Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets

Alex Ozdemir *Dan Boneh*
{aozdemir,dabo}@cs.stanford.edu

Abstract

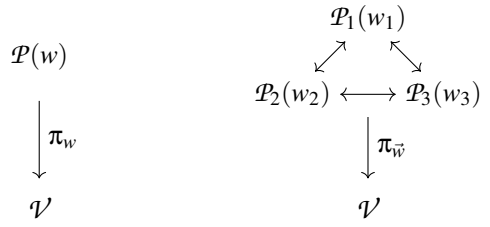
A zk-SNARK is a powerful cryptographic primitive that provides a succinct and efficiently checkable argument that the prover has a witness to a public NP statement, without revealing the witness. However, in their native form, zk-SNARKs only apply to a secret witness held by a single party. In practice, a collection of parties often need to a prove a statement where the secret witness is distributed or shared among them.

We implement and experiment with *collaborative zk-SNARKs*: proofs over the secrets of multiple, mutually distrusting parties. We construct these by lifting conventional zk-SNARKs into secure protocols among N provers to jointly produce a single proof over the distributed witness. We optimize the proof generation algorithm in pairing-based zk-SNARKs so that algebraic techniques for multiparty computation (MPC) yield efficient proof generation protocols. For some zk-SNARKs, optimization is more challenging. This suggests MPC "friendliness" as an additional criterion for evaluating zk-SNARKs.

We implement 3 collaborative proofs and evaluate the concrete cost of proof generation. We find that over a good network, security against a malicious minority of provers can be achieved with *approximately the same runtime* as a single prover. Security against $N-1$ malicious provers requires only a $2\times$ slowdown. This efficiency is unusual: most computations slow down by several orders of magnitude when securely distributed. It is also significant: most server-side applications that can tolerate the cost of a single-prover proof should also be able to tolerate the cost of a collaborative proof.

1 Introduction

Zero-knowledge succinct, non-interactive arguments of knowledge (zk-SNARKs) [22] are publicly verifiable proofs that the prover has secret data (a witness) that satisfies a certain public NP relation. The proof reveals nothing about the secret data other than its validity. zk-SNARKs have two key strengths. First, they are very general: there are zk-SNARKs that can prove any relation expressible as a bounded size arithmetic circuit. Second, they can be easily verified: proof size and verification time are sublinear in the amount of secret data (typically kilobytes and tens of milliseconds), and verification can be performed by anyone. However, there are two key limitations. First, generating the proof is expensive: typically thousands of times slower than checking the relation directly [82, 89, 97, 99]. Second, the secret data must be held by a single party: generating a proof when the secret data is distributed among multiple parties is not natively supported.



(a) Conventional zk-SNARKs concern one party's secrets (b) Collaborative zk-SNARKs concern multiple parties' secrets

Figure 1: Conventional and collaborative zk-SNARKs

Being limited to secrets that are held by a single party bars some applications. We describe some examples in Section 2.

Collarborative zk-SNARKs In this work we generalize zk-SNARKs to *collaborative zk-SNARKs*. That is, we generalize from public proofs about a secret w held by a single party, to public proofs about a secret $\vec{w} = (w_1, \ldots, w_N)$ distributed among N parties, where party i has w_i, for $i \in [N]$, as in Fig. 1. The proof generation process should reveal nothing new about $\vec{w}$ to a coalition of parties, other than the validity of $\vec{w}$.

A natural approach for constructing a collaborative zk-SNARK is as follows: start from a standard single-prover zk-SNARK, and run its proof generation algorithm as a *secure multi-party computation* (MPC) [45] among the N provers.

Recall that, informally, MPC allows N parties to compute a public function $f : \mathcal{X}^N \to \mathcal{Y}$ over secret inputs $x_1, \ldots x_N \in \mathcal{X}$, where party i has x_i for all $i \in [N]$. At the end of the protocol, all parties learn the output $y \leftarrow f(x_1, \ldots x_N)$, but learn nothing else about each others' inputs. MPC protocols work for any function f that can be expressed as an arithmetic circuit.

By taking the MPC function f to be the circuit representation of a zk-SNARK proof generator, the provers can use a generic MPC protocol to jointly generate the desired proof without revealing anything else about their secret inputs. One technicality is that proof generation is a randomized algorithm, but this is easily supported by MPC protocols.

Another issue is that any prover can choose not to participate in the MPC, thereby preventing the proof from being generated. Hence, collaborative zk-SNARKs only make sense in a settings where all N provers want to jointly generate a valid proof. This lets us use a slightly simpler kind of MPC,

called *MPC-with-aborts*, that does not guarantee output delivery. Regardless, using an MPC protocol to generate a proof ensures that no matter what a malicious set of provers does, it learns nothing about the secret data of the honest provers, other than the validity of the witness tuple $\vec{w}$.

As we will see, a direct application of this approach leads to very poor performance. Generating a standalone zk-SNARK proof is already computationally expensive. Running this entire process naively through a general purpose MPC protocol would likely result in unacceptable proving time. In particular, since MPC protocols and zk-SNARK provers are each thousands of times slower than their underlying functionality, their composition is likely to be millions of times slower.

Instead, we show that for some zk-SNARKs, proof generation lends itself to a very efficient MPC protocol. For these zk-SNARKs it is quite efficient to generate the proof distributively, without the provers leaking undesired information about their secrets to the public or to each other. This suggests MPC "friendliness" as a new criterion for evaluating zk-SNARKs. As we will see, some zk-SNARKs are MPC-friendly, while others are less so.

Our techniques We design optimized multi-prover protocols by using three ideas:

(1) Building on previous and new techniques, we apply MPC techniques based on secret-sharing *directly* to elliptic curves, making secure, distributed curve operations cheap.
(2) By using secret sharing over curve points we streamline key prover bottlenecks, such as multi-scalar multiplication, polynomial division, and Fourier transform.
(3) We employ an optimized MPC protocol for computing sequences of partial products. In particular, for field elements $x_1, \ldots, x_n$ we use an efficient MPC protocol from [4] to compute $p_j := \prod_{i=1}^{j} x_i$ for all $j \in \{1, \ldots, n\}$.

