Siniel: Distributed Privacy-Preserving zkSNARK

Yunbo Yang[†‡§], Yuejia Cheng[¶], Kailun Wang[‖], Xiaoguo Li[**], Jianfei Sun[††], Jiachen Shen[†], Xiaolei Dong [†(✉)], Zhenfu Cao [†], Guomin Yang [††], and Robert H. Deng[††]
[†]Engineering Research Center of Blockchain Data Management, Ministry of Education, East China Normal University
[‡]The State Key Laboratory of Blockchain and Data Security, Zhejiang University
[§]Hangzhou High-Tech Zone (Bin jiang) Institute of Blockchain and Data Security
[¶]Shanghai DeCareer Consulting Co., Ltd
[‖]Beijing Jiaotong University
[**]College of Computer Science, Chongqing University
[††]School of Computing and Information Systems, Singapore Management University
52215902015@stu.ecnu.edu.cn, chengyuejia@foxmail.com, wangkailun@bjtu.edu.cn, csxgli@cqu.edu.cn,
jfsun@smu.edu.sg, {jcshen, dongxiaolei, zfcao}@sei.ecnu.edu.cn, {gmyang, robertdeng}@smu.edu.sg

***Abstract*—Zero-knowledge Succinct Non-interactive Argument of Knowledge (zkSNARK) is a powerful cryptographic primitive, in which a prover convinces a verifier that a given statement is true without leaking any additional information. However, existing zkSNARKs suffer from high computation overhead in the proof generation. This limits the applications of zkSNARKs, such as private payments, private smart contracts, and anonymous credentials. Private delegation has become a prominent way to accelerate proof generation.**

In this work, we propose Siniel, an efficient private delegation framework for zkSNARKs constructed from polynomial interactive oracle proof (PIOP) and polynomial commitment scheme (PCS). Our protocol allows a computationally limited prover (a.k.a. delegator) to delegate its expensive prover computation to several workers without leaking any information about the private witness. Most importantly, compared with the recent work EOS (USENIX'23), the state-of-the-art zkSNARK prover delegation framework, a prover in Siniel needs not to engage in the MPC protocol after sending its shares of private witness. This means that a Siniel prover can outsource the entire computation to the workers.

We compare Siniel with EOS and show significant performance advantages of the former. The experimental results show that, under low bandwidth conditions (10MBps), Siniel saves about 16% time for delegators than that of EOS, whereas under high bandwidth conditions (1000MBps), Siniel saves about 80% than EOS.

I. Introduction

Zero-knowledge Succinct Non-interactive Argument of Knowledge (zkSNARK) is a fundamental cryptographic primitive. In the zero-knowledge proof, a prover P wants to convince a verifier V some statements of the form 'Given a function F and a public instance $\mathbf{x}$, there exists a private witness $\mathbf{w}$ such that $F(\mathbf{x}, \mathbf{w}) = 1$'. The function can be any computation (in the NP space) such as the hash function, digital signature, and some other common cryptographic operations. zkSNARK enjoys short proof size and fast verification time. They are also considered one of the most promising approaches to real-world applications such as private payments [1], [2], private smart contracts [3], [4] and anonymous credentials [5], [6].

Bünz [7] pointed out that modern zkSNARKs are built from three components: a polynomial interactive oracle proof (PIOP), a polynomial commitment scheme (PCS), and the Fiat-Shamir transformation. At a high level, the function F is first represented as an arithmetic circuit and then transformed into a constraint system that includes a set of mathematical constraints, which is commonly encoded as a series of polynomials. Second, a PIOP is designed to prove that a witness-instance pair satisfies the constraint system. In this phase, a prover computes the prover polynomials, and a verifier has oracle access to these prover polynomials. Third, the PIOP is compiled into an interactive argument with PCS. In this phase, a prover sends commitments to the prover polynomials instead of the polynomial itself and interacts with a verifier to prove that the given statement is correct. Finally, the proof system can be made non-interactive with the Fiat-Shamir transformation.

Unfortunately, the costly proof generation hinders zkSNARK's practical deployment. Specifically, its performance bottleneck comes from two parts. First, the arithmetic circuit C representing computation F is often much larger and more complex than F itself. For example, although computing SHA256 is extremely fast, we still need around 20,000 multiplication gates to express the SHA256 function. Second, most existing zkSNARKs [8], [9], [10], [11] generate proof over polynomials over large prime fields, and the polynomial degrees are at least linear to the circuit size $|C|$. In addition, provers also suffer from expensive operations over high-degree polynomials including Fast Fourier Transform (FFT) and Multi-scalar Multiplication (MSM). The computation complexity grows at least linearly in $|C|$. Moreover, MSM requires tens of field operations for curve addition and thousands of field operations for multiplication by scalars [12], while the computation complexity of FFT [13] is $O(N \log N)$, in which N is the number of evaluation points. For example, we need

✉ Xiaolei Dong is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240152
www.ndss-symposium.org

to perform multiple heavy FFT and MSM operations over polynomials with degrees around 20,000 for a SHA256 compression function, which is unaffordable for a computationally limited device.

Private delegation [14], [15], [16], a special use case of secure multiparty computation (MPC), is a way to resolve the above-mentioned issues, in which a computationally limited device (e.g., mobile phone) can delegate its computation to several powerful machines (i.e., workers) without leaking any additional information about its input. Naturally, one may consider delegating the computationally expensive 'zkSNARK prover' to several powerful machines. Yet, most existing works are either with weak security, or inefficient in real-world applications. For example, Garg et al. [17] proposed zkSaaS, a general framework for private delegation. However, their protocol only achieves security in the honest majority setting against semi-honest parties. If one corrupted party conducts malicious behavior, the security of zkSaaS will be compromised. Meanwhile, Garg et al. [18] used homomorphic encryption to let a delegator directly outsource the ciphertext of its witness to a server. However, the computation overhead of homomorphic encryption is too high to be practical in real-world applications. Chiesa et al. [19] proposed EOS, a private delegation for zkSNARK prover. The EOS delegator engages in the MPC protocol with workers to check the correctness of PIOP computation. Yet, EOS suffers from high round complexity on the computationally limited delegator side.

Naturally, we raise the following question:

Is there a general framework for the private delegation of zkSNARK prover that simultaneously achieves (1) no extra interaction during the online phase for the delegator, (2) lightweight operations for the delegator, and (3) malicious security against workers?

A. Our Contributions

This paper proposes Siniel, a novel delegation framework of zkSNARK provers, to answer the above question affirmatively. We summarize our contributions as follows:

- A New General Delegation Framework of zkSNARK Prover. We construct Siniel, a novel general delegation framework of zkSNARK prover. Like EOS, this delegation framework applies to all zkSNARKs built from PIOP and PCS. Siniel allows a computationally limited prover to delegate the expensive proof generation to several powerful machines without leaking any private information. Compared with EOS, the Siniel delegator only requires offline computation and can delegate the entire computation to several workers after sending its shares of witness **without further interaction**. In addition, the security proof shows that Siniel is secure against malicious workers.
- A New 'Consistency Checker'. EOS introduces the notion of a 'consistency checker' to enable the delegator to check the consistency of prover polynomials computed by the workers. The consistency checker prevents an adversary from performing maliciously during the online phase. However, the EOS delegator needs to perform the consistency checker interactively after it receives all prover polynomials in each PIOP round. To eliminate the interaction, Siniel proposes a new non-interactive consistency checker, which is **only executed by workers** during the online phase. In brief, the Siniel delegator generates some additional information (e.g., authentication tag and authentication key) about the shares of the witness in the offline phase, and all workers jointly verify that all PIOP computations and corresponding polynomial commitments are consistent with shares of the private witness generated by the delegator. Therefore, the delegator can delegate its entire proof generation to workers **without any further interaction** during the online phase.
- Implementation and Evaluation Results. We conduct experiments to compare Siniel with EOS in terms of the computation and communication overhead on the delegator side with different bandwidths, and the computation overhead of the online phase on the worker side. The experimental results show that compared with EOS, a delegator utilizing Siniel only takes 6.5 seconds with 10MBps bandwidth to generate the proof for the SHA256 compression function, compared with 8.8 seconds of EOS, while under 1000MBps bandwidth, the Siniel delegator takes 0.17 seconds compared with 2.07 seconds of EOS delegator. Moreover, the Siniel delegator does not engage in the MPC protocol with workers. Therefore, the network is no longer a bottleneck for the private delegation protocol, and workers do not need to wait for the response from the delegator. Specifically, the Siniel consistency checker is **only executed by the workers** during the online phase.

B. Use Cases of Siniel

We discuss some use cases in which Siniel can speed up proof generation while preserving the privacy of the delegator.

Private Payment. Private payment is an essential part of web3 applications. However, it is inefficient for a resource-constrained device to generate a zkSNARK proof. Although powerful devices can help generate proofs, it is insecure for the resource-constrained device to directly delegate its private key to the powerful devices. With Siniel, a computationally limited device can outsource the proof generation to several powerful workers without leaking any private information (i.e., private key). Hence, it significantly reduces the time to complete a spend transaction and brings users a seamless experience similar to centralized payment.

Decentralized Applications (dApps). As part of the Web3 ecosystem, dApps provide decentralized services like DeFi, on-chain gaming, and digital identity management. We take the digital identity as an example. In DID, users prove that they possess certain attributes (e.g., being over 18 years old or having specific certifications) to access certain services without leaking private information. To achieve this, zkSNARK can help users generate zero-knowledge proofs of their identity attributes without revealing the actual data. However, it is inefficient for a computationally limited device to locally generate a proof. With Siniel, such device outsource the proof generation to several powerful worker to generate the final proof while preserving the privacy of the private information. This can significantly reduce the time to generate zkSNARK proofs and allow more users to participate in the web3 network.

C. Related Work

Private delegation [14], [15], [16] enables a party to delegate its computation to several workers without leaking any additional information about its private input. It is natural to consider delegating the 'zkSNARK prover' work to several workers. For example, Ozdemir et al. [12] first constructed a delegation protocol in which a set of parties with shares of the witness jointly generate proofs with respect to that witness. They proposed protocols for three zkSNARKs [10], [8], [9] based on SPDZ [20], a dishonest majority MPC protocol with additive secret sharing, and GSZ [15], an honest majority MPC protocol with guaranteed output delivery. Chiesa et al. [19] found that the delegation protocols of [12] rely on expensive cryptographic primitives to ensure the correctness of protocol execution, and they optimized [12] by introducing a 'consistency checker', in which a delegator participates in the MPC protocol to check the consistency of prover polynomials. In addition, this protocol is secure against any number of malicious adversaries. Although the security level of the proposed Siniel (i.e., honest majority) is weaker than EOS (i.e., dishonest majority), it achieves better usability and efficiency. Most importantly, the honest majority assumption is sufficient for many real-world decentralized applications, where a dishonest majority threat model is often unnecessarily strong. Siniel can be applied in decentralized systems with consensus protocols that assume an honest majority, such as those with corruption thresholds up to 1/3 for BFT or 1/2 for PoS/PoW, allowing efficient detection of malicious behavior through voting.

In a concurrent and independent work of [19], Grag et al. [18] proposed another private delegation of zkSNARK provers called zkSaaS. In zkSaaS, they design several dedicated secure multiparty computation protocols for polynomial arithmetization and MSM operations. However, this protocol is only secure under the honest majority setting against semi-honest adversaries. The security level of zkSaaS (i.e., semi-honest corruption) is weaker than that of Siniel (i.e., malicious corruption). Subsequently, Grag et al. [17] delegated the zkSNARK computation to a single untrusted server instead of several servers based on fully homomorphic encryption (FHE). Compared with Siniel, the FHE-based private delegation protocol is high in computation overhead and makes this delegation protocol less practical in real-world applications.

