_j^{(k)}$ of every party $\mathcal{P}_j$ computes the k-th part. Party $\mathcal{P}_j$ can collect all parts to obtain $\mathrm{eq}(\boldsymbol{x}, \boldsymbol{r})$.
3. $\mathcal{D}$ and all parties execute the private-delegation protocol of sumcheck shown in Prot. 10 (resp., Prot. 11) to generate the sumcheck messages for checking $\sum_{\boldsymbol{b} \in \{0,1\}^n} f(\boldsymbol{b}) \cdot g(\boldsymbol{b}) \cdot eq(\boldsymbol{b}) = 0$ (resp., $\sum_{\boldsymbol{b} \in \{0,1\}^n} h(\boldsymbol{b}) \cdot eq(\boldsymbol{b}) = 0$), where f, g and h are private polynomials.

Private delegation of polynomial evaluation. All parties hold linear secret sharings $[\![p(\boldsymbol{x})]\!]$ on a private polynomial $p(\boldsymbol{x})$ along with a public evaluation point $\boldsymbol{z} \in \mathbb{F}^n$. Delegator, who interacts with the parties, aims to output $p(\boldsymbol{z}) \in \mathbb{F}$.

Prot. 13: Private Delegation of Polynomial Evaluation

1. All parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ run the $\mathsf{LinearComb}$ and Fold algorithms to compute secret sharings $[\![p(\boldsymbol{z})]\!]$ for a private polynomial $p(\boldsymbol{x})$ and a public evaluation point $\boldsymbol{z}$.
2. The parties run the reconstruction procedure $\mathsf{Rec}([\![p^*]\!], \mathcal{D})$ to make $\mathcal{D}$ obtain a polynomial $p^*(\boldsymbol{x})$. Then, $\mathcal{D}$ computes and outputs the polynomial evaluation $p^*(\boldsymbol{z}^*) \in \mathbb{F}$.

Private delegation of polynomial commitments. The private-delegation protocol of the Commit and Open algorithms on a polynomial commitment scheme is simple, since all operations are linear. In the commitment phase, all parties input a vector of secret sharings $[\![p(\boldsymbol{x})]\!]$ and a pair of public committed keys $(\mathsf{ck}, \mathsf{ck}^*)$; the delegator $\mathcal{D}$ holds $(\mathsf{ck}, \mathsf{ck}^*)$ and outputs a commitment on the private polynomial $p(\boldsymbol{x})$. In the open phase, the parties have $[\![p(\boldsymbol{x})]\!]$ and a public evaluation point $\boldsymbol{z}$, and $\mathcal{D}$ holds a masking polynomial $\tau(\boldsymbol{x})$ and another public evaluation point $\boldsymbol{z}^*$; $\mathcal{D}$ would output the opening proof on $p(\boldsymbol{z})$.

Prot. 14: Private Delegation of Polynomial Commitments

$\mathsf{PST.Commit}(\mathsf{ck}, [\![p(\boldsymbol{x})]\!]) \to \mathsf{cm}$

1. All parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ run the $\mathsf{MSM}([\![p(\boldsymbol{x})]\!], \mathsf{ck})$ algorithm to obtain a secret sharing $[\![\mathsf{cm}_0]\!]$.
2. The parties execute the $\mathsf{Rec}([\![\mathsf{cm}_0]\!], \mathcal{D})$ procedure to let $\mathcal{D}$ obtain cm_0.
3. $\mathcal{D}$ samples a random masking polynomial $\tau(\boldsymbol{x})$, and produces a commitment $\mathsf{cm}_1 = \mathsf{MSM}(\tau(\boldsymbol{x}), \mathsf{ck})$.
4. $\mathcal{D}$ computes and outputs $\mathsf{cm} = \mathsf{cm}_0 + \mathsf{cm}_1$.