Ultimately, we adapt four zk-SNARKs to multiple provers: Groth16 [59], Plonk [48], Marlin [35], and Fractal [36].

Our results Our contributions are as follow.

- First, we formally define collaborative zk-SNARKs (§4).
- Next, we design four collaborative proofs by adapting four existing zk-SNARK provers into MPC protocols that exploit the algebraic nature of these provers: (§5):
  - Groth16 and Marlin give very efficient protocols.
  - Plonk results in a protocol with more communication between the provers, due to the larger number of multiplications during proof generation.
  - Fractal can achieve MPC efficiency, but at the expense of increasing the proof size and verification time by a factor of N.

  This list classifies these zk-SNARKs by their MPC-friendliness, from most to least friendly.
- We implement our Groth16-, Marlin-, and Plonk-based protocols. We do so by lifting a single-prover code base called `arkworks` [43] and making it support multiple provers. The key implementation technique is to replace `arkworks`' low level implementation of *fields and curves* with *secret shares* that implement secure multi-party protocols for these types (§6).
- Finally, we evaluate our implementation. We find that with a good network, protocols secure against a malicious minority of provers run in essentially the same time as a single-party proof. Protocols secure against a malicious majority run in twice the time of a single-party prover (§7). These results are unusual in that MPC typically incurs a significant overhead. These results also show that collaborative proofs are practical.
- Since communication costs dominate in low-capacity networks, we give an $\Omega(n)$ lower bound on the communication needed to build a collaborative proof.

Publicly auditable MPC Collaborative zk-SNARKs give an efficient construction for a cryptographic primitive called *publicly auditable MPC* (PA-MPC) [7]. A PA-MPC protocol is an MPC that also produces a proof by which the public can verify that the computation was performed correctly with respect to commitments to the inputs. Classic PA-MPC constructions [7] have proofs of linear size. Collaborative zk-SNARKs yield PA-MPC with proofs of constant size. We discuss further in Sections 2 and 4.1. Concurrent work [65] builds on a similar idea.

2 Example applications

Collaborative zk-SNARKs come up naturally in many settings, both for real world applications and for conceptual ones. In this section we briefly survey a number of situations where they are needed.

(1) Healthcare statistics Healthcare providers provide services (medication, operations, etc.) to millions of patients every day. Their actions concern the intimate details of individuals' well-being and **must not** be publicly revealed. Yet, the public benefits from understanding costs in the healthcare system, in aggregate. For example, a recent investigation revealed that many hospitals routinely charge uninsured patients the highest rates [46].

The challenge is to compute aggregate statistics over sensitive data in a fashion that certifies the accuracy of the aggregates, despite the participants' incentives to omit data or mis-aggregate.

Collaborative proofs yield a clean solution. First, every healthcare provider publishes a short Merkle commitment to the list of services that it provided. Second, regulators can request aggregate statistics over the set of services provided by all healthcare providers. The requested statistic is defined by an arithmetic circuit that computes the statistic. Third, providers can compute the aggregate statistic via an MPC among the providers, *and* publish a collaborative proof that

the claimed result is consistent with all of their local commitments. This protects both the privacy of patient data and ensures the integrity of the aggregate.

The Merkle commitments to the underlying data are crucial for integrity. If at a later time there is a dispute over the accuracy of the aggregate statistic, a forensics investigator can request the providers to open their local Merkle commitments. A provider that committed to incorrect data will immediately be caught and penalized, and this provides a strong incentive for providers to commit to accurate data (refusing to open a commitment could similarly carry a penalty). Patients are also incentivized to check (via a Merkle proof) that the provider accurately committed to the service they received.

Overall, the collaborative proof that the computed aggregate is consistent with the committed data ensures accuracy of the aggregate while maintaining data privacy.

(2) Computing credit scores The three US credit bureaus collect financial information from many institutions and use that data to compute a credit score for all US individuals. Credit scores are used to make credit related decisions such as who gets a loan. When a credit bureau is hacked [42], everyone's information is exposed.

Collaborative proofs provide a very different solution for computing credit scores. Let $A_1, \ldots, A_N$ be N institution whose data is used to compute a person's credit score. Each A_i could publish a short Merkle commitment to the entirety of its local dataset. When Alice needs her credit score, the N institutions engage in an MPC to compute her score based on their local data, and collaboratively construct a proof that the score is consistent with the public commitments. Alice is given her score s and the short collaborative proof π. She can present (s, π) to a lender, who can validate π with respect to the public commitments to ensure that Alice's score is correct.

A_i's commitment to its local data ensures that if there is a dispute over Alice's credit score, her leaves of the Merkle commitment can be opened. If an incorrect credit score is computed, there is enough information to identify the cause. This is a more satisfying guarantee than simply asking the institutions to certify the computed credit score by signing it.

A concrete example. Consider a lender that is reviewing a loan application. The lender wants to know the sum of the applicant's credits and debits with the banks where the applicant has an account. The applicant could request a collaborative proof from its banks to prove that the difference between it debits (assets) and it credits (debts) is larger than some threshold T. In Section 7 we evaluate the time required to generate this proof.

(3) Private audits of multiple parties The global financial system can be modeled as a transaction graph: every account is a node, and there is an edge from u to v if account u issued a payment to account v. Financial regulators often need to answer the following question: did account u at Bank A pay account v at Bank B, either directly or via intermediate accounts? If the entire graph were stored at a single location this would be easy to answer: the entity that holds the graph could commit to the graph and provide a succinct proof that there is no chain of transactions $u \to a_1 \to \cdots \to a_\ell \to v$.

In reality, the transaction graph is distributed and siloed across a number of institutions. Each financial institution only sees the portion of the graph that touches accounts under its control. Yet, the regulator needs an answer to questions about the entire graph. Previous work applies MPC techniques to this problem [30, 77, 87], but the results are not verifiable by a third party (i.e., the regulator or the public).

Collaborative proofs provide a clean solution. First, every bank commits to its local view of the graph. They then collaboratively generate a succinct proof that there is no path from u to v in the union of their graphs. More generally, this applies to any query about the global graph that can be computed using an arithmetic circuit of reasonable size. The banks can collaboratively compute an answer to the query, along with a proof that the answer is consistent with each bank's commitment to its local view of the graph. In case of a later dispute, the commitments can be opened to identify who is at fault.