Additionally, some works focus on distributed ZKP, in which it scales existing ZKP to large circuits with several distributed algorithms. Namely, assume the size of the entire circuit is N and a prover holds M machines participating in the protocol. After that, each machine is responsible for generating a proof for a subcircuit of size $T = \frac{N}{M}$ with a part of the plain witness.

For example, DIZK [21] focused on delegating the prover computation to several workers. However, the communication overhead of DIZK is linear in the circuit size. Then, Xie et al. [22] proposed deVirgo, a distributed zero-knowledge proof protocol of Virgo [23]. However, deVirgo also has a linear communication cost among the workers, and the proof size is relevant to the number of workers. Recently, Liu et al. proposed Pianist [24], a distributed zero-knowledge proof protocol of Plonk [9], to improve the overall performance of DIZK. Although these protocols do not hide the part of the witness from each machine, they are still important as these protocols mentioned above focus on optimizing the space complexity of each worker. Therefore, these works are complementary to Siniel, as we can employ the techniques of distributed ZKP to improve the scalability of Siniel, and also protect the privacy of the delegator.

II. Technical Overview

In this section, we first introduce the background of zkSNARKs. After that, we review EOS and zkSaaS, two state-of-the-art private delegation protocols of zkSNARKs, as a starting point for Siniel, and point out the disadvantages of both protocols. Finally, we address these issues step-by-step.

A. Background: Design Paradigm for zkSNARKs

The design of the state-of-the-art zkSNARKs relies on two components, polynomial interactive oracle proof and polynomial commitment scheme. In this subsection, we review these components and show how to combine these two cryptographic primitives to obtain zkSNARKs.

Polynomial Interactive Oracle Proof (PIOP) for a relation $\mathbb{R} = \{(\mathbf{x}, \mathbf{w})\}$ is an interactive proof with a tuple $PIOP = (F, K, S, P, V)$ in which F is a finite field with a large prime order, K is the total rounds of PIOP, $S(j)$ is the number of prover polynomials in the j_{th} PIOP round. In each round, P receives a message from V and replies with $S(j)$ prover polynomials. Then, V can have oracle access to these prover polynomials with several evaluation points. Finally, V decides whether to accept or reject based on the response from the polynomial oracles.

Polynomial Commitment Scheme (PCS) allows a prover to first commit to a private polynomial. Then the prover opens the polynomial at a given point along with an opening proof. It is hard for a malicious prover to alter the private polynomial inside the public commitment once committed. In this paper, we focus on the KZG polynomial commitment scheme [25].

Constructing zkSNARKs from PIOP and PCS. A zkSNARK in the random oracle model for a relation $R = \{(x, w)\}$ is a tuple of algorithms $ARG = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ defined as below.

The interactive argument prover $\mathcal{P}$ and verifier $\mathcal{V}$ invoke PIOP prover $\mathbb{P}$ and verifier $\mathbb{V}$, respectively. In each PIOP round, $\mathcal{P}$ commits to the prover polynomials generated by $\mathbb{P}$ using $PCS.Commit$. Then, $\mathcal{P}$ sends these commitments to $\mathcal{V}$ instead of the polynomial oracles. After the PIOP phase, $\mathcal{V}$ invokes $\mathbb{V}$ to generate its query to polynomials inside the commitments. $\mathcal{P}$ responds to the evaluation at the given query along with an opening proof. Finally, $\mathcal{V}$ accepts or rejects based on the response.

In addition, one can apply the Fiat-Shamir transformation under the random oracle model to achieve non-interactivity. The random oracle can be instantiated with a standard cryptographic hash function.

B. Siniel: An Efficient Private Delegation Framework of zk-SNARKs

Starting point: EOS and zkSaaS. First, we review the system model of EOS and zkSaaS, as shown in Fig. 1.(a) and Fig. 1.(b), respectively.

EOS consists of an offline phase and an online phase. In the offline phase, a delegator D prepares shares of the (private) witness and distributes each share to each worker, while in the online phase, all workers jointly execute a predefined MPC protocol to generate the final proof. In EOS, D is always online and needs to perform a consistency checker in each PIOP round against malicious workers. D should wait for all workers to finish the MPC computation before executing the consistency checker. These drawbacks limit the applications of EOS.

zkSaaS also consists of an offline phase and an online phase. In the offline phase, a delegator D generates shares of the private witness and outsources the entire zkSNARK computation to several workers. In the online phase, all workers jointly generate the final proof without any further interaction with D. However, this protocol is only secure under an honest majority assumption with a semi-honest corruption model. If one of the workers conducts malicious behavior, the security and privacy of the entire protocol will be compromised.

We show how to simultaneously overcome the limitations of EOS (i.e., excessive interaction with the delegator) and zkSaaS (i.e., semi-honest security) below.

System Architecture of Siniel. We first introduce the system architecture of Siniel, shown in Fig. 1.(c). Concretely speaking, a delegating prover (a.k.a. delegator D) outsources its entire zkSNARK computation to a set of workers $P_1, P_2, \ldots, P_n$. D will not interact with workers during the online computation. In the offline phase, D generates shares of the private witness and some additional information about the shares of the private witness. Then, D distributes them to workers. In the online phase, each worker executes the delegation protocol with its share of the private witness. All workers also jointly conduct the consistency checker to check the protocol execution. Finally, if the consistency checker passes, then they aggregate the final proof and forward it to D.

Siniel guarantees that the private witness $\vec{w}$ is **completely hidden from all workers** if more than half of the workers are honest and do not collude with others. Malicious workers can arbitrarily deviate from the protocol.

Next, we show how to simultaneously eliminate the online interaction with the delegator and achieve malicious security step-by-step.

Step 1: Dividing Circuit for zkSNARK Computation into Multiple Chunks. First, we divide the circuit for zkSNARK computation into three chunks, as shown in Fig. 2. The Siniel delegator first distributes each share of the private witness along with a public instance to each worker. In the online phase (i.e., PIOP computation and proof generation), each worker executes the computation with its share of the private witness. Finally, all workers jointly reconstruct the final proof.

For the sake of simplicity, we first consider all workers to be semi-honest, in which each semi-honest worker honestly follows the protocol but tries to get private information from other honest workers.

First, each worker takes a share of the private witness and a public instance as inputs to the PIOP circuit and outputs a share of prover polynomials. PIOP circuit only consists of polynomial arithmetization so it only consists of addition and multiplication gates over the finite field, which is referred to as Add_F and Mul_F, respectively. In the PIOP circuit, each worker locally computes the sum of shares and uses the Beaver multiplication protocol to compute the multiplication of shares.

Second, each worker commits to all prover polynomials. This stage only consists of addition and multiplication over an elliptic curve group, which is referred to as Add_G and Mul_G gates, respectively. It is natural that the worker can compute the Add_G locally. The worker can also compute the Mul_G locally as at least one of the inputs is public. Therefore, each worker can locally compute the commitment to the shares of prover polynomials without any further interaction.

Third, all workers share the same Fiat-Shamir randomness and get the same evaluation point with a call to a random oracle. Each worker then evaluates the prover polynomials at the evaluation point along with an opening polynomial. This part involves polynomial arithmetization and only consists of Add_F and Mul_F. After that, each worker commits to the opening polynomial. This stage relies on operations over an elliptic curve and consists of Add_G and Mul_G. Finally, all workers jointly reconstruct the final proof with their shares of the proof.

Step 2: Enforcing Malicious Security. Second, a malicious worker may conduct various attacks in each chunk, as shown in Fig. 2. In this step, we introduce potential attacks in each chunk. We address these potential attacks by introducing a novel consistency checker in step 3.

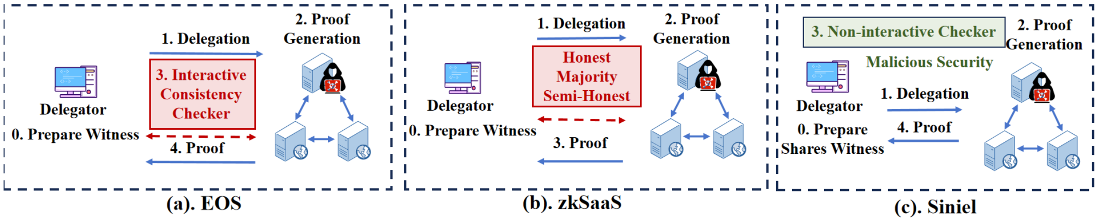
1. Inputs to PIOP Circuit: The potential attacks are as follows:

- Attack 1.1: Generate inconsistent commitment to a share of the private witness. A malicious worker may first commit to an incorrect share. This will directly lead to the failure of generating an invalid proof.

Therefore, before the PIOP computation, all workers should jointly verify that all commitments are consistent with all shares of the private witness.

2. PIOP Computation: The potential attacks are as follows:

- Attack 2.1: Generate a consistent commitment to the share of the private witness but use another share to execute the PIOP computation. A malicious worker may



(a). EOS (b). zkSaaS (c). Siniel

Fig. 1: System Model.

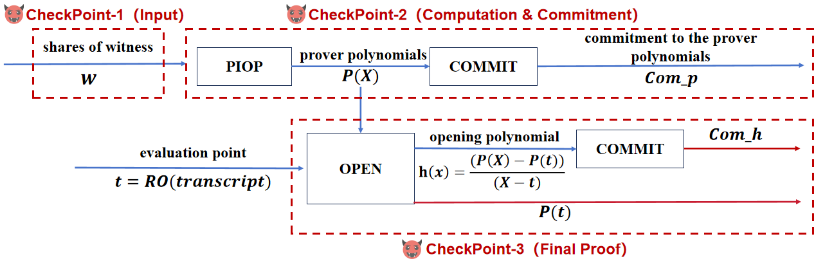


Fig. 2: Potenital Attacks on zkSNARK Circuit.

first generate a consistent commitment to the share. After that, it tampers with the share, takes the tampered share as input to the PIOP circuit, and honestly follows the protocol. Chiesa et al. [19] showed that this may leak some parts of the private witness when the proof is invalid.

Let us take an example to illustrate this attack. Suppose three workers jointly compute a final proof for a bit constraint $b \cdot (1 - b) = 0$ with additive secret sharing, and the plain witness is $b = 1$. The delegator distributes a share $[b]_i$ of $b = 1$ to worker P_i. Suppose P_1 receives $[b]_1 = -1$, P_2 receives $[b]_2 = 1$ and P_3 receives $[b]_3 = 1$ such that $[b]_1 + [b]_2 + [b]_3 = 1$.

In the EOS setting (malicious majority), two of the workers are malicious and one is honest. An adversary conducts an attack as follows: First, it controls P_2, P_3 and sees their shares of the witness. It then guesses that the actual witness b is 1, and infers that the share held by P_1 is $[b]_1 = -1$. After that, it can alter the shares held by P_2 and P_3 to shares of $b' = 2$. It sets $[b']_2 = 2$ and $[b']_3 = 1$. The share held by P_1 is 'obliviously' changed to the share of $b' = 2$. Then, the adversary honestly follows the protocol but the final proof is invalid. This invalid proof reveals the information about the original witness b.

This problem also occurs in the Siniel setting (honest majority with Shamir secret sharing). Briefly speaking, suppose there are three workers, two of them are honest and one is malicious. An adversary controls the malicious one. The adversary holds the share held by the malicious worker and guesses the plain witness. It can reconstruct the 'guessed' polynomial and infer 'guessed' shares held by the other two honest workers. Then, it alters the share held by the malicious worker and honestly follows the protocol. The shares held by two honest workers are 'obliviously' changed to other shares of an invalid witness. Finally, the proof is also invalid and leaks some information about the original witness.