$\mathsf{PST.Open}((\mathsf{ck}, \mathsf{ck}^*), ([\![p(\boldsymbol{x})]\!], \tau(\boldsymbol{x})), (\boldsymbol{z}, \boldsymbol{z}^*)) \to \pi_{\mathsf{PC}}$

1. All parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ run the $\mathsf{LinearComb}$ and Fold algorithms to compute $[\![y]\!]$ with $y = p(\boldsymbol{z})$.
2. For each $i \in [1 \ldots n]$, the parties run the $\mathsf{LinearComb}$ and Fold algorithms to compute the i-th witness polynomial $[\![q_i(\boldsymbol{x})]\!]$, where all n witness polynomials satisfy $p(\boldsymbol{x}) - y = \sum_{i=1}^n q_i(\boldsymbol{x}) \cdot (\boldsymbol{x}_i - \boldsymbol{z}_i)$.
3. For each $i \in [1 \ldots n]$, all parties run $\mathsf{MSM}([\![q_i(\boldsymbol{x})]\!], \mathsf{ck})$ to compute a secret sharing $[\![\pi_i]\!]$.
4. For each $i \in [1 \ldots n]$, the parties run the $\mathsf{Rec}([\![\pi_i]\!], \mathcal{D})$ procedure to let $\mathcal{D}$ obtain π_i. Then, $\mathcal{D}$ sets $\pi^0_{\mathsf{PC}} = (\pi_1, \ldots, \pi_n)$.
5. $\mathcal{D}$ generates the opening proof on the masking polynomial by running $\pi^1_{\mathsf{PC}} \leftarrow \mathsf{PST.Open}(\mathsf{ck}^*, \tau(\boldsymbol{x}), \boldsymbol{z}^*)$.
6. $\mathcal{D}$ outputs the opening proof $\pi_{\mathsf{PC}} := \pi^0_{\mathsf{PC}} + \pi^1_{\mathsf{PC}}$.

Private delegation of DFS. Using the sub-protocols described as above, we design an efficient private-delegation protocol for DFS, which is shown as below. Without loss of generality, we assume that the CRS and preprocessing keys (ipk, ivk) have been produced, which are omitted for simplicity in the description of this protocol. Delegator $\mathcal{D}$ employs all parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ (where each party holds N_w nodes) to generate a proof that proves validity of $\boldsymbol{z}_A = A \cdot \boldsymbol{z}$, $\boldsymbol{z}_B = B \cdot \boldsymbol{z}$, $\boldsymbol{z}_C = C \cdot \boldsymbol{z}$ and $\boldsymbol{z}_A \circ \boldsymbol{z}_B = \boldsymbol{z}_C$. Following prior works [25, 64, 39, 59] about private delegation of zkSNARKs, we also assume that $\mathcal{D}$ has run the Share procedure to make the parties obtain $[\![\boldsymbol{z}]\!]$, $[\![\boldsymbol{z}_A]\!]$, $[\![\boldsymbol{z}_B]\!]$ and $[\![\boldsymbol{z}_C]\!]$ in the setup phase. The communication of the setup phase can be further reduced by letting $\mathcal{D}$ only share $\boldsymbol{z}$ and all parties compute $[\![\boldsymbol{z}_A]\!]$, $[\![\boldsymbol{z}_B]\!]$ and $[\![\boldsymbol{z}_C]\!]$ by running the $\mathsf{LinearComb}$ algorithm. At the end of protocol execution, $\mathcal{D}$ outputs a proof that consists of the outputs of sub-protocol executions.

Prot. 15: Private Delegation of DFS

Setup: All parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ hold secret sharings $[\![\boldsymbol{z}]\!]$, $[\![\boldsymbol{z}_A]\!]$, $[\![\boldsymbol{z}_B]\!]$ and $[\![\boldsymbol{z}_C]\!]$. Recall that $\hat{A}$, $\hat{B}$, and $\hat{C}$ are the multilinear extensions of $\boldsymbol{z}_A$, $\boldsymbol{z}_B$ and $\boldsymbol{z}_C$, respectively. Let $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ be the matrix encodings of matrices A, B, C. For each $M \in \{A, B, C\}$, the sparse matrix encodings $\boldsymbol{r}_M$, $\boldsymbol{c}_M$, and $\boldsymbol{v}_M$ are splited into N_w parts, which allow the nodes of every party to perform distributed computation. Let s be the logarithm of the number of constraints.