Publicly auditable MPC The three applications above are special cases of *publicly auditable MPC* (PA-MPC) [7]: a general primitive that extends secure MPC to produce a proof that can be independently checked to verify that the computation was performed correctly. The auditor need not participate in this computation, yet it can still check that the claimed output is consistent with commitments to the inputs.

We will see in Section 4.1 that collaborative proofs yield PA-MPC with proof size and verification time sublinear in the circuit size.

Reducing the number of parties Some of the applications in this section could involve many online parties. As we will see (§7), this can be expensive in communication. However, by having each data provider secret share their data among a small set of non-colluding servers, we can make the cost independent of the number of data providers. Furthermore, only the small set of non-colluding servers need to be simultaneously online. A similar approach has been successfully used in MPC-based auctions [24].

3 Background

Let $\mathbb{F}_p$ denote the finite field of integers mod p, a prime. We omit p when unambiguous. Let $\mathbb{G}$ be an *additive cyclic group* of prime order q, group operation $+$, inverse operation $-$, identity 0, and generator g.

Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be cyclic groups of prime order q with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable and non-degenerate *pairing*, so that

$$e(\alpha \cdot h_1,\ \beta \cdot h_2) = \alpha\beta \cdot e(h_1, h_2)$$

for all α, β in $\mathbb{F}_q$ and all $h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$.

3.1 Multi-party computation

Let $f : \mathcal{X}^N \to \mathcal{Y}$ be a function and let $P_1, \ldots, P_N$ be N parties such that P_i has input $x_i \in \mathcal{X}$. A multi-party computation (MPC) for f is a protocol Π among the parties which computes and reveals $y := f(x_1, \ldots, x_N)$. Informally, Π is secure if it reveals nothing except y, and it requires malicious parties to chose their inputs independently from the honest parties. In more detail, Π is secure against t dishonest parties if for every efficient adversary $\mathcal{A}$ corrupting t parties, there is an efficient *simulator* $\mathcal{S}$ which can produce a fake protocol transcript from the output and the inputs of those t parties, that is indistinguishable from the view of those parties in a real protocol execution. We refer the reader to [72] for the complete definition.

Many protocols are secure with one flaw: aborts. That is, an adversary can cause the protocol to abort thereby denying the output to the honest parties. Many protocols achieve security-with-abort (or better) assuming an honest majority of parties [9, 10, 20, 25, 34, 39, 40, 49, 50, 52, 55, 57, 73, 80, 85] or even in the presence of a single honest party [41, 53, 63, 71, 74, 75, 76, 79]. The latter is called the *dishonest majority* setting.

Arithmetic circuit MPC We build on MPC protocols that can securely compute any function defined as a bounded size *arithmetic circuit*. An arithmetic circuit is a directed acyclic graph of gates and wires. Wires carry values from a finite field (e.g., $\mathbb{F}_p$). Input wires can be public or private to a single party. Gates are binary and perform multiplication or addition. Certain wires are designated as outputs.

The two protocols we build on—SPDZ [41] and GSZ [40, 57]—are based on secret sharing. They evaluate the circuit by representing each wire's value using a k-out-of-N *secret-sharing*: N tokens distributed among the parties such that no $k-1$ tokens reveal the true value, but any k tokens do. We use $[y]$ to denote that y is shared using a secret sharing scheme.

- SPDZ [41] uses *additive secret sharing* where $y \in \mathbb{F}$ is represented by random shares $y_1, \ldots, y_N$ that sum to y. All N shares are required to recover y.
- GSZ [40, 57] uses *Shamir secret sharing* [90] where $y \in \mathbb{F}$ is shared among N parties as $y_1, \ldots, y_N$ such that the points $(1, y_1), \ldots, (N, y_N)$ lie on a polynomial $p(X)$ of degree at most d, with $p(0) = y$. Any $d+1$ points suffice to reconstruct the polynomial and y.

3.2 zk-SNARKs

Informally, a *proof* for a *relation* $\mathfrak{R}$ is a protocol between a prover $\mathcal{P}$ and an efficient verifier $\mathcal{V}$ by which $\mathcal{P}$ convinces $\mathcal{V}$ that $\exists w : \mathfrak{R}(x, w) = 1$, where x is a called an *instance*, and w a *witness* for x. If the proof comprises a single message from $\mathcal{P}$ to $\mathcal{V}$, it is said to be non-interactive and has syntax:

- $\mathsf{Setup}(\mathfrak{R}, 1^\lambda) \to \mathsf{pp}$: setup public parameters for $\mathfrak{R}$.
- $\mathsf{Prove}(\mathsf{pp}, x, w) \to \pi/\perp$: if $(x, w) \in \mathfrak{R}$, output a proof π, otherwise $\perp$.
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0, 1\}$: check a proof.

Generally, proofs support a class of relations (e.g., bounded size arithmetic circuits). The supported class must also define the *size* of a relation—denoted $|\mathfrak{R}|$—the time needed to evaluate it (e.g., the number of gates in the circuit). In the next subsection we discuss common classes of relations.

A zk-SNARK is a proof with the following properties [22]:

- *Completeness:* If $\mathfrak{R}(x, w) = 1$, then an honest $\mathcal{P}$ convinces $\mathcal{V}$ except with negligible probability. If $\mathfrak{R}(x, w) = 0$, Prove outputs $\perp$.
- *Zero knowledge:* Informally, a proof reveals nothing about the witness w.
- *Knowledge soundness:* Informally, for every $\mathcal{P}^\star$ there exists an efficient algorithm, called an *extractor*, such that whenever $\mathcal{P}^\star$ convinces $\mathcal{V}$ that $\exists w : \mathfrak{R}(x, w) = 1$, the extractor can interact with $\mathcal{P}^\star$ and output w such that $\mathfrak{R}(x, w) = 1$ [12].
- *Succinctness:* proof size and verification time are $o(|\mathfrak{R}|)$.