- Attack 2.2: Deviate from the PIOP protocol. A malicious worker may arbitrarily deviate from the PIOP computation and generate incorrect prover polynomials. This will lead to the failure of generating a correct proof.
- Attack 2.3: Generate inconsistent commitments to the outputted prover polynomials. A malicious worker may generate inconsistent commitments to the prover polynomial or commit to other random polynomials. This will lead to the failure of generating a correct proof.

Therefore, all workers should jointly verify (1) all prover polynomials generated by each worker are consistent with the share of the private witness held by each worker (i.e., the input is correct), (2) the PIOP computation is correct, and (3) all commitments generated by each worker are consistent with all prover polynomials.

3. Proof Generation: In the third part, the potential attacks are as follows:

- Attack 3.1: Generate invalid final proofs. A malicious worker may output incorrect evaluations or corresponding opening proofs. It will lead to the failure of reconstructing a correct proof.

Therefore, all workers should jointly verify the validity of the final proof.

Step 3: A non-interactive consistency checker for checkpoints. In this step, we introduce the non-interactive consistency checker for the above-mentioned checkpoints. Note that "non-interactive" means the consistency checker is only executed among workers during the online phase.

For the first checkpoint (i.e., inputs to the PIOP circuit), we need to ensure that the commitment to the input (share of witness) generated by each worker is consistent with the share held by each worker. The delegator in Siniel generates some additional information in the offline phase to help workers verify during the online phase. Technically, for the witness $\vec{w}$, the delegator first generates a random element α, then computes $\vec{w}(\alpha)$, and generates shares of $\vec{w}(\alpha)$. In the verification phase, each worker computes shares of witness polynomials at α along with an evaluation proof. After that, they broadcast shares of witness polynomials at α and recover the witness polynomials at α. We refer to it as $\vec{w'}(\alpha)$. Finally they jointly recover $\vec{w}(\alpha)$, and check whether $\vec{w}(\alpha) = \vec{w'}(\alpha)$ and that each evaluation proof is valid. We refer to it as the **witness consistency checker**.

For the second checkpoint (i.e., PIOP computation), all workers jointly check that (1) all prover polynomials are consistent with the shares of the private witness, (2) the PIOP computation is correct, and (3) all commitments are consistent with the prover polynomials. We introduce the authentication mechanism to add verifiability to all shares. Technically, for each worker P_i, a delegator generates a share $[\vec{w}]_i$ of witness $\vec{w}$ along with an authentication tag $\tau_{[\vec{w}]_i}$ and two authentication keys $\mu, v_{[\vec{w}]_i}$ such that $\tau_{[\vec{w}]_i} = \mu \cdot [\vec{w}]_i + v_{[\vec{w}]_i}$. Then for every other worker $P_j \neq P_i$, the delegator generates a share of authentication keys $[\mu]_j$ and $v_{[\vec{w}]_j}$. The delegator distributes $[\vec{w}]_i$ along with $\tau_{[\vec{w}]_i}$ to P_i, and distributes $[\mu]_j$ and $v_{[\vec{w}]_j}$ to P_j. A malicious worker P_i can forge a tag with only a negligible probability without the authentication keys. In addition, PIOP only consists of addition and multiplication over scalar fields, and both operations enjoy linear homomorphism. Therefore, each P_i can locally update the corresponding authentication tag, while each P_j can locally update the corresponding shares of authentication keys. We refer to it as the **PIOP consistency checker**.

At the end of PIOP computation, suppose each worker P_i holds a prover polynomial $f(X) = f_0 + f_1 \cdot X + \ldots + f_d \cdot X^d$ along with corresponding tags $\tau_{f_0}, \tau_{f_1}, \ldots, \tau_{f_d}$, while each other worker P_j holds corresponding authentication keys $[\mu]_j, [v]_{f_0}, [v]_{f_1}, \ldots, [v]_{f_d}$. To check the correctness of the prover polynomial and the consistency of commitment, each worker P_i acts as a prover, and all other workers P_j act as a verifier. First, P_j sends a random challenge β to P_i. P_i responses $f(\beta)$ along with corresponding tag $\tau_{f(\beta)} = \tau_{f_0} + \tau_{f_1} \cdot \beta + \ldots + \tau_{f_d} \cdot \beta^d$ and an opening proof. Each P_j updates the corresponding authentication key as $[v_{f(\beta)}]_j = [\tau_{f_0}]_j + [v_{f_1} \cdot \beta]_j + \ldots + [v_{f_d} \cdot \beta^d]_j$. Then all P_j jointly reconstruct μ and $v_{f(\beta)}$ and verify whether the tag is consistent with the key such that $\tau_{f(\beta)} = \mu \cdot f(\beta) + v_{f(\beta)}$ and the opening proof is valid. The tag ensures that the prover polynomial is consistent with the input share as well as the PIOP computation is correct, while the opening proof ensures that the commitment is consistent with the prover polynomial. We refer to it as the PIOP consistency checker.

The third checkpoint (i.e., proof generation) is straightforward. All workers jointly reconstruct the final proof and then verify the validity of the final proof.

Final Protocol. In brief, the final protocol works as follows:

Setup. A trusted third party or an MPC ceremony is utilized to generate commitment key ck.

Offline. A delegator D distributes a share of the witness along with the authentication tag and some other additional information about the witness to P_i while distributing shares of the authentication keys to other workers.

PIOP Computation.

- Invoke the Witness Consistency Checker. Upon generating commitments to all shares of the private witness, all workers jointly invoke the witness consistency checker to check the consistency of the commitment.
- PIOP Computation. Each worker computes the prover polynomials with a PIOP circuit and commits to the prover polynomials. In addition, each worker also updates the corresponding authentication tags, while each other worker updates the corresponding authentication keys.
- Invoke the PIOP Consistency Checker. Each worker P_i and all other workers jointly invoke the PIOP consistency checker to check the correctness of PIOP computation and the consistency of the prover polynomials and the commitments.

Proof Generation. Each worker generates evaluations of prover polynomials along with opening proofs. Then, all workers reconstruct the final proof. Finally, they verify the validity of the final proof before sending it to the delegator.

By combining these optimizations, we propose Siniel, a novel private delegation protocol of zkSNARK provers, that achieves no online interaction with the delegator as well as malicious security.

III. Preliminaries

Notations. In this paper, we denote $[s]_i$ is a share of secret s held by the worker P_i, while $[\vec{s}]_i$ is a share of a vector of secret $\vec{s}$ held by P_i. τ_w, μ, v_w are denoted as the authentication tag of the share w, the global authentication key, and the local authentication key of v_w, respectively. In addition, $\mu^{(i)}$ means the global authentication key of the worker P_i. In addition, n is the number of workers, while t is the maximum number of corrupted workers (i.e., threshold).

A. Shamir Secret Sharing

In Siniel, we use the Shamir secret sharing (SSS) scheme. For a finite field F_p, a degree-t secret sharing is a vector $(s_1, \ldots, s_n)$, which satisfies that, there exists a polynomial f of degree at most t, such that $f(0) = s$. We denote each share of secret s held by worker P_i as $[s]_i$. Formally, a (t, n) threshold SSS consists of two algorithms:

- SSS.$Share(t, n, s) \rightarrow ([s]_1, [s]_2, \ldots, [s]_n)$: Select a random polynomial $f(x) = s + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$, where s is the secret. Choose n different values id_i, compute the value $[s]_i$ over $f(x)$, set $[s]_i = f(id_i)$. In default, we can use $id_i = i$, and each share has the form $(i, [s]_i)$. Usually, we write $(i, [s]_i)$ as $[s]_i$ for simplicity.
- SSS.$Recover(t, n, ([s]_1, [s]_2, \ldots, [s]_n)) \rightarrow s$: Upon receiving $([s]_1, [s]_2, \ldots, [s]_n)$, interpolate as a polynomial $f(x)$ over $(x_i, [s]_i)$ for $i \in [n]$,compute $s = f(0)$.

Additionally, the Shamir shares of commitments and opening proofs lie in the exponentiation of the generator. Therefore, we adopt the algorithm proposed by Applebaum et al. [26] to reconstruct the secret over the exponent with $O(n)$ additions.

B. Multiparty Computation with Authentication Tags

Multiparty computation allows several parties to compute a given function without leaking any additional information. Some existing MPC protocols [15] use the authentication tag mechanism to enhance security. An authentication tag is a new way to commit to shares generated by a dealer. It helps detect any dishonest parties who lie about their shares during computation. This is necessary for the honest majority setting, where SSS is used to do the secret sharing.

We use the idea of [27] to instantiate the authentication tag. In more detail, a worker P_i, acting as a prover, holds a share $[x]_i$ and an attached authentication tag τ_i, while another worker P_j, acting as a verifier, holds the corresponding authentication key (μ, v), such that $\tau_i = \mu \cdot [x]_i + v$. P_i is not able to tamper the share $[x]_i$ to another share $[x']_i$ without the authentication key (μ, v) except with negligible probability. For incorrect shares, the verifier can easily detect and filter out.

In some previous works, the tag is locally computed by the dealer. In Siniel, the delegator honestly follows the protocol in the offline phase. It can directly generate shares of the private witness, and distribute authentication keys and tags to other workers. In addition, with the nice linear homomorphism of the authentication tag, the prover and the verifier can locally update authentication tags and authentication keys during the MPC computation, respectively. At the end of the protocol, the verifier can check the correctness of protocol execution by checking the corresponding authentication tags. The protocol is executed correctly if the authentication tags of the final shares held by P_i are consistent with the authentication keys held by P_v. Namely, in the Siniel setting, each worker P_i acts as a prover with a share $[w]_i$ and corresponding tag τ_i, while each other worker P_j acts as a verifier with a share of authentication tags $[\mu]_j$ and $[v]_j$. During the PIOP computation, P_i updates authentication tags while each other worker P_j updates a share of authentication keys, In the verification phase, P_i responds with a final share $[x]_i$ along with a corresponding tag τ_i, then all P_j reconstruct authentication keys μ and v, and check whether $\tau_i = \mu \cdot [x]_i + v$.

C. Beaver Triples

In SSS-based multiparty computation, a dealer shares a secret s and all parties compute over the shares of secret s. Finally, they jointly aggregate the final results with their computed shares. For the addition gate, it is cheap as all parties can locally compute the sum of corresponding shares without any further interaction. Meanwhile, for the multiplication gate, all parties should use the Beaver multiplication protocol to compute the product of shares.

Suppose there exists a trusted third party (or a secure protocol) in the preprocessing phase to generate a Beaver triple $([a]_i, [b]_i, [c]_i)$ such that $c = a \cdot b$. Then every party P_i holds a Beaver triple $([a]_i, [b]_i, [c]_i)$. For two private inputs $([x]_i, [y]_i)$, P_i executes the following step to get $[x \cdot y]_i$.