Protocol:

1. $\mathcal{D}$ and all parties $\mathcal{P}_1, \ldots, \mathcal{P}_{N_p}$ execute the privately delegated polynomial commitment sub-protocol (Prot. 14) to let $\mathcal{D}$ output a commitment on the private polynomial $\boldsymbol{z}(\boldsymbol{x})$.
2. Let $\boldsymbol{F}(\boldsymbol{x}) := \hat{A}(\boldsymbol{x}) \cdot \hat{B}(\boldsymbol{x}) - \hat{C}(\boldsymbol{x})$. $\mathcal{D}$ and the parties execute the privately delegated zerocheck sub-protocol (Prot. 12) on the input of polynomial $\boldsymbol{F}$. At the end of the sub-protocol execution, $\mathcal{D}$ outputs the sumcheck messages for an evaluation claim of the form $\boldsymbol{F}(\boldsymbol{\rho_x})\text{eq}(\boldsymbol{\rho_x}, \boldsymbol{r}) = e_x$ for a vector of random points $\boldsymbol{\rho_x} \in \mathbb{F}^s$ as well as a zerocheck challenge $\boldsymbol{r} \in \mathbb{F}^s$.
3. For each $M \in \{A, B, C\}$, $\mathcal{D}$ and the parties execute the privately delegated polynomial evaluation sub-protocol (Prot. 13) to output $v_M = M(\boldsymbol{\rho_x})$.
4. $\mathcal{D}$ performs the Fiat-Shamir transform to generate three random challenges $r_A, r_B, r_C \in \mathbb{F}$, and then sends them to the parties.
5. Let $M_{\boldsymbol{\rho_x}}(\boldsymbol{y}) := (r_A \cdot \boldsymbol{A}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_B \cdot \boldsymbol{B}(\boldsymbol{\rho_x}, \boldsymbol{y}) + r_C \cdot \boldsymbol{C}(\boldsymbol{\rho_x}, \boldsymbol{y})) \cdot \boldsymbol{z}(\boldsymbol{y})$. $\mathcal{D}$ and all parties engage in a privately delegated sumcheck sub-protocol (Prot. 11) for the claim $\sum_{\boldsymbol{y} \in \{0,1\}^s} M_{\boldsymbol{\rho_x}}(\boldsymbol{y}) = r_A \cdot v_A + r_B \cdot v_B + r_C \cdot v_C$. Through the sub-protocol execution, $\mathcal{D}$ and the parties obtain a vector of random points $\boldsymbol{\rho_y} \in \mathbb{F}^s$.
6. $\mathcal{D}$ and the parties invoke the privately delegated polynomial commitment sub-protocol (Prot. 14) to output the opening proof of $\boldsymbol{z}(\boldsymbol{\rho_y})$.
7. For each $M \in \{A, B, C\}$, $\mathcal{D}$ and all the nodes from all parties execute the last step (i.e., the step 8) of the public-delegation sub-protocol Prot. 8.
8. $\mathcal{D}$ outputs a proof π by collecting the proof components from all sub-protocol executions. $\mathcal{D}$ invokes the verification algorithm of the zkSNARK scheme DFS to verify the validity of π, and aborts if the verification fails.

I USENIX Security '25 Artifact Appendix

I.1 Abstract

This artifact appendix provides a roadmap for evaluators to test the functionality of our implementation. The artifact is based on the Arkworks framework and offers both single-prover and distributed setups.

I.2 Description & Requirements

I.2.1 Security, privacy, and ethical concerns

We attest that we have thoroughly reviewed the ethics considerations as outlined in the conference call for papers, the detailed submission instructions, and the ethics guidelines document provided by the conference organizers. The research team has carefully evaluated the ethical implications of our work on DFS, ensuring that the research has been conducted in accordance with the highest ethical standards.

Our team has considered all potential ethical issues arising from this research, including the responsible disclosure of findings, the privacy implications of the technologies developed, and the potential for both positive and negative impacts on stakeholders. We have also proactively assessed the possible risks and mitigated them where necessary. We believe that our research was conducted ethically and in a manner that aligns with both the principles of beneficence and respect for persons as described in the Menlo Report.

Additionally, our next steps following publication have been carefully planned with ethical considerations in mind. We commit to following responsible procedures for the further dissemination and application of our findings, particularly in terms of sharing data and code in compliance with the conference's open science policy. We are prepared to engage with the broader community to address any ethical concerns that may arise as the research progresses.

Finally, we have also provided this additional Ethics Considerations and Compliance with the Open Science Policy section to ensure that all relevant ethical issues are transparent and addressed appropriately.

I.2.2 How to access

The artifact can be found in DOI 10.5281/zenodo.14677896.