When soundness is computational, the protocol is called an *argument*, and referred to as a (z)ero-(k)nowledge (S)uccinct (N)on-interactive (AR)gument of (K)nowledge

R1CS format Different zk-SNARKs represent the relation $\mathfrak{R}$ in different ways: binary arithmetic circuits [48], quadratic arithmetic programs [82, 88], AIR [14], low-depth circuits [23, 37, 54, 92, 95, 96, 98, 101, 102] and more [18, 19, 51, 82]. We focus on proofs for rank-1 constraint systems (R1CS); these are the most widely used, and are the interface to many state-of-the-art proof systems [16, 35, 59, 60].

For R1CS, x is represented as $\mathbf{x} \in \mathbb{F}^\ell$, w is represented as $\mathbf{w} \in \mathbb{F}^{m-\ell}$, and the relation itself is defined by three matrices: $A, B, C \in \mathbb{F}^{n \times m}$ such that $A\mathbf{a} \circ B\mathbf{a} = C\mathbf{a}$ where $\mathbf{a} := \mathbf{x} \,||\, \mathbf{w}$ and $\circ$ is the Hadamard (element-wise) product. This $\mathbf{a}$ is said to be a satisfying assignment for the R1CS relation (n, ℓ, m, A, B, C). R1CS generalizes arithmetic circuit satisfiability. The matrices are assumed to have only $\Theta(n)$ non-zero entries, and $|\mathfrak{R}|$ is defined to be n.

The R1CS formalism introduces an additional challenge: each relation $\mathfrak{R}$ must be compiled to an R1CS relation $R := (n, \ell, m, A, B, C)$. Additionally, the compiler produces a witness extension procedure Extend that maps any satisfying $(x, w) \in \mathfrak{R}$ to a satisfying assignment $\mathbf{a} \in \mathbb{F}^n$ for R. Thus, when writing a proof, the prover $\mathcal{P}$ operates in two steps:

- first, $\mathcal{P}$ extends the witness w to a satisfying assignment $\mathbf{a} \in \mathbb{F}^n$ for the R1CS relation R,
- second, $\mathcal{P}$ builds an argument of knowledge for a satisfying assignment for the R1CS relation R.

The first step is relation-specific, non-cryptographic, and typically inexpensive. The second step is relation-generic, crypto-

Definition 1. *Collaborative zk-SNARK for a relation $\mathfrak{R}$, secure against t malicious provers:*
Let $p(\lambda)$ be a polynomial bound on the collaborative proof's total communication, and $\mathcal{U}(\lambda)$ be the set of functions from $\{0,1\}^{\leq p(\lambda)}$ to $\{0,1\}^\lambda$. In the random oracle model, a collaborative zk-SNARK for a relation $\mathfrak{R}$ that is secure against t malicious provers is a collaborative proof $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ with the following properties:

- *Completeness: For all $(x, \vec{w}) \in \mathfrak{R}$, the following is negligible:*

$$\Pr\left[\mathsf{Verify}^H(\mathsf{pp}, x, \pi) = 0 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(\mathfrak{R}, 1^\lambda) \\ \pi \leftarrow \Pi^H(\mathsf{pp}, x, \vec{w}) \end{array}\right]$$

- *Knowledge soundness: For all x, for all sets of efficient algorithms $\vec{\mathcal{P}} = (\mathcal{P}_1^*, \ldots, \mathcal{P}_N^*)$, there exists an efficient extractor Ext with*

$$\Pr\left[(x, \vec{w}) \in \mathfrak{R} : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(\mathfrak{R}, 1^\lambda) \\ \vec{w} \leftarrow \mathsf{Ext}^{H, \vec{\mathcal{P}}^H}(\mathsf{pp}, x) \end{array}\right] \geq \Pr\left[\mathsf{Verify}^H(\mathsf{pp}, x, \pi) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(\mathfrak{R}, 1^\lambda) \\ \pi \leftarrow \vec{\mathcal{P}}^H(\mathsf{pp}, x) \end{array}\right] - \varepsilon$$

  for negligible ε. Here, $\mathsf{Ext}^{H, \vec{\mathcal{P}}^H}$ denotes that Ext has oracle access to H and may re-run the collection of provers $\vec{\mathcal{P}}(\mathsf{pp}, x)$, reprogramming the random oracle H each time, and receiving only the final output produced by $\vec{\mathcal{P}}$.

- *Succinctness: Proof size and verification time are $o(|\mathfrak{R}|)$.*

- *t-zero-knowledge: For all efficient $\mathcal{A}$ controlling $k \leq t$ provers: $\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}$, there exists an efficient simulator Sim such that for all $x, \vec{w}$, and for all efficient distinguishers D,*

$$\left|\Pr\left[D^{H[\mu]}(\mathsf{tr}) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(\mathfrak{R}, 1^\lambda) \\ b \leftarrow \mathfrak{R}(x, \vec{w}) \in \{0,1\} \\ (\mathsf{tr}, \mu) \leftarrow \mathsf{Sim}^H(\mathsf{pp}, x, w_{i_1}, \ldots, w_{i_k}, b) \end{array}\right] - \Pr\left[D^H(\mathsf{tr}) = 1 : \begin{array}{l} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(\mathfrak{R}, 1^\lambda) \\ \mathsf{tr} \leftarrow \mathsf{View}_{\mathcal{A}}^H[x, \vec{w}] \end{array}\right]\right|$$

  is negligible, where tr is a transcript, $\mathsf{View}_{\mathcal{A}}^H[x, \vec{w}]$ denotes the view of $\mathcal{A}$ when provers $\mathcal{P}_1, \ldots, \mathcal{P}_N$ interact with input x and witnesses $\vec{w}$ (the honest provers follow Π, but dishonest ones may not), μ is a partial function from the domain of H, and $H[\mu]$ maps x to $\mu(x)$ if it $x \in \mathsf{Domain}(\mu)$ and $H(x)$ otherwise. Thus, $H[\mu]$ is the random oracle H after it has been re-programmed by Sim at a few points.

graphic, and typically the bottleneck. Thus, most proof systems research focuses on the second step, as do we.

4 Collaborative zk-SNARKs

A *collaborative proof* is a zk-SNARK where the witness is distributed among many provers.