- Locally compute

$$[A]_i = [x]_i - [a]_i = [x - a]_i$$

$$[B]_i = [y]_i - [b]_i = [y - b]_i$$

- Open $[A]_i$, $[B]_i$ to let all parties learn $A = x + a$, $B = y + b$.
- Let all workers compute

$$[z]_i = A \cdot [b]_i + B \cdot [a]_i + [c]_i + A \cdot B$$

In addition, in the Siniel setting, except for computing the share, each P_i needs to compute a corresponding authentication tag, while each other worker P_j needs to update the share of the corresponding authentication key as follows:

- Each P_i broadcasts

$$[A]_i = [x]_i - [a]_i = [x - a]_i$$

$$[B]_i = [y]_i - [b]_i = [y - b]_i$$

- Each P_i reconstructs A and B.
- Each P_i computes

$$[z]_i = [x \cdot y]_i = A \cdot [b]_i + B \cdot [a]_i + [c]_i + A \cdot B$$

$$\tau_{[z]_i} = A \cdot \tau_{[b]_i} + B \cdot \tau_{[a]_i} + \tau_{[c]_i}$$

while P_j computes

$$[v_{[z]_i}]_j = A \cdot [v_{[b]_i}]_j + B \cdot [v_{[a]_i}]_j + [v_{[c]_i}]_j - [\mu^{(i)}]_j \cdot A \cdot B$$

D. Polynomial Commitment Scheme

In a polynomial commitment scheme (PCS), a sender first commits to a private polynomial p, and later opens the polynomial p at a given point α. It should satisfy completeness and knowledge soundness under the algebraic group model (AGM). In Siniel, we use the KZG polynomial commitment scheme [25] to instantiate the PCS, and KZG is a tuple of algorithms $(Setup, Commit, Open, Verify)$ defined as follows:

- $KZG.Setup(1^\lambda) \rightarrow ck$: On input a security parameter λ, output a commitment key ck.
- $KZG.Commit(pub\ ck, priv\ p) \rightarrow pub\ C$: On input ck and a private polynomial p, output a public commitment C to p.
- $KZG.Open(pub\ ck, priv\ p, pub\ C, pub\ point\ z) \rightarrow (pub\ v, pub\ \pi)$: On input ck, p, C and a given point z, output its public evaluation $v = p(z)$ along with the opening proof π.
- $KZG.Verify(pub\ ck, pub\ \pi, pub\ C, pub\ v, pub\ z) \rightarrow 1/0$: On input ck, π, C, $v = p(z)$ and output accept (1) or reject (0) based on π.

E. Polynomial Interactive Oracle Proof

A polynomial interactive oracle proof (PIOP) for a relation $\mathbb{R}$ is an interactive proof with a tuple $PIOP = (F, K, S, \mathbb{P}, \mathbb{V})$ in which F is a finite field, K is the total round of PIOP, $S(j)$ is the number of prover polynomials in the j_{th} round, $\mathbb{P}$ and $\mathbb{V}$ are defined as below.

In each round $j \in [K]$, $\mathbb{P}(F, \mathbf{x}, \mathbf{w})$ receives a message $\mu_j \in F^*$ from $\mathbb{V}(x)$ and replies with $S(j)$ prover polynomials $p_{j,1}, \dots, p_{j,s(j)}$. $\mathbb{V}$ then have oracle access to these prover polynomials with several evaluation points. After the interaction, the verifier accepts or rejects. The PIOP should satisfy perfect completeness, negligible knowledge soundness error, and zero knowledge.

F. zkSNARK

A zkSNARK in the random oracle model is a tuple of algorithm $ARG = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ for a relation $\mathbb{R} = \{(\mathbf{x}, \mathbf{w})\}$ shown as follows.

Constructing zkSNARKs from PIOP and PCS. The interactive argument prover $\mathcal{P}$ and verifier $\mathcal{V}$ invoke PIOP prover $\mathbb{P}$ and verifier $\mathbb{V}$, respectively. In each round, $\mathcal{P}$ commits to the prover polynomials generated by $\mathbb{P}$ using $PCS.Commit$ instead of directly sending the polynomial oracles. Then, $\mathcal{P}$ forwards these commitments to $\mathcal{V}$. After the interaction, $\mathcal{V}$ invokes $\mathbb{V}$ to generate its query to the committed polynomials. $\mathcal{P}$ responds to the evaluation at the given query along with an opening proof. Finally, $\mathcal{V}$ accepts or rejects based on the response. To get a zkSNARK, one can apply the Fiat-Shamir transformation under the random oracle model. In addition, we can use any cryptographic hash function to instantiate the random oracle.

IV. Circuits for Common Operations

A. Circuit Model

Many efficient MPC protocols express a computation as an arithmetic circuit, only with field addition and multiplication gates. In our case, the computation also requires elliptic curve group arithmetic, random oracle as well as recovering the secret. The gates are defined in Definition 1, while the gate executions are shown in Fig. 3.

The proof of KZG-based zkSNARKs includes three parts, commitments to the prover polynomials, evaluations of the prover polynomials at given points, and opening proofs to the evaluations. The commitments and opening proofs are group elements, while the evaluations are scalar elements. In Siniel, each worker holds a share of these elements after the MPC protocol. They aggregate these shares to get the final zkSNARK proof. To recover various types of shares, we extend the EOS circuit model by introducing two new gates, $Output_F$ for scalar aggregation and $Output_G$ for group element aggregation. In addition, private visibility means that a worker holds a share of the secret, while public visibility means that a worker holds a public value.

When evaluating the Add_F and Mul_F, each worker P_i needs to compute its share $[w]_i$ along with a corresponding authentication tag $\tau_{[w]_i}$, while other party P_j holds a share of global authentication key $[\mu^{(i)}]_j$ and needs to compute a new authentication key $[v_{[w]_i}]_j$ corresponding to the share $[w]_i$, such that $\tau_{[w]_i} = \mu^{(i)} \cdot [[w]]_i + v_{[w]_i}$.

Definition 1: Let F be a finite field with a large prime p, G be the $p-$order subgroup of an elliptic curve, and $transcript$ stores all public values and is initialized empty. Then, the circuit model consists of the following gates:

- $Add_F(w_i \in F, w_j \in F) \to w_k \in F$: Set $w_k = w_i + w_j$, where $+$ denotes scalar addition in F.
- $Mul_F(w_i \in F, w_j \in F) \to w_k \in F$: Set $w_k = w_i \cdot w_j$, where $\cdot$ denotes scalar multiplication in F.
- $Add_G(w_i \in G, w_j \in G) \to w_k \in G$: Set $w_k = w_i + w_j$, where $+$ denotes addition in G.
- $Mul_G(w_i \in F, w_j \in G) \to w_k \in G$: Set $w_k = w_i \cdot w_j$, where $\cdot$ denotes multiplication in G. In KZG circuits, $w_j \in G$ has public visibility.
- $Reveal(w_i) \to pub\ w_i$: Set w_i public, and put it into the $transcript$.
- $RO(public\ transcript) \to pub\ w_k = \rho(transcript)$: Output a public random challenge w_k with a call to the random oracle ρ, and add the output to $transcript$.
- $Output_F([w_i] \in F) \to pub\ w_i \in F$: Get shares of $w_i \in F$ from at least $t + 1$ workers, and recover $w_i \in G$.
- $Output_G([w_i] \in G) \to pub\ w_i \in G$: Get shares of $w_i \in G$ from at least $t + 1$ workers, and recover $w_i \in G$.

The circuit model satisfies:

- Each wire takes an element in either F or G, and is either private or public. If the input w is public, each party holds a publicly known value w, otherwise, each party P_i holds a share $[w]_i$ of w.
- For every gate except Reveal, RO, and Output, the output w_k is public if and only if both w_i and w_j are public.

B. Circuit for PIOP

The fundamental objects in PIOP-based zkSNARKs are polynomial arithmetic. We now describe efficient circuits for common operations. If one of the inputs to the PIOP circuit is private, then the output is private. PIOP circuits include polynomial addition, FFT, inverse FFT (IFFT), polynomial evaluation at a public point, polynomial multiplication, and polynomial division. All operations in PIOP circuits are over gates Add_F and Mul_F. The difference from EOS is that we support multiplication between two polynomials with any visibility. The circuits for PIOP are shown in Fig. 4.

C. Circuit for KZG

A circuit for the KZG polynomial commitment scheme consists of two operations: committing to a private polynomial and opening the committed private polynomial at a given point. Except for evaluating the polynomial at a given point, all other KZG operations are over the group element (Add_G and Mul_G). Therefore, the MSM is the core building block for KZG. In addition, the difference from EOS is that each party should reveal its commitment to the shared polynomial once committed and its opening proofs once evaluated. The circuits for KZG are shown in Fig. 5.

Each worker P_i and each other worker P_j $(i \neq j)$ proceed to each gate as follows:

- $Add_F(w_a \in F, w_b \in F) \rightarrow w_c \in F$:
  - If w_a and w_b are both public, P_i locally computes pub $w_c = w_a + w_b$, while P_j does nothing.
  - If w_a and w_b are both private, P_i locally computes priv $[w_c]_i = [w_a]_i + [w_b]_i$ and $\tau_{[w_c]_i} = \tau_{[w_a]_i} + \tau_{[w_b]_i}$, while P_j locally computes $[v_{[w_c]_i}]_j = [v_{[w_a]_i}]_j + [v_{[w_b]_i}]_j$.
  - If w_a is public and w_b is private, then P_1 sets $[w_c]_1 = w_a + [w_b]_1$ and $\tau_{[w_c]_1} = \tau_{[w_b]_1}$,other worker P_i sets $[w_c]_i = [w_b]_i$ and $\tau_{[w_c]_i} = \tau_{[w_b]_i}$, while P_j sets $[v_{[w_c]_1}]_j = [v_{[w_b]_1}]_j - [\mu^{(i)}]_j \cdot w_a$ and $[v_{[w_c]_i}]_j = [v_{[w_b]_i}]_j$.
- $Mul_F(w_a \in F, w_b \in F) \rightarrow w_c \in F$:
  - If w_a and w_b are both public, locally computes pub $w_c = w_a \cdot w_b$.
  - If w_a and w_b are both private, P_i and P_j jointly compute priv $[w_c]_i = [w_a \cdot w_b]_i$, $\tau_{[w_c]_i}$ and $[v_{[w_c]_i}]_j$ with the Beaver multiplication protocol as shown in section III-C.
  - If w_a is public and w_b is private, P_i locally computes priv $[w_c]_i = w_a \cdot [w_b]_i$ and $\tau_{[w_c]_i} = w_a \cdot \tau_{[w_b]_i}$, while P_j locally computes $[v_{[w_c]_i}]_j = w_a \cdot [v_{[w_b]_i}]$.

  Each worker P_i proceeds to each gate as follows:
- $Add_G(w_a \in G, w_b \in G) \rightarrow w_c \in G$:
  - If w_a and w_b are both public, P_i computes pub $w_c = w_a + w_b$.
  - If w_a and w_b are both private, P_i computes priv $[w_c]_i = [w_a]_i + [w_b]_i$.
  - If w_a is public and w_b is private, then P_1 sets $[w_c]_i = w_a + [w_b]_i$, while other worker P_i sets $[w_c]_i = [w_b]_i$.
- $Mul_G(w_a \in F, w_b \in G) \rightarrow w_c \in G$:
  - If w_a and w_b are both public, P_i locally computes pub $w_c = w_a \cdot w_b$.
  - If w_a is private and w_b is public, P_i locally computes priv $[w_c]_i = [w_a]_i \cdot w_b$.
- $Reveal(w_a) \rightarrow w_b$:
  - If w_a is public, set $w_b = w_a$.
  - If w_a is private, make w_a public, and set $w_b = w_a$.
- $RO(transcript) \rightarrow w_b = \rho(transcript)$:
  Set $w_b = \rho(transcript)$.
- $Output_F(\{[w_a]_i\} \in F\}_{i\in[n]}) \rightarrow w_a \in F$:
  1. P_i broadcasts its share $[w_a]_i \in F$.
  2. Upon receiving all shares, recover secret

$$w_a = SSS.Recover(t, n, ([w_a]_1, [w_a]_2, \ldots, [w_i]_n))$$

- $Output_G(\{[w_a]\} \in G\}_{i\in[n]}) \rightarrow w_a \in G$:
  1. P_i broadcasts its share $[w_a]_i \in G$.
  2. Upon receiving all shares, use algorithm in [26] to recover $w_a \in G$.

Fig. 3: Gate Execution of the Siniel Online Phase.

V. Consistency Checker for PIOPs

As described in the previous section, a corrupted worker may alter its share of witness and generate the final proof with this invalid share. This may leak some parts of the private witness and cause privacy leakage. In the Siniel setting, all workers should jointly verify the correctness of the prover polynomials. In addition, the delegator does not engage in the online phase after it distributes each share of the private witness to each worker.