I.2.3 Hardware dependencies

A local machine with the following specifications is sufficient for functionality testing:

- At least 10-core CPU
- 16 GB RAM

I.2.4 Software dependencies

- **Operating System:** Ubuntu 20.04+ (recommended) or MacOS
- **Rust Compiler:** nightly toolchain (v1.75+)
- **GNU Bash:** Required for running scripts

I.2.5 Benchmarks

None.

I.3 Set-up

I.3.1 Installation

Follow the instructions in the `README.md` file to set up the environment and dependencies. Note that we are using nightly toolchain for Rust.

I.3.2 Basic Test

To verify the basic functionality of the artifact, follow these steps:

Setup and Compilation.

```
cargo build --release --examples
```

Running Functional Tests. Execute the following commands:

```
cargo test --release
```

I.4 Evaluation workflow

I.4.1 Major Claims

(C1): For public delegations, DFS achieves logarithimic communication overheads.

(C2): For private delegations, DFS achieves logarithimic communication overheads for RSS-based implementations, and linear communication overheads for AddSS-based implementations.

I.4.2 Experiments

To run the experiments for example/rss_snark, example/ass_snark and example/snark, use the following commands:

```
./setup.sh
./work.sh
```

Before running the setup.sh, inst_folder folder must be created in the corresponding directory to store the generated files. The work.sh script will then run the tests.

mpirun $-$ n$*$ can be used to specify the number of cores to run. If it is changed, other parameters such as log-num-parties must also be changed accordingly.

I.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.

J Ideal functionalities and security proofs of our protocols

Discussion on multi-round Fiat-Shamir transform. In our private-delegation protocol shown in Figure 15, there are $O(\log m)$ rounds between the delegator $\mathcal{D}$ and parties for Fiat-Shamir transform, where m is the number of constraints in R1CS. In each round, the parties send a part of components (included in the proof) to $\mathcal{D}$, and then $\mathcal{D}$ sends the random Fiat-Shamir challenges to the parties. In the following, we will discuss that no selective-failure attack is allowed, even though the multi-round Fiat-Shamir transform is used. For our protocol, each sub-circuit has only one layer of multiplications, and the output is hashed for Fiat-Shamir transform. Therefore, any additive attack would lead to different Fiat-Shamir challenges, which makes the proof invalid with overwhelming probability. In addition, our protocol avoids cross-terms between adversarial-chosen errors and secret values, which further prevents any possible selective-failure attack.

J.1 Ideal functionality for building blocks

In this section, we define the ideal functionality to model the security of the protocols that only consist of these building blocks. Delegator $\mathcal{D}$ is able to obtain the *unopened* components in a proof by calling the output command, and can get the *opened* components in the proof from all parties after they execute the public-delegation phase. Here, to simplify the description, we refer the elements as "opened" components, even if they are computed from the values that have been opened in the public-delegation phase. This functionality only guarantees the privacy, and allows the adversary to introduce some errors into the values to be opened or output. Note that the delegator can verify the proof to assure the correctness, after it receives the whole proof.

Func. 1:
IDEAL FUNCTIONALITY FOR BUILDING BLOCKS

This functionality interacts with an honest delegator $\mathcal{D}$, the parties $\mathcal{P}_1,\ldots,\mathcal{P}_n$ and an adversary who corrupts at most t parties. Then, this functionality operates as follows:

- Upon receiving $(\mathsf{share},\mathsf{id},w)$ from $\mathcal{D}$ and $(\mathsf{share},\mathsf{id})$ from all parties (i.e., $\mathcal{P}_i$ for all $i\in[1\ldots n]$), where id is a fresh identifier and $w\in\mathbb{F}$, store (id,w).
- Upon receiving $(\mathsf{msm},\{\mathsf{id}_i\}_{i\in[1\ldots\ell]},\mathsf{id},\boldsymbol{X})$ from all parties, where $\boldsymbol{X}\in\mathbb{G}^\ell$ is public and id_i for all $i\in[1\ldots\ell]$ are present in memory, retrieve $(\mathsf{id}_i,\boldsymbol{y}_i)$ for each $i\in[1\ldots\ell]$. Then run $\mathsf{MSM}(\boldsymbol{y},\boldsymbol{X})\to Z$ and store (id,Z).
- Upon receiving $(\mathsf{linearcomb},\{\mathsf{id}_i\}_{i\in[1\ldots\ell]},\mathsf{id},\boldsymbol{c})$ from all parties, where $\boldsymbol{c}\in\mathbb{F}^{\ell+1}$ is public and id_i for all $i\in[1\ldots\ell]$ are present in memory, retrieve $(\mathsf{id}_i,\boldsymbol{x}_i)$ for each $i\in[1\ldots\ell]$. Then run $\mathsf{LinearComb}(\boldsymbol{x},\boldsymbol{c})\to y$ and store (id,y).
- Upon receiving $(\mathsf{innerprod},\{\mathsf{id}_i\}_{i\in[1\ldots\ell]},\{\mathsf{id}'_i\}_{i\in[1\ldots\ell]},\mathsf{id})$ from all parties, where id_i, and id'_i for all $i\in[1\ldots\ell]$ are present in memory, retrieve $(\mathsf{id}_i,\boldsymbol{x}_i)$ and $(\mathsf{id}'_i,\boldsymbol{y}_i)$ for each $i\in[1\ldots\ell]$. Then, run $\mathsf{InnerProd}(\boldsymbol{x},\boldsymbol{y})\to z$ and store (id,z).
- Upon receiving $(\mathsf{fold},\{\mathsf{id}_i\}_{i\in[1\ldots\ell]},\{\mathsf{id}'_i\}_{i\in[1\ldots\ell]},r)$ from all parties, where id_i for all $i\in[1\ldots\ell]$ are present in memory and $r\in\mathbb{F}$, retrieve $(\mathsf{id}_i,\boldsymbol{x}_i)$ for each $i\in[1\ldots\ell]$. Then run $\mathsf{Fold}(\boldsymbol{x},r)\to\boldsymbol{y}$, and store $(\mathsf{id}'_i,\boldsymbol{y}_i)$ for each $i\in[1\ldots\ell/2]$.
- Upon receiving $(\mathsf{rec},\mathsf{id})$ from all parties and $\mathcal{D}$, where id is present in memory, retrieve (id,z), wait for the adversary to input an error e, then send $z+e$ to $\mathcal{D}$. Ignore any subsequent $(\mathsf{open},\mathsf{id})$ commands.

J.2 Proof for RSS-based DFS

Theorem J.1. *RSS-based protocol (Prot. 1) securely realizes the ideal functionality defined in Func. 1 in the presence of a static, malicious adversary who corrupts at most one party.*

Proof. It is easy to construct a probabilistic polynomial time (PPT) simulator, who invokes a PPT adversary as a subroutine. Then, we analyze that the ideal-world execution is computationally indistinguishable from the real-world execution via a series of hybrids.

Hybrid 1 This is the real world execution.

Hybrid 2 In this hybrid, we replace the input sharing step with the simulation strategy. By the privacy property of the secret sharing scheme, this does not change the output distribution of the environment.

Hybrid 3 In this hybrid, we use the simulation strategy for the open and output commands to extract the error e from the adversary. This change is purely conceptual as the functionality allows the adversary to introduce errors. This is the ideal world execution.

□

J.3 Proof of AddSS-based DFS

Theorem J.2. *AddSS-based protocol (Prot. 9) securely realizes the ideal functionality defined in Func. 1 in the presence of a static, malicious adversary.*

Proof. It is easy to construct a PPT simulator, who invokes a PPT adversary as a subroutine. Then, we analyze that the ideal-world execution is computationally indistinguishable from the real-world execution via a series of hybrid games.

Hybrid 1 The real world execution with private input values.

Hybrid 2 In this hybrid, we replace the honest parties' output from PCG.Expand with uniform random values. By the security of PCG this action brings negligible change to the output behavior of the environment $\mathcal{Z}$.

Hybrid 3 In this hybrid, we replace the honest parties' checking in the BatchCheck step with the simulation strategy. By the property of the random oracle and that the honest parties' global keys are uniformly random, the statistical difference between previous hybrid is bounded by $(q_{\mathsf{H}}+3)/|\mathbb{F}|$.

Hybrid 4 In this hybrid, we extract the error e from $\mathcal{A}$ in the open and output commands and sends them to the ideal functionality. The change in this step is only conceptual.

Hybrid 5 In this hybrid, we replace the $\boldsymbol{d}$ values of the input command and $\boldsymbol{g},\boldsymbol{h}$ in the innerprod command with uniformly random values. Since the honest parties' secret masks are uniformly random, this step does not change the output distribution of $\mathcal{Z}$. This is the ideal world execution.

□