A *non-interactive collaborative proof* for N provers and a relation $\mathfrak{R}(x, w_1, \ldots, w_N)$ is a tuple $(\mathsf{Setup}, \Pi, \mathsf{Verify})$:

- $\mathsf{Setup}(\mathfrak{R}, 1^\lambda) \to \mathsf{pp}$: setup public parameters.
- $\Pi(\mathsf{pp}, x, w_1, \ldots, w_N) \to \pi$: a protocol for N provers with private inputs w_1 through w_N that produces a proof.
- $\mathsf{Verify}(\mathsf{pp}, x, \pi) \to \{0,1\}$: check the proof.

Definition 1 states completeness, knowledge soundness, succinctness, and zero-knowledge for collaborative proofs in the random oracle model (generalized from [17]). For knowledge soundness, the extractor can program the oracle accessed by all provers participating in Π.

Zero-knowledge is more subtle. The single-prover definition of zero knowledge requires that the proof π for any $(x, w) \in \mathfrak{R}$ can be simulated without w. The restriction to *valid* witnesses is acceptable because an honest prover would never write a proof with an invalid witness. However, for collaborative proofs, each prover controls only a part of the witness, so no single prover can guarantee that $(x, \vec{w}) \in \mathfrak{R}$. Moreover, prover i can choose w_i arbitrarily, and then learn if the resulting combined witness w satisfies $(x, \vec{w}) \in \mathfrak{R}$. This is unavoidable, and reveals to prover i some information about the witnesses held by the other provers. We model this by explicitly providing the validity of the witness, denoted by $b \leftarrow \mathfrak{R}(x, \vec{w}) \in \{0,1\}$, to the simulator.

In sum, the zero-knowledge property applies to *all* witnesses, for up to t malicious provers, and guarantees that only the validity of the witness is revealed.

Knowledge soundness for a collaborative proof is also subtle: it establishes only that the provers have *distributed knowl-*

edge of the N witnesses. That is, it proves that if they pooled their information, they could determine the N witnesses.[1] It does **not** prove that $\mathcal{P}_i$ knows w_i. This limitation is intrinsic to the non-interactive syntax of the proof, which obscures from $\mathcal{V}$ the number of provers participating in the protocol. In some cases, the restriction might be partially circumvented by having $\mathfrak{R}$ itself establish knowledge of secrets believed (for reasons external to the proof) to be in the custody of different entities.

Theorem 1. *If* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is a zk-SNARK, and* Π *is an MPC for* Prove *that is secure-with-abort against up to t corruptions, then* $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ *is a collaborative zk-SNARK secure against t malicious provers.*

Proof sketch Completeness follows from the completeness of the zk-SNARK and the correctness of Π as an MPC. Knowledge soundness follows directly from the knowledge soundness of the zk-SNARK: from the perspective of an extractor, properties that hold for a malicious prover also hold for a malicious collection of provers. Succinctness follows from the succinctness of the zk-SNARK. Security-with-abort is sufficient for Π because the collaborative proof completeness definition does not cover malicious provers.

Zero-knowledge follows from the zero-knowledge of the zk-SNARK and the security of Π. In the case that $b = 1$ (the witness is valid), then the SNARK's zero-knowledge implies that π can be simulated from x. Then, the security of Π implies that the adversary's view can be simulated from π and the witnesses of the corrupted provers. If $b = 0$ (the witness is invalid), then the security of Π implies that adversary's view can be directly simulated from the witnesses of corrupted provers and $\perp$. □

Collaborative proofs for R1CS In this work, we consider non-interactive collaborative proofs for secret-shared R1CS witnesses. That is, we consider an R1CS instance $\mathbf{x}$ and provers $\mathcal{P}_1, \ldots, \mathcal{P}_N$ which hold shares $[\mathbf{w}]$ of an R1CS witness $\mathbf{w}$, such that $\mathbf{x}||\mathbf{w}$ is a satisfying assignment for an R1CS relation R. As in a single prover SNARK for R1CS, this approach requires a relation-specific witness extension protocol which maps the input and witness for $\mathfrak{R}$ to shares of the complete R1CS witness.

4.1 PA-MPC from collaborative proofs

Publicly-auditable MPC [7] (PA-MPC) extends MPC with a publicly-verifiable proof that the output of the MPC is correct with respect to commitments to each party's inputs. In Appendix D we define PA-MPC formally, and show that collaborative proofs can be used to construct a PA-MPC protocol. Specifically, we show how a suitable MPC protocol, a commitment scheme, and a collaborative proof give a PA-MPC protocol. The derived PA-MPC protocol outputs proofs whose size is about the same as those output by the collaborative proof. In particular, if the collaborative proof scheme outputs constant size proofs, then so does the derived PA-MPC protocol. This is a dramatic improvement over the classic construction [7] (where proof size and verification time are linear in the size of the evaluated circuit) and it is competitive with recent [94] and concurrent [65] work.

[1] This is the Halpern and Moses definition of distributed knowledge [61].

5 Design: efficient MPC for SNARK proofs

5.1 Review: two algebraic MPC protocols

We extend two secure MPC protocols for arithmetic circuit evaluation. The first, GSZ (§5.1), is secure if a majority of parties are honest. The second, SPDZ (§5.1), is secure if at least one party is honest. We first describe important features of both protocols. Then in Section 5.2, we discuss how to generalize these protocols to circuits over elliptic curve points and their scalars.

Honest majority MPC using GSZ The first protocol, GSZ was developed in [57], implemented in [55], and based on [25, 40, 50]. The protocol builds on Shamir secret-sharing. It evaluates an arithmetic circuit one gate at a time. Shamir shares can be added, shifted by public values, and scaled by public values using a single field operation and no communication between parties. Shares can be multiplied with an interactive protocol [40]. This protocol is not secure against malicious behavior [50], so all multiplication triples $([x], [y], [z])$ must be saved and checked for consistency, namely $xy = z$, before any share is opened. The *multiplication checking protocol* requires communication sub-linear in the number of triples [57].