Therefore, we introduce two consistency checkers named witness consistency checker and PIOP consistency checker. The witness consistency checker checks that the commitment to each share of witness is correct, while the PIOP consistency checker checks that (1). the prover polynomials are consistent with the input shares as well as the commitments, and (2). the prover polynomial is correctly generated. The security models for the witness consistency checker and PIOP consistency checker are shown in the Appendix.A.

A. Construction of Witness Consistency Checker

Before the PIOP computation, all workers jointly execute the witness consistency checker to verify whether the commitment is consistent with the share of the private witness. The protocol for the consistency checker is proposed in Fig. 6. The security proof for the witness consistency checker protocol is given in Appendix.C.

In more detail, each worker P_i first broadcasts its share $[\alpha]_i$. Upon receiving all shares of α from other workers, it broadcasts the evaluation $[\vec{w}'(\alpha)]_i$ along with an opening proof. After that, P_i aggregates the evaluation $\vec{w}'(\alpha)$ along with the opening proof, then broadcasts its share $[\vec{w}(\alpha)]_i$.

$PolyAdd(poly\ p_1, poly\ p_2) \rightarrow poly\ p_3$:
For $i \in 0, \ldots, d$, set $p_{3,i} := Add_F(p_{1,i}, p_{2,i})$.
Claim 1: This circuit requires no interaction.

$FFT(poly\ p_1, pub\ subgroupH) \rightarrow \{p(w^j)\}_{i=0}^{|H|-1}$:
Compute the FFT with the standard algorithm [13], using only additions and multiplications by public values.
Claim 2: This circuit requires no interaction.

$IFFT(evaluations\ \{p(w^j)\}_{i=0}^{|H|-1}, pub\ subgroup\ H) \rightarrow poly\ p$:
Compute the IFFT with the standard algorithm [13], using only additions and multiplications by public values.
Claim 3: This circuit requires no interaction.

$PolyEval(poly\ p_1, pub\ point\ z) \rightarrow poly\ p(z)$:
Set $p(z) = \sum_{i=0}^{d} p_{1,i} \cdot z^i$
Claim 4: This circuit requires no interaction.

$PolyMul(poly\ p_1, poly\ p_2) \rightarrow poly\ p_3$:
1. Construct a domain H with at least $2d + 1$ points.
2. Compute $e_1 := FFT(p_1, H)$ and $e_2 := FFT(p_2, H)$.
3. Compute $e_{3,i} = Mul_F(e_{1,i}, e_{2,i})$.
4. Compute $p_3 := IFFT(e_3, H)$.
Claim 5: If both polynomials are private, This circuit requires one round of interaction. otherwise, it requires no interaction.
Proof: If both polynomials are private, then we need to use Beaver multiplication protocol to compute the $Mul_F(e_{1,i}, e_{2,i})$, which consists of one interaction round, otherwise, they can locally compute the final results.

$PolyDiv(poly\ p_1, pub\ poly\ d) \rightarrow (q, r)$:
1. Obtain quotient q and remainder r via Euclidean division.
Claim 6: This circuit requires no interaction.

Fig. 4: Circuit for PIOP

Finally, upon receiving all shares of $\vec{w(\alpha)}$ along with the opening proofs, then P_i checks if $\vec{w}'(\alpha) = \vec{w(\alpha)}$ and the corresponding opening proof is valid. If at least $t + 1$ workers disagree with the verification, then the protocol outputs 0.

B. Construction of PIOP Consistency Checker

For each worker P_i, each other worker P_j acts as a verifier and runs the PIOP consistency checker as shown in Fig. 7 to check the correctness of the PIOP computation executed by P_i. The PIOP consistency checker checks that (1). the prover polynomials are computed correctly with a given PIOP circuit; (2). the prover polynomials are consistent with the input share of the private witness; (3). the commitments are consistent with the prover polynomials. The security proof for the PIOP consistency checker protocol is given in Appendix.D.

$MSM(priv\ \vec{c} \in F^d, pub\ \vec{G} \in G^n) \rightarrow priv\ R \in G$:
Output $R := \sum_{i=0}^{d} Mul_G(c_i, G_i)$.
Claim 7: This circuit requires no interaction.
Proof: As is claimed above, one can locally compute the Mul_G gate without any interaction.

$KZG.Commit(pub\ ck, priv\ poly\ p) \rightarrow C$:
1. Parse ck as $\{\alpha^i \cdot G\}_{i=0}^{d}$.
2. $C := MSM(p, \{\alpha^i \cdot G\}_{i=0}^{d})$.
3. $Reveal(C)$.
Claim 8: This circuit requires one round of interaction.
Proof: Once a polynomial is committed, the party invokes the $Reveal$ gate to broadcast its commitment to the private polynomial.

$KZG.Open(pub\ ck, priv\ poly\ p, pub\ C, pub\ point\ z) \rightarrow (v = p(z), proof\ \pi)$:
1. Compute $priv\ v = p(z) = PolyEval(p, z)$.
2. Compute proof polynomial $\hat{w} := PolyDiv(p - v, X - z)$.
3. Commit to $\hat{w}$ as $\hat{W} = KZG.Commit(ck, w)$.
4. $Reveal(v, \pi = \hat{W})$.
Claim 9: This circuit requires one round of interaction.
Proof: Once a polynomial is opened, the party invokes the $Reveal$ gate to broadcast its evaluation result $v = p(z)$ along with the proof π.

Fig. 5: Circuit for PCS

Parameters:
$[\vec{w}]_i$ is the share of the private witness $\vec{w}$ held by P_i.
$[\vec{w}]_i(\alpha)$ is the evaluation of the corresponding (share of) witness polynomial at point α.
Protocol: Each worker P_i proceeds the protocol as follows:
1. Broadcast share $[\alpha]_i$, and recover α upon receiving all shares.
2. Compute the opening proof

$$([\vec{w}]_i(\alpha), \pi_{[\vec{w}]_i}) := KZG.Open(ck, [\vec{w}]_i, C_{[\vec{w}]_i}, \alpha)$$

3. Aggregate all shares of opening proof

$$C_{\vec{w}} := Output_G(C_{[\vec{w}]_i})$$

$$\vec{w}'(\alpha) := Output_F([\vec{w}]_i(\alpha))$$

$$\pi_{\vec{w}} := Output_G(\pi_{[\vec{w}]_i})$$

4. Invoke $Reveal([\vec{w}(\alpha)]_i)$, and computes

$$\vec{w}(\alpha) := Output_F([\vec{w}(\alpha)]_i)$$

5. Output 1 if $\vec{w}'(\alpha) = \vec{w}(\alpha)$ and

$$KZG.Verify(ck, \pi_{\vec{w}}, C_{\vec{w}}, \vec{w}(\alpha), \alpha) = 1$$

otherwise outputs 0.

Fig. 6: The Protocol for Witness Consistency Checker π_{wcc}

At a high level, the delegator distributes a share of the private witness along with corresponding authentication tags to each worker P_i, while each other worker P_j gets a share of the authentication keys. It is hard for P_i to generate another valid authentication tag for another share without the corresponding authentication keys. With the linear homomorphism of the authentication tag mechanism, during the PIOP computation, each worker P_i updates its corresponding authentication tag, while each other worker P_j updates the share of the authentication keys. Then, P_i generates commitments to the prover polynomials.

After that, each worker P_i and all other workers jointly execute the PIOP consistency checker as follows. For a sake of simplicity, we assume that P_i outputs a prover polynomial $f(X) = f_0 + f_1 \cdot X + \ldots + f_d \cdot X^d$ along with corresponding tags $\tau_{f_0}, \tau_{f_1}, \ldots, \tau_{f_d}$ and a commitment com_f to $f(X)$, while each other worker P_j outputs shares of final authentication key $[\mu]_j, [v_{f_0}]_j, [v_{f_1}]_j, \ldots, [v_{f_d}]_j$ such that $\tau_{f_k} = \mu \cdot f_k + v_{f_k}$, for each $k \in [0..d]$.

First, all other workers send a random challenge β to P_i. P_i responses with an evaluation $f(\beta)$, a corresponding authentication tag $\tau_{f(\beta)} = \tau_{f_0} + \tau_{f_1} \cdot \beta + \ldots + \tau_{f_d} \cdot \beta^d$, and a KZG opening proof to the evaluation. Then, each other worker P_j updates the corresponding authentication key as $[v_{f(\beta)}]_j = [v_{f_0}]_j + [v_{f_1}]_j \cdot \beta + \ldots + [v_{f_d}]_j \cdot \beta^d$. Finally, they jointly reconstruct μ and $v_{f(\beta)}$, check whether $\tau_{f(\beta)} = \mu \cdot f(\beta) + v_{f(\beta)}$ and the KZG opening proof is valid.

The authentication tag mechanism ensures that the prover polynomials are computed correctly with a given PIOP circuit and are consistent with the input shares, while the KZG opening proof ensures that the commitments are consistent with the prover polynomials.

VI. Siniel: Delegated zkSNARK

Siniel consists of setup, offline, PIOP computation, and proof generation phases, as shown in Fig. 8. Before PIOP computation, all workers invoke the witness consistency checker to jointly verify that the commitments to the input shares are correct. At the end of each PIOP round, each worker P_i proves to all other parties that its PIOP computation is correct and all commitments to the prover polynomials are consistent. Finally, all workers jointly open the prover polynomials at given evaluation points and generate corresponding opening proofs. If more than $t + 1$ workers do not agree with the verification at any time, then the protocol aborts. In addition, the security model and the security proof of Siniel are shown in Appendix.B and Appendix.E, respectively.

Setup: A trusted setup will be performed to initialize the commitment key ck. This operation will be performed only once as we can reuse ck in subsequent executions. This stage can be securely implemented with an MPC ceremony.

Offline: The delegator distributes a share of the private witness along with authentication tags to each worker P_i and sends the share of the authentication keys to each other worker P_j. The authentication mechanism ensures that a malicious worker P_i is hard to alter the share and generate a valid authentication tag without the authentication keys. To compute the Beaver multiplication protocol, the delegator generates Beaver triples, the corresponding authentication tags, and authentication keys. In addition, the delegator also generates some additional information about the private witness to help all workers execute the witness consistency checker.

Parameter:
$S(t)$ is the number of prover polynomials in the t_{th} PIOP round.
d is the maximum degree of prover polynomials.
$com_{f^{(k)}}$ is the commitment to a prover polynomial $f^{(k)}$.
Protocol: For each worker P_i and each other worker P_j, they proceed the protocol as follows:
1. All workers except P_i send a random challenge β to P_i.
2. P_i responses with $(f^{(k)}(\beta), \tau_{f^{(k)}(\beta)}, \pi_k)$ to each worker P_j, for each $k \in S(t)$, in which

$$(f^{(k)}(\beta), \pi_k) = KZG.Open(ck, f, com, \beta)$$

$$\tau_{f^{(k)}(\beta)} = \tau_{f_0^{(k)}} + \tau_{f_1^{(k)}} \cdot \beta + \ldots + \tau_{f_d^{(k)}} \cdot \beta^d$$

3. For each $k \in S(t)$, each P_j computes

$$[v_{f^{(k)}(\beta)}]_j = [v_{f_0^{(k)}}]_j + [v_{f_1^{(k)}}]_j \cdot \beta + \ldots + [v_{f_d^{(k)}}]_j \cdot \beta^d$$

4. For each $k \in S(t)$, all workers except P_i reconstruct the authentication key $(\mu, v_{f^{(k)}(\beta)})$, and verify whether

$$\tau_{f^{(k)}(\beta)} = \mu \cdot f^{(k)}(\beta) + v_{f^{(k)}(\beta)}$$

$$KZG.Verify(ck, \pi_k, com_{f^{(k)}}, f^{(k)}(\beta), \beta) = accept$$

Fig. 7: The Protocol for PIOP Consistency Checker π_{wcc}

PIOP Computation: PIOP computation consists of three stages. First, all workers jointly execute the witness consistency checker to ensure that each worker uses the correct share of the private witness to generate the initial commitment. Second, all workers jointly execute the PIOP computation and commit to the prover polynomials. At the end of each PIOP round, each worker P_i and all other workers execute the PIOP consistency checker to ensure the correctness of the PIOP computation and the consistency of commitments.

Proof Generation: If both verifications pass in the PIOP computation, each worker P_i first applies the Fiat-Shamir transformation for public randomness to get evaluation points. Then, P_i computes evaluations of the prover polynomials along with opening proofs. After that, all workers jointly reconstruct the final proof and verify its validity. If at least $t + 1$ workers agree with the verification, then all workers send the final proof to the delegator.

VII. Implementation and Performance

A. Experimental Setup

We use one delegator and three workers in the experiment. In the private delegation of zkSNARK like Siniel and EOS, an excessive number of workers would significantly increase

Setup: Run $KZG.Setup(1^\lambda) \rightarrow ck = (g, \tau \cdot g, \tau^2 \cdot g, \dots, \tau^t \cdot g)$.

Offline: For each worker P_i and each other worker P_j, D does following:
1. Parse $\mathbf{w}$ as a vector $\vec{w}$, and generate share $[\vec{w}]_i$ of witness $\vec{w}$ with SSS.
2. Sample a random point $\alpha \leftarrow F$, and compute $\vec{w}(\alpha)$ with barycentric evaluation [28] in $O(n)$ field operations, and generate share $[\vec{w}(\alpha)]_i$ and $[\alpha]_i$.
3. Choose a global authentication key $\mu^{(i)}$ and for each $[w_k]_i \in [\vec{w}]_i$ choose a local authentication key $v_{[w_k]_i}$.
4. For each $[w_k]_i \in [\vec{w}]_i$, compute $\tau_{[w_k]_i} = \mu^{(i)} \cdot [w_k]_i + v_{[w_k]_i}$, generates shares $[\mu^{(i)}]_j$ and $[v_{[w_k]_i}]_j$.
5. Generate a sufficient number of Beaver triples $\{([a_m]_i, [b_m]_i, [c_m]_i)\}_{m \in [\gamma \cdot C_{Mul}]}$, in which γ is a predefined constant, C_{Mul} is the number of Mul_F.
6. Generate corresponding authentication keys $\{(v_{[a_m]_i}, v_{[b_m]_i}, v_{[c_m]_i})\}_{m \in [\gamma \cdot C_{Mul}]}$ and authentication tags $\{(\tau_{[a_m]_i} = \mu^{(i)} \cdot [a_m]_i + v_{[a_m]_i}, \tau_{[b_m]_i} = \mu^{(i)} \cdot [b_m]_i + v_{[b_m]_i}, \tau_{[c_m]_i} = \mu^{(i)} \cdot [c_m]_i + v_{[c_m]_i})\}_{m \in [\gamma \cdot C_{Mul}]}$.
7. Generate shares of corresponding keys $\{([v_{[a_m]_i}]_j, [v_{[b_m]_i}]_j, [v_{[c_m]_i}]_j)\}_{m \in [\gamma \cdot C_{Mul}]}$.
8. Send $([\alpha]_i, \mathbf{x}, [\vec{w}]_i, [\vec{w}(\alpha)]_i, \{\tau_{[w_k]_i}\}_{[w_k]_i \in [\vec{w}]_i}, \{([a_m]_i, [b_m]_i, [c_m]_i)\}_{m \in [\gamma \cdot C_{Mul}]}, \{(\tau_{[a_m]_i}, \tau_{[b_m]_i}, \tau_{[c_m]_i}\}_{m \in [\gamma \cdot C_{Mul}]})$ to P_i, and $([\mu^{(i)}]_j, \{[v_{[w_k]_i}]_j\}_{[w_k]_i \in [\vec{w}]_i}, \{([v_{[a_m]_i}]_j, [v_{[b_m]_i}]_j, [v_{[c_m]_i}]_j)\}_{m \in [\gamma \cdot C_{Mul}]})$ to P_j.

PIOP Computation:
1. Each worker P_i do following:
1.1 Initialize $transcript = \mathbf{x}$, and prover state $st_p = (\mathbf{x}, [\vec{w}]_i)$.
1.2 P_i sets the round $j = 1$, and total PIOP round is K.
1.3 Compute commitment to shares of witness: $\vec{C}_{[\vec{w}]_i} := KZG.Commit([\vec{w}]_i, ck)$.
2. All workers invoke the ideal functionality F_{wcc} of the witness consistency checker as shown in Fig. 11, each P_i takes $([\vec{w}]_i, [\vec{w}]_i(\alpha), \vec{C}_{[\vec{w}]_i}, [\alpha]_i)$ as inputs, if F_{wcc} outputs 0, then the protocol aborts.
3. For each $k \in K$, each worker P_i and each other worker P_j do following:
3.1 P_i computes $\beta_k := RO(transcript)$ as randomness of the PIOP verifier $\mathbb{V}$.
3.2 P_i computes prover polynomials $[\vec{p}_k]_i$ with the PIOP prover $\mathbb{P}$: $(st_p, [\vec{p}_k]_i) := \mathbb{P}(st_p, \beta_k)$, and updates corresponding authentication tags $\tau_{[\vec{p}_k]_i}$, while P_j updates corresponding authentication keys $([\mu^{(i)}]_j, [v_{[\vec{p}_k]_i}]_j)$, as shown in Fig. 3.
3.3 P_i commits to all prover polynomials: $\vec{C}_{[\vec{p}_k]_i} := KZG.Commit([\vec{p}_k]_i, ck)$.
3.4 P_i and all other workers invoke the ideal functionality F_{pcc} of the PIOP consistency checker as shown in Fig. 12. P_i takes $(ck, \mathbf{x}, [\vec{w}]_i, \tau_{[\vec{w}]_i}, \vec{p}_k]_i, \tau_{[\vec{p}_k]_i}, \vec{C}_{[\vec{p}_k]_i})$ as inputs, while each P_j takes $([\mu^{(i)}]_j, [v_{[\vec{p}_k]_i}]_j)$ as inputs, if F_{pcc} outputs 0, then the protocol aborts.
3.5 Set $k = k + 1$.

Proof Generation: For each $k \in [K]$, each worker P_i does following:
1. Compute query set $Q := RO(transcript)$, and evaluate shared polynomials at Q, compute proofs as

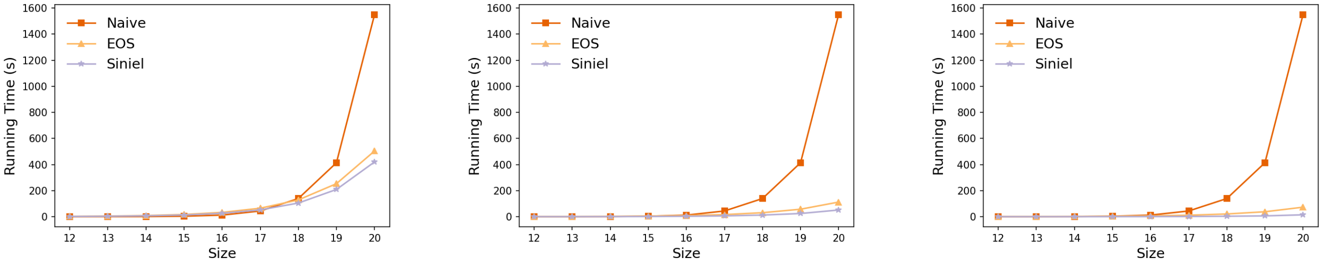
$$([\vec{p}_k]_i(Q), \pi_{[\vec{p}_k]_i}) := KZG.Open(ck, [\vec{p}_k]_i, C_{[\vec{p}_k]_i}, Q)$$

2. Aggregate all shares of proofs $C_{\vec{p}_k} := Output_G(C_{[\vec{p}_k]_i})$, $\vec{p}_k(Q) := Output_F([\vec{p}_k]_i(Q))$, and $\pi_{\vec{p}_k} := Output_G(\pi_{[\vec{p}_k]_i})$.
3. Check whether $\mathbb{V}$ outputs accept and $KZG.Verify(ck, \pi_{\vec{p}_k}, C_{\vec{p}_k}, \vec{p}_k(Q), Q) = 1$.
If at least $t + 1$ workers do not agree with the verification, then the protocol aborts. Otherwise, for each $k \in [K]$, assemble and send the proof $(C_{\vec{p}_k}, \vec{p}_k(Q), \pi_{\vec{p}_k})$ to D.

Fig. 8: Siniel Construction

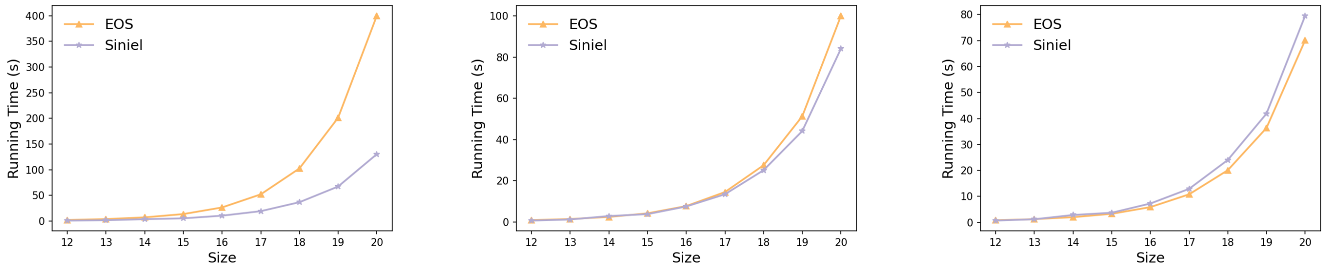
the cost, and thus sacrifice the efficiency of the entire system. A small number of workers is often optimal as it balances efficiency and security without overburdening the delegator. Therefore, we set the number of workers to three, which is the minimum required for an honest majority setting.

Implementation. We use the arkworks library [29] as a baseline to implement Siniel and EOS. The Marlin code is open-sourced. All implementations are written in Rust. Different from the basic operations including addition, multiplication, and MSM in the original arkworks library, all operations in Siniel and EOS are over shares. Therefore, we modify the arkworks library to accommodate our settings as follows. First, we add codes for Shamir secret sharing to share the private witness and reconstruct the secret. These codes include operations over scalar elements and group elements. Second, we add codes for the multiplication of two shares. These codes include generating Beaver triples and executing the Beaver multiplication protocol. Third, based on the codes for share multiplication, we add codes for shared polynomial arithmetizations including addition and multiplication.



(a) 10MBps (b) 100MBps (c) 1000MBps

Fig. 9: Running time in seconds of the naive Marlin prover, EOS delegator, and Siniel delegator under different network bandwidths (10MBps, 100MBps, 1000MBps) and circuit sizes (from 2^{12} to 2^{20}).



(a) 10MBps (b) 100MBps (c) 1000MBps

Fig. 10: Running time in seconds of the EOS worker and Siniel worker under different network bandwidths (10MBps, 100MBps, 1000MBps) and circuit sizes (from 2^{12} to 2^{20}).