Dishonest majority MPC using SPDZ The SPDZ protocol [41] uses additive sharing. Similar to Shamir shares, additive shares can be added, shifted by public values, or scaled by public values with at most one field operation and with no communication between parties. Multiplying shares requires an interactive protocol. SPDZ does not need to check multiplication triples because it maintains a shared *message authentication code* (MAC) $[\Delta x]$ for each shared value $[x]$ ($\Delta \in \mathbb{F}$, the MAC key, is unknown to all). Together, a share and its MAC form an *authenticated share*. The opening protocol for authenticated shares assures the integrity of the revealed value. However, since each operation on an authenticated share changes the MAC, the operation takes twice as long.

Both GSZ and SPDZ are typically split into a preprocessing phase (which can run before the circuit inputs are known) and an online phase (which runs after the inputs are known). We briefly discuss the relative costs of these phases in Section 7.

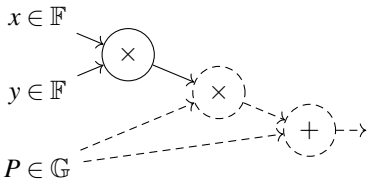


Figure 2: An elliptic curve circuit. The sub-circuit drawn with solid lines is an arithmetic circuit; the dashed extension includes curve operations

.

5.2 MPC using elliptic curve secret sharing

While SPDZ and GSZ natively support arithmetic over a finite field, the zk-SNARKs we consider perform many elliptic curve group operations, such as curve additions and multiplications by scalars.

Since elliptic curve operations decompose into operations over the curve's base field, there is an obvious way to perform them in an MPC. Unfortunately, this naive approach is *very* expensive: curve additions require tens of field operations, and scalar products require thousands. Furthermore, curve *additions* require multiple field *multiplications*, so even curve additions require rounds of communication.

Fortunately there is a better way: both the SPDZ and GSZ protocols can be adapted to operate directly on elliptic curves, where all secret sharing is done over the elliptic curve.

An *elliptic curve circuit* (Fig. 2) is a generalization of an arithmetic circuit over $\mathbb{F}_q$, adding support for computations over an elliptic curve group $\mathbb{G}$, with order q. In an elliptic curve circuit, wires can hold elliptic curve points from $\mathbb{G}$ or scalars from $\mathbb{F}_q$ and the gate set is extended to include curve addition and the multiplication of curve points by scalars. The multiplication of two curve points is prohibited.

Prior work shows [91] that SPDZ generalizes to elliptic curve circuits; we observe that GSZ does too. To start, observe that Shamir secret sharing can be used to share an elliptic curve point: the N shares are themselves elliptic curve points, and the secret is reconstructed by a weighted linear combination of sufficiently many shares. In Appendix A, we show how to apply GSZ's sub-protocols—the multiplication protocol of [40] and the multiplication checking protocol of [57]—to scalar-curve products.[2]

5.3 Commitments to shared polynomials

SNARK provers have many components. In this section and the next we show efficient MPC protocols for the core building blocks needed to generate a SNARK proof. We begin with an MPC protocol for a *polynomial commitment scheme*.

Polynomial commitments are a primitive—and bottleneck—in many recent SNARKs. SNARKs are built from polynomial commitments using the *polynomial interactive oracle proof* (P-IOP) paradigm [31, 35]. In a P-IOP, $\mathcal{P}$'s messages to $\mathcal{V}$ are *oracles* to polynomials which $\mathcal{V}$ accesses by querying for the outputs at various inputs. The polynomials are encodings of the SNARK witness, and $\mathcal{V}$ checks the SNARK relation using the results of its queries. To compile a P-IOP to a standard protocol, a polynomial commitment scheme is used to implement the oracles. For each oracle, $\mathcal{P}$ sends $\mathcal{V}$ a commitment to the underlying polynomial. For each query, $\mathcal{V}$ sends the input, $\mathcal{P}$ sends the output, and $\mathcal{P}$ and $\mathcal{V}$ engage in an *evaluation protocol* to validate the claimed evaluation. Informally, the evaluation protocol forces $\mathcal{P}$ to answer queries consistently with the underlying polynomial, so the compiled protocol is equivalent to the P-IOP.

A polynomial commit scheme is a tuple $(\mathsf{PC.Setup}, \mathsf{PC.Commit}, \mathsf{PC.Eval})$ where

- $\mathsf{PC.Setup}(d) \to \mathsf{pp}$: prepares public parameters supporting polynomials of degree up to d.
- $\mathsf{PC.Commit}(\mathsf{pp}, f, r) \to c$: computes a commitment c to an input polynomial f, using randomness r.
- $\mathsf{PC.Eval}(\mathsf{pp}, x, y, c; f, r) \to \{0,1\}$: a protocol where $\mathcal{P}$ convinces $\mathcal{V}$ that c commits to an f such that $f(x) = y$. Only $\mathcal{P}$ knows f and r.

We refer to [31, 35] for the security properties that a polynomial commitment must satisfy for the compiled P-IOP to be secure.

For a collaborative proof, the polynomial f is shared since it encodes the witness. However, the commitment to f, and various polynomial evaluations that are part of the proof are public. Hence, our goal is to generalize PC.Commit and PC.Eval to efficiently operate on a shared polynomial.

KZG commitments The KZG polynomial commitment [66] is among the most efficient known schemes and in particular has the smallest evaluation proofs. It is defined by four algorithms:

- $\mathsf{PC.Setup}(d)$: sample $\alpha \xleftarrow{\$} \mathbb{F}$; output
$$\mathsf{pp} \leftarrow (\alpha \cdot g_1,\ \ldots,\ \alpha^d \cdot g_1,\ \alpha \cdot g_2)$$
- $\mathsf{PC.Commit}(\mathsf{pp}, f)$: output $c = f(\alpha) \cdot g_1$, computed as $c = \sum_{i=0}^{d} f_i \cdot (\alpha^i \cdot g_1)$
- $\mathsf{PC.Prove}(\mathsf{pp}, f, x)$: compute the remainder and quotient
$$q(X), r(X) \leftarrow \big(f(X) - f(x)\big)/(X - x);$$
output $\pi = q(\alpha) \cdot g_1$, computed as $\pi = \sum_{i=0}^{d} q_i \cdot (\alpha^i \cdot g_1)$
- $\mathsf{PC.Check}(\mathsf{pp}, c, x, y, \pi)$: accept if
$$e(\pi,\ \alpha \cdot g_2 - x \cdot g_2) = e(c - y \cdot g_1,\ g_2)$$

[2] Generalizing SPDZ and GSZ to elliptic curves suffices for implementing SNARK provers, but the protocols can do even more: they apply to any bilinear operation, e.g., pairings and polynomial products.