Evaluation. The delegator is an AWS c5a.4xlarge instance with 32 GB of RAM and an AMD EPYC 7R32 CPU at 3.3 GHz with 16 cores, while all three workers are AWS c5a.8xlarge instances with 64GB of RAM and an AMD EPYC 7R32 CPU at 3.3 GHz with 32 cores. We measure the private delegation of the Marlin prover with different circuit sizes (from 2^{12} to 2^{20}). We evaluate the performance with different bandwidths (10MBps, 100MBps, 1000MBps). We evaluate both EOS and Siniel ten times and take the average as the final result. The round-trip latency is only for the setup phase, and we did not include it in the time for online proof-generation. Therefore, the total running time of Siniel and EOS includes the online communication time as well as the computation time.

B. Cost of the Delegator

In this subsection, we measure the running time of the delegator, as shown in Fig. 9 under different network bandwidths (10MBps, 100MBps, 1000MBps). The naive Marlin is the baseline that a delegator directly runs the Marlin computation without outsourcing its computation to several workers.

The running time of the Siniel delegator is much faster than that of the EOS delegator and naive Marlin prover. For example, for circuit size 2^{20} and network bandwidth 10MBps, Siniel completes in 419 seconds compared to the naive Marlin's 1549 seconds, an approximate 3-fold speedup, and compared to the EOS's 503 seconds, an approximate 1.2-fold speedup. As the network bandwidth grows, the Siniel delegator spends less time than the EOS delegator. This is because the running time of the EOS delegator consists of two parts, the preprocessing phase, in which the delegator distributes the share of the private witness, and the online phase, in which the delegator engages the computation with workers to check the PIOP computation, while that of the Siniel delegator only consists of the preprocessing phase. In EOS, the delegator should engage in the MPC computation for the PIOP consistency checker and wait for workers to finish the computation. Meanwhile, the Siniel delegator can **entirely outsource the zkSNARK computation** to several workers without any further interaction. The communication overhead of Siniel is a major bottleneck under low network bandwidth since the communication overhead of Siniel consists of Beaver triples, authentication keys, authentication tags, and shares of the private witness, while that of EOS consists of shares of the private witness.

First, the Siniel delegator distributes tags and keys to workers to help them verify the correctness of zkSNARK computation. Compared with EOS, no further interaction is needed for the delegator during the online phase. On the other hand, EOS adopts a technique to reduce the communication overhead in which a delegator sends a full share of witness to a single party while others get PRG seeds. This reduces communication overhead from $O(n|\vec{w}|)$ to $O(|\vec{w}|)$, in which n is the number of workers and $|\vec{w}|$ is the length of witness.

However, this technique is only applicable for additive secret sharing rather than Shamir secret sharing. In addition, offline time includes the time to generate shares of the private witness and send the shares to all workers, and it is mainly determined by the worker receiving the largest amount of communication. In EOS, all workers must wait until one worker receives the entire share before the online phase, while in Siniel, D concurrently sends each share along with corresponding authentication keys and tags to each worker.

C. Cost of the Worker

In this subsection, we evaluate the running time of the worker in the online phase, as shown in Fig. 10. The experimental results show that the Siniel workers consume less time than EOS workers in the low bandwidth environment (i.e., 10MBps), while Siniel workers consume more time in the high bandwidth environment (i.e., 1000MBps). For example, for the circuit size 2^{20} and 10MBPS network bandwidth, Siniel and EOS spend around 130 and 400 seconds, respectively, while Siniel and EOS respectively spend around 79 and 70 seconds for the circuit size 2^{20} and 1000MBPS network bandwidth.

The running time of the EOS worker consists of online MPC computation as well as the time to wait for delegator verification. As the verification for the delegator is very fast, the communication overhead becomes a major bottleneck in low network bandwidth (i.e., 10MBps). On the other hand, in Siniel, the verification for the delegator is replaced with wcc and pcc executed by the workers. In the high network bandwidth, the time to transfer data in the online phase can be neglected. Therefore, the communication is no longer the bottleneck for EOS workers.

Compared with EOS, Siniel should execute two additional consistency checkers to ensure the correctness of zkSNARK computation. The communication overhead of wcc and pcc is negligible as it only consists of a few elements. Thus, both checkers are not limited to the network bandwidth.

Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2701400, and in part by the National Natural Science Foundation of China under Grant 62132005, Grant 62172162, and Grant 62172161. Guomin Yang is supported by the Lee Kong Chian Fellowship awarded by Singapore Management University. This work was also supported by AXA Research Fund.

References

[1] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy (SP)*, pages 459–474, 2014.

[2] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In *Financial Cryptography and Data Security (FC)*, pages 81–98, 2017.

[3] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[4] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016.

[5] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby x. 509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE Symposium on Security and Privacy (SP)*, pages 235–254, 2016.

[6] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In *IEEE Symposium on Security and Privacy (SP)*, pages 790–808, 2023.

[7] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Advances in Cryptology (EUROCRYPT)*, pages 677–706, 2020.

[8] Nicholas Ward, Alessandro Chiesa, Pratyush Mishra, Yuncong Hu, Noah Vesely, and Mary Maller. Marlin: Preprocessing zksnarks with universal and updatable srs. pages 738–768, 2020.

[9] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[10] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology (EUROCRYPT)*, pages 305–326, 2016.

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

[12] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-snarks zero-knowledge proofs for distributed secrets. In *USENIX Security*, pages 4291–4308, 2022.

[13] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[14] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. pages 618–646, 2020.

[15] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Annual International Cryptology Conference (CRYPTO)*, pages 618–646, 2020.

[16] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: efficient and scalable mpc in the honest majority setting. In *Annual International Cryptology Conference (CRYPTO)*, pages 244–274, 2021.

[17] Sanjam Garg, Aarushi Goel, and Mingyuan Wang. How to prove statements obliviously? *Cryptology ePrint Archive*, 2023.

[18] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zksaas: Zero-knowledge snarks as a service. pages 4427–4444, 2023.

[19] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zksnark provers. In *USENIX Security*, pages 6453–6469, 2023.

[20] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual International Cryptology Conference (CRYPTO)*, pages 643–662, 2012.

[21] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *USENIX Security*, pages 675–692, 2018.

[22] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3003–3017, 2022.

[23] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE Symposium on Security and Privacy (SP)*, pages 859–876, 2020.

[24] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *IEEE Symposium on Security and Privacy (SP)*, pages 35–35, 2024.

[25] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology (ASIACRYPT)*, pages 177–194, 2010.

[26] Benny Applebaum, Oded Nir, and Benny Pinkas. How to recover a secret with o(n) additions. In *Advances in Cryptology (CRYPTO)*, pages 236–262, 2023.

[27] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.

[28] Jin Li and Jinzheng Qu. Barycentric lagrange interpolation collocation method for solving the sine–gordon equation. *Wave Motion*, 120:103159, 2023.

[29] A. developers. arkworks, 2020. In *https://github.com/arkworks-rs*.

APPENDIX

A. SECURITY DEFINITIONS FOR CONSISTENCY CHECKERS

First, we define the security model of the witness consistency checker, as shown in Fig. 11. In the witness consistency checker, all workers jointly verify the consistency of commitments to the input shares (i.e., shares of the private witness). In the ideal world, a trusted third party receives each commitment cm_i to $[\vec{w}]_i$ held by the worker P_i. If cm and $\vec{w}$ are consistent with $\mathbf{w}$ and $\vec{w}(\alpha) = \mathbf{w}(\alpha)$, then the trusted third party informs to all workers that the verification passes.

Formally, the protocol for the witness consistency checker π securely implements the ideal functionality F_{wcc}, if it is a protocol between n workers $[P_i]_{i=1}^n$ such that for every efficient adversary $\mathcal{A}$ in the real world, there exists a simulator $\mathcal{S}$ in the ideal world, such that the view in the real world is computationally indistinguishable from the view in the ideal world.

1. For each $i \in [n]$, receive $([\vec{w}]_i, [\vec{w}]_i(\alpha), cm_i, [\alpha]_i)$ from each worker P_i.
2. Aggregate cm_i, $[\vec{w}]_i$, $[\vec{w}]_i(\alpha)$, $[\alpha]_i$ and gets cm, $\vec{w}$, $\vec{w}(\alpha)$ and α.
3. If cm and $\vec{w}$ are consistent with $\mathbf{w}$ and $\vec{w}(\alpha) = \mathbf{w}(\alpha)$, it outputs 1 to all parties, otherwise, it outputs 0.

Fig. 11: Witness Consistency Checker Ideal Functionality F_{wcc}

Second, we define the security model of the PIOP consistency checker, as shown in Fig. 12. In the PIOP consistency checker, all other workers jointly verify that the PIOP computation executed by P_i is correct. In the ideal functionality, a trusted third party receives the prover polynomial $f(X)$, corresponding authentication tag $\tau_{\vec{f}}$, and corresponding commitment $comm$ from P_i, in which $\vec{f}$ is the coefficients of the polynomial $f(X)$. If the authentication tags $\tau_{\vec{f}}$ are consistent with the prover polynomials $f(X)$ (i.e., $f(X)$ is correctly computed with a PIOP circuit C), $f(X)$ is consistent with the input share $[\vec{w}]_i$, and the commitment $comm$ to $f(X)$ is correct, then the trusted third party informs other parties that the verification succeeds.

Formally, the protocol for the PIOP consistency checker π securely implements the ideal functionality F_{pcc}, if it is a protocol between a worker P_i and all other workers $[P_j]_{j=1 \cap j \neq i}^n$ such that for every efficient adversary $\mathcal{A}$ in the real world, there exists a simulator $\mathcal{S}$ in the ideal world, such that the view in the real world is computationally indistinguishable from the view in the ideal world.

1. Receive $(ck, \mathbf{x}, [\vec{w}]_i, \tau_{[\vec{w}]_i}, f(X), \tau_{[\vec{f}]_i}, comm)$ from P_i.
2. Receive $([\mu]_j, [v_{[\vec{f}]_i}]_j)$ from each other worker P_j.
3. Reconstruct μ and $v_{[\vec{f}]_i}$.
3. Output 1 to all parties P_j $(i \neq j)$, if
(1). For each $f_k \in \vec{f}$, $\tau_{[f_k]_i} = \mu \cdot [f_k]_i + v_{[f_k]_i}$,
(2). $comm$ is consistent with $f(X)$, and
(3). $f(X)$ is consistent with $[\vec{w}]_i$,
otherwise, output 0.

Fig. 12: PIOP Consistency Checker Ideal Functionality F_{pcc}

1. Receive $(ck, \mathbf{x}, \mathbf{w})$ from D.
2. Compute $\pi \leftarrow \mathcal{P}(ck, \mathbf{x}, \mathbf{w})$.
3. Send $(ck, \mathbf{x}, \pi)$ to all workers (and hence to $\mathcal{S}$).
4. If at least $t + 1$ workers output **reject**, output $\perp$, otherwise, send π to D.

Fig. 13: Ideal Functionality F_{SNARK}

B. SECURITY DEFINITION FOR SINIEL

Siniel focuses on the honest majority with secure with abort setting. 'Honest majority' means that at least half of the workers do not behave maliciously, while 'secure with abort' means that the protocol halts if malicious behavior is detected. In addition, Siniel guarantees that the private witness $\mathbf{w}$ is completely hidden from all workers if no more than t workers collude ($n = 2t + 1$ in total). Malicious workers can arbitrarily deviate from the protocol.

Formally, let $ARG = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ be a SNARK for an NP relation $\mathbb{R} = \{(\mathbf{x}, \mathbf{w})\}$. Then π_{SNARK} is a delegation protocol for ARG and securely implements the ideal functionality F_{SNARK} as shown in Fig. 13, if it is a protocol between a delegator D and n workers $[P_i]_{i=1}^n$ such that for every $(\mathbf{x}, \mathbf{w}) \in \mathbb{R}$ and every efficient adversary $\mathcal{A}$ in the real world, there exists a simulator $\mathcal{S}$ in the ideal world, such that the view of the real execution is indistinguishable from the view of the ideal execution. We give a formal security proof in section E.

C. SECURITY PROOF FOR WITNESS CONSISTENCY CHECKER

Theorem 1: The protocol for the witness consistency checker as shown in Fig. 6 securely implements the ideal functionality F_{wcc} as shown in Fig. 11.

Proof. The simulator $\mathcal{S}$ for the corrupted workers does as follows:

- Recover α' with shares $[\alpha']_i$.
- Receive shares of evaluations and opening proofs $([\vec{w}]_i(\alpha'), \pi_{[\vec{w}]_i})$ from the adversarial workers.
- Compute

$$C_{\vec{w}} := Output_F(C_{[\vec{w}]_i})$$

$$\vec{w}'(\alpha') := Output_F([\vec{w}]_i(\alpha'))$$

$$\pi_{\vec{w}} := Output_G(\pi_{[\vec{w}]_i})$$

- Check whether
$$\vec{\hat{w}}'(\alpha') = \vec{\hat{w}}(\alpha')$$
$$KZG.Verify(\hat{ck}, \pi_{\vec{\hat{w}}}, C_{\vec{\hat{w}}}, \vec{\hat{w}}'(\alpha'), \alpha') = 1$$
- If the proof does not pass, it outputs 0. If the proof passes, run the KZG extractor to obtain $\vec{\hat{w}}$ inside the commitment $C_{\vec{\hat{w}}}$. If the extracted $\vec{\hat{w}}$ is the expected one, then output 1, otherwise, output 0.

We prove the indistinguishability between the real world and the ideal world as follows:

- Hybrid 0: The real protocol.
- Hybrid 1: This hybrid is identical to hybrid 0 except that α in the real execution is replaced with α' in the ideal execution. This hybrid is indistinguishable as it leaks nothing about the witness.
- Hybrid 2: This hybrid is identical to hybrid 1 except that shares of $\vec{w}$ in the real world are replaced with the shares of $\vec{\hat{w}}$ in the ideal world. With the property of the KZG commitment scheme, the commitment and opening proof do not leak any information about the plain witness. Therefore, this hybrid is indistinguishable from the previous one.
- Hybrid 3: This hybrid is identical to hybrid 2 except that $\mathcal{S}$ extracts the witness inside the commitment. With the knowledge soundness of the KZG polynomial commitment scheme, if the proof is not valid, then the extraction fails with a non-negligible probability. Therefore, this hybrid is indistinguishable from the previous hybrid.

D. Security Proof for PIOP Consistency Checker

Theorem 2: The protocol for the PIOP consistency checker as shown in Fig. 7 securely implements the ideal functionality F_{pcc} as shown in Fig. 12.

Proof. The simulator $\mathcal{S}$ for the corrupted P_i does as follows:

- If P_i is an honest party, send a random challenge β' to P_i, otherwise, send β' to the adversarial worker.
- For each $k \in S(i)$, receive $(f^{(k)}(\beta'), \tau_{f^{(k)}(\beta')}, \pi_k)$ from the adversarial worker.
- For each $k \in S(i)$, honestly update the authentication key $v_{f^{(k)}(\beta')}$, and check that
$$\tau_{f^{(k)}(\beta)} = \mu \cdot f^{(k)}(\beta) + v_{f^{(k)}(\beta)}$$
$$KZG.Verify(ck, \pi_k, com_{f^{(k)}}, f^{(j)}(\beta), \beta) = accept$$
- If the above verification passes, then run a KZG extractor to extract each $f^{(k)}(X)$ inside the commitment $com_{f^{(k)}}$.

We prove the indistinguishability between the real world and the ideal world as follows:

- Hybrid 0: The real protocol.
- Hybrid 1: This hybrid is identical to hybrid 0 except that $\mathcal{S}$ chooses another random challenge β' instead of β. This hybrid is indistinguishable from hybrid 0.
- Hybrid 2: This hybrid is identical to hybrid 0 except that $\mathcal{S}$ extracts the polynomial inside the commitment. With the knowledge soundness of the KZG polynomial commitment scheme, the extractor fails with a negligible probability. Therefore, this hybrid is indistinguishable from the hybrid 1.

E. Security Proof for Siniel

Theorem 3: Let $\mathbf{R} = \{\mathbf{x}, \mathbf{w}\}$ be an NP relation with the following components:

- PIOP=$(F, K, S, \mathbb{P}, \mathbb{V})$ is a PIOP for $\mathbf{R}$ satisfying completeness, knowledge soundness and zero-knowledge.
- KZG=$(Setup, Commit, Open, Verify)$ is a PCS satisfying completeness and knowledge soundness under AGM.
- SSS=$(Share, Recover)$ is a (t, n) threshold SSS.
- An ideal functionality F_{wcc} as shown in Fig. 11.
- An ideal functionality F_{pcc} as shown in Fig. 12.

Let $ARG = (\mathcal{G}, \mathcal{P}, \mathcal{V})$ be a zkSNARK for the relation $\mathbf{R}$ based on PIOP and KZG. Then Siniel as shown in Fig. 8 securely implements F_{SNARK} in the $F_{wcc}, F_{pcc}-$ hybrid model with corruption at most t workers ($n = 2t + 1$ workers).

Proof: The simulator $\mathcal{S}$ for the Siniel is shown in Fig. 14. We prove the indistinguishability between the ideal world and the real world as follows:

Input. From the view of corrupted parties, the received shares of $\vec{\hat{w}}$ are indistinguishable from random.

Linear Gates. No interaction is needed between workers. Therefore, no information is exchanged in the real world and the ideal world.

Multiplication Gates with Two Private Wires. We use the standard Beaver multiplication protocol plus the verification mechanism. Therefore, the adversary learns nothing about the shared values.

Random Oracle Gate. The programmed random oracle is chosen uniformly at random. Therefore, no adversary can distinguish them.

Reveal Gate. The inputs to 'Reveal' have two kinds of values: the scalar field and the group element. We analyze them as follows:

- Evaluations of Shared Prover Polynomials. In the ideal world, we reveal evaluations of shares of random polynomials, while in the real world, we reveal evaluations of shares of prover polynomials. Since PIOP is zero-knowledge, the views of both worlds are indistinguishable.
- Commitments to Shared Polynomials and Opening Proofs of the Evaluations. In the ideal world, we reveal shares of commitments to random polynomials and corresponding opening proofs, while in the real world, we reveal shares of commitments to the real prover polynomials and corresponding opening proofs. Since PIOP is zero-knowledge, the prover polynomials seem random. Therefore, the distribution of both worlds is indistinguishable.

Output Gate. 'Output' gate has two kinds of values: the $Output_F$ for the scalar field and $Output_G$ for the group element. We analyze them as follows:

Setup and Offline: For each $P_i \in Corr$ and each other party P_j, $\mathcal{S}$ does the following:
1. Sample a random $\hat{\tau} \in F$, and sets

$$\hat{ck} = (g, \hat{\tau} \cdot g, \hat{\tau}^2 \cdot g, \ldots, \hat{\tau}^t \cdot g)$$

2. Receive $\mathbf{x}$ and π from F_{SNARK}.
3. Set a simulated witness $\hat{\vec{w}} \leftarrow F^{|\vec{w}|}$, and compute

$$[\hat{\vec{w}}]_i \leftarrow SSS.Share(t, n, \hat{\vec{w}})$$

4. Set $\alpha' \leftarrow F$, compute $\hat{\vec{w}}(\alpha')$,

$$[\hat{\vec{w}}(\alpha')]_i \leftarrow SSS.Share(t, n, \hat{\vec{w}}(\alpha'))$$

$$[\alpha']_i \leftarrow SSS.Share(t, n, \alpha'))$$

5. Choose authentication keys $\mu^{(i)}$ and $v_{[\hat{\vec{w}}]_i}$, computes $\tau_{[\hat{\vec{w}}]_i} = \mu^{(i)} \cdot [\hat{\vec{w}}]_i + v_{[\hat{\vec{w}}]_i}$.
6. For each other worker P_j ($j \neq i$), generate shares of authentication key $([\mu^{(i)}]_j, [v_{[\hat{\vec{w}}]_i}]_j)$, send to adversarial parties $(\hat{ck}, \mathbf{x}, [\alpha']_i, [\hat{\vec{w}}(\alpha')]_i, [\hat{\vec{w}}]_i, \tau_{[\hat{\vec{w}}]_i})$ and to P_j $([\mu^{(i)}]_j, [v_{[\hat{\vec{w}}]_i}]_j)$.

PIOP Computation: $\mathcal{S}$ proceeds as follows:
1. $\mathcal{S}$ invokes the ideal functionality F_{wcc} as shown in Fig. 11.
2. For each $P_i \in Corr$, $\mathcal{S}$ does the following:
Simulate the protocol execution as follows:

- There is no interaction between workers for gates Add_F, Add_G, Mul_G, so computation proceeds locally according to Siniel.
- If at least one of the wires in Mul_F is public, proceed locally according to Siniel. Otherwise, evaluate it with a Beaver multiplication protocol shown in Fig. III-C.
- To evaluate the random oracle gate RO, read all outputs $\vec{O}$ of ρ. Add the mapping $in_i \rightarrow o_i$ to the programming μ, for each $o_i \in \vec{O}$.
- To evaluate $Reveal$, proceed the computation according to Siniel.

3. $\mathcal{S}$ invokes the ideal functionality F_{pcc} as shown in Fig. 12.

Proof Generation: $\mathcal{S}$ proceeds as follows:
1. Compute the query set Q, and sends Q to adversarial workers.
2. Receive all shares of evaluations and opening proofs of the prover polynomials from the adversarial workers.
3. Compute $Output_G$ over all shares of commitments to prover polynomials, and the opening proofs of evaluation, and $Output_F$ over all shares of evaluations of prover polynomials. Assemble these aggregated proofs π'.
4. If $V(\hat{ck}, \mathbf{x}, \pi') = 0$, send $\perp$ to F_{SNARK}.
Finally, if the protocol does not abort, send π' to D.

Fig. 14: Simulator $\mathcal{S}$ for Siniel

- $Output_F$. This gate as shown in Fig. IV is identical to $SSS.Recover$.
- $Output_G$. This gate as shown in Fig. IV is almost identical to $SSS.Recover$ except that it recovers the secret over group elements.

Leakage in the Verify Phase. We discuss the verification as follows:

- Verification for the witness. There is no leakage in this phase other than the validity of the witness. If this verification aborts, no information about the final proof will be revealed to D.
- Verification for the PIOP computation. There is no leakage in this phase other than the validity of this verification. If this verification aborts, no information about the final proof will be revealed to D.
- Verification for the final proof. There is no leakage in this phase other than the final proof and its validity. If this verification aborts, no information about the final proof will be revealed to D, since in the real world, D receives the final proof when at least $t+1$ workers agree with the verification, and in the ideal world, D receives the final simulated proof when at least $t+1$ workers send **accept** to F_{SNARK}

Rejection in the PIOP Computation Phase. If the adversary behaves honestly, with the correctness property of PIOP and KZG extractor, F_{pcc} will output 1. $\mathcal{S}$ also extracts the share of the private witness from F_{pcc}. The extracted share equals the simulated share with non-negligible probability.

Rejection in the Proof Generation Phase. If the adversary behaves honestly, with the correctness property of PIOP and KZG extractor, both verifications will pass. If the adversary does not follow the protocol, with the knowledge soundness of PIOP and KZG, the probability that the ideal world passes both verifications and the ideal world does not pass the verifications is negligible.

Probability of Rejection due to an Incorrect Output Proof. In both worlds, all workers check if the proof is correct. The knowledge soundness of zkSNARK ensures that the probability of the real check passing (D receives the proof) but the ideal check failing (D does not receive the proof) is negligible.