Here, PC.Setup samples a trapdoor α, which must be discarded for security; for this reason the procedure is called a *trusted setup*. Together, PC.Prove and PC.Check form the evaluation protocol for the scheme. At a high level, the scheme's security follows from the fact that $X - x$ divides $f(X) - y$ if and only if $f(x) = y$. See [35, 47, 66] for security proofs and extensions of the scheme.

Generalizing KZG to multiple provers We now describe our adaptation of KZG to polynomials shared among multiple provers.

How can PC.Commit and PC.Prove be implemented for a secret shared f? First, we represent $[f]$ as a list of shared coefficients: $[f_0], [f_1], \ldots, [f_d]$. Let $f_0^{(i)}, f_1^{(i)}, \ldots f_d^{(i)}$ denote $\mathcal{P}_i$'s shares of those coefficients, defining a polynomial $f^{(i)}$, known only to $\mathcal{P}_i$. Remarkably, when using any linear sharing scheme [11] (including the additive and Shamir schemes we use) the provers can correctly create shares of commitments to (respectively, proofs for) f by locally committing to (respectively, proving evaluations for) their individual $f^{(i)}$ and interpreting the results as shares of the desired commitment (respectively, proof). More explicitly, Figure 3 holds our multi-party commitment and proving protocols: PC.Commit′ and PC.Prove′.

Here, we show that PC.Prove′ is correct; Appendix C shows security and PC.Commit′ has a similar proof. We argue generically over a linear secret sharing scheme [11]. A linear secret sharing scheme has *linear reconstruction*: for any sufficient subset of parties there is a linear combination of their shares that reconstructs the value. Let $\vec{\gamma} \in \mathbb{F}^N$ encode this linear combination, that is, for any value v with shares $v^{(i)}$, $v = \sum_{i=1}^N \gamma_i v^{(i)}$.

We begin with a lemma about the division in PC.Prove′ from Figure 3b.

Lemma 2. *For any shared polynomial $[f(X)]$ and public polynomial $d(X)$, if each party $\mathcal{P}_i$ computes $q^{(i)}, r^{(i)} \leftarrow \frac{f^{(i)}}{d}$, then $q^{(i)}$ and $r^{(i)}$ are shares of q and r such that $(q,r) = \frac{f}{d}$.*

Proof. Per each party's euclidean division, we have that $q^{(i)} \cdot d + r^{(i)} = f^{(i)}$. Summing the equations—scaled by the γ_i—yields:

$$\left(\sum_{i=1}^N \gamma_i q^{(i)}\right) \cdot d + \left(\sum_{i=1}^N \gamma_i r^{(i)}\right) = \sum_{i=1}^N \gamma_i f^{(i)} = f$$

Since each $r^{(i)}$ has degree less than d, their scaled sum does too, so $\sum_{i=1}^N \gamma_i q^{(i)}$ and $\sum_{i=1}^N \gamma_i r^{(i)}$ are a quotient-remainder pair for f divided by d. Since polynomial division is unique, these sums must be equal to q and r respectively. □

Claim 3. PC.Prove′ *is correct, that is* $\mathsf{PC.Prove}(\mathsf{pp}, f, x) = \mathsf{PC.Prove}'(\mathsf{pp}, [f], x)$.

Proof. It suffices to show that $\sum_{i=1}^N \gamma_i \pi^{(i)}$ is equal to $\mathsf{PC.Prove}(\mathsf{pp}, f, x)$. By Lemma 2, we have that for the $q^{(i)}$ that each $\mathcal{P}_i$ computes, $\sum_i \gamma_i q^{(i)} = q$. The we have that $\sum_{i=1}^N \gamma_i \pi^{(i)}$ is just $\sum_{i=1}^N \sum_{j=1}^d \gamma_i q_j^{(i)} \cdot \alpha^j \cdot g_1$. Swapping the order of the sum and simplifiying yields $\sum_{j=1}^d \sum_{i=1}^N \gamma_i q_j^{(i)} \cdot \alpha^j \cdot g_1 = \sum_{j=1}^d \gamma_i q^{(i)}(\alpha) = q(\alpha) = \mathsf{PC.Prove}(\mathsf{pp}, f, x)$. □

These simple and efficient protocols for commitments to shared polynomials are encouraging: the bottleneck in many SNARKs incurs little overhead in the multi-prover setting. In the next section we'll see that many other key SNARK operations have similarly efficient protocols.

5.4 Optimizing SNARK provers for MPC

In this subsection we discuss techniques for optimizing four SNARKs: Groth16 [59], Marlin [35], Plonk [48], and Fractal [36] to multiple-provers. Marlin and Plonk are defined generically over a polynomial commitment scheme. We instantiate them with the KZG-derived scheme from [35], adapted per the previous subsection. Here, we focus on other potential bottlenecks, especially those common to multiple provers. Figure 4 shows the operations we consider and the SNARKs they appear in.

Traditional bottlenecks: FFTs and MSMs Fast fourier transforms (FFTs) and multi-scalar multiplications (MSMs) are bottlenecks for most SNARK provers. MSMs are computations of the form $\sum_i \gamma_i \cdot g_i$ for scalars γ_i and curve points g_i; informally, provers use them to "commit" to the scalars γ_i. The fourier transform computes a matrix-vector product $M\vec{x}$, where M is derived from a root of unity ω. Provers use FFTs to convert between the coefficient- and evaluation-forms of polynomials.

Both the FFT and MSM are linear operators in the scalars γ_i and the vector $\vec{x}$ respectively. Thus, if other inputs (the curve points g_i and the root of unity ω) are public information (as they are in SNARKs) and a linear sharing scheme is used, each operator can be applied directly to each party's input shares, yielding shares of the output. Thus, FFTs and MSMs in a multi-prover SNARK reduce to performing the operation share-wise, with no communication required. Share-wise computation is cheap in an MPC: identical to the base cost for Shamir shares and twice that for authenticated shares.

We now discuss other operations which are not traditional bottlenecks, but might become bottlenecks in the multi-prover setting.

Polynomial division All provers do large polynomial divisions: a complex and non-linear operation that might have high cost in an MPC. However, within SNARKs, polynomial division always has a witness-independent—and therefore public—divisor. The KZG scheme is one example—the divisor was $X - x$, for public x. Elsewhere, provers divide by

procedure $\mathsf{PC.Commit}'(\mathsf{pp}, [f]) \to c$:
  each party i:
    $c^{(i)} \leftarrow \sum_{j=0}^{d} f_j^{(i)} \cdot (\alpha^j \cdot g_1)$
    interpret $c^{(i)}$ as a share of $[c]$
  output $[c]$

(a) Commitments

procedure $\mathsf{PC.Prove}'(\mathsf{pp}, [f], x) \to \pi$:
  each party i:
    $q^{(i)}, r^{(i)} \leftarrow f^{(i)}/(X - x)$ (compute quotient and remainder)
    $\pi^{(i)} \leftarrow \sum_{j=0}^{d} q_j^{(i)} \cdot (\alpha^j \cdot g_1)$
    interpret $\pi^{(i)}$ as a share of $[\pi]$
  output $[\pi]$

(b) Evaluation proofs

Figure 3: KZG [66] procedures for shared polynomials

SNARK	FFT	MSM	Poly. Division	Vector Commit.	Sum-Check	Product-Check
Groth16 [59]	✓	✓	✓			
Marlin [35]	✓	✓	✓		✓	
Plonk [48]	✓	✓	✓			✓
Fractal [36]	✓		✓	✓		

Figure 4: Key components of SNARKs

far larger polynomials, but these polynomials are still public, admitting—as shown Section 5.3—a simple protocol.

Sum- and product-checks Two of the proof systems listed, Plonk and Marlin, combine the aforementioned components with two different but related proof primitives: Marlin relies on SUMCHECK while Plonk relies on PRODCHECK. To explain these primitives, let c_f be a commitment to a polynomial $f \in \mathbb{F}_p[X]$, let $\Omega := \{1, \omega, \omega^2, \ldots \omega^{n-1}\}$ be a subset of $\mathbb{F}_p$, and let $c \in \mathbb{F}_p$. Then SUMCHECK and PRODCHECK are sub-SNARKs for the relations $\mathfrak{R}_+$ and $\mathfrak{R}_\times$ respectively, where

- $\mathfrak{R}_+(c_f, c; f)$ holds when $\sum_{x \in \Omega} f(x) = c$ and c_f is a commitment to f;
- $\mathfrak{R}_\times(c_f, c; f)$ holds when $\prod_{x \in \Omega} f(x) = c$ and c_f is a commitment to f.

The SUMCHECK SNARK is described in [35] while the PRODCHECK SNARK is described in [48]. The SUMCHECK prover is fairly direct. However the PRODCHECK prover operates by first computing all partial products

$$t_i := \prod_{j=0}^{i} f(\omega^j) \quad \text{for } i = 0, \ldots, n-1. \tag{1}$$

An MPC implementation of Plonk and Marlin requires proving SUMCHECK and PRODCHECK for shared polynomials. At first glance, one might expect that that an MPC for the PRODCHECK prover would be very costly. Indeed, the distributed prover has to compute a sharing of the n partial products in (1), which naively takes $n-1$ rounds.

We show that an MPC for the PRODCHECK prover can actually be implemented in a constant number of rounds. To do so, we use a technique due to Bar-Ilan and Beaver [4] to compute all partial products in a constant number of rounds:

- *step 1:* The provers generate shared values $[r_0], [r_0^{-1}], [r_1], [r_1^{-1}], \ldots, [r_n], [r_n^{-1}]$ for random $r_0, r_1, \ldots, r_n$ in $\mathbb{F}_p$.
- *step 2:* The provers use a multiplication protocol to compute and open the quantities $[r_{i-1} t_i r_i^{-1}]$ for $i \in \{1, \ldots, n\}$. Let $t'_1, \ldots, t'_n \in \mathbb{F}_p$ be the resulting public quantities.
- *step 3:* Use n multiplications in parallel to compute $[t_i] = (\prod_{j=1}^{i} t'_j) \cdot [r_0^{-1} \cdot r_i]$ for $i \in \{1, \ldots, n\}$.

The end result is all the partial products needed to run the PRODCHECK prover. The remainder of computing the PRODCHECK proof can be adapted to an MPC with previously discussed techniques.

Vector commitments and hashing Fractal, like Marlin and Plonk, can be viewed as a P-IOP, but with a very different kind of polynomial commitment. Rather than using the pairing-based KZG scheme, the prover commits to a vector of polynomial evaluations (over an enormous domain) using a *vector commitment*, which can be opened to any index. Then, $\mathcal{P}$ and $\mathcal{V}$ use a *low-degree test* to establish that most elements in vector are consistent with a low-degree polynomial. Then, future polynomial evaluation queries reduce to vector commitment index openings. By using a hash-based vector commitment, the security of Fractal rests only on cryptographic hashing; which is plausibly quantum secure—unlike pairings.

Fractal's best low-degree test, DEEP-FRI [13, 21], is based on the fast Fourier transform, and adapts naturally to an MPC.

Implementing the vector commitment efficiently in an MPC seems difficult. Fractal's best vector commitment is a Merkle tree: a binary tree of hash evaluations that reduce a vector to a single hash that serves as the commitment to the vector. Hash functions, by necessity, are extremely non-linear, so evaluating them on shared data is extremely expensive. Even hash functions explicitly designed to minimize non-linearity [2, 58] still require hundreds of field multiplications.

```rust
struct AddShare<F> {
  f: F,

}

impl<F: Field> FieldShare<F>
 for AddShare<F> {
  fn shift(&mut self, c: F) {
    if net::am_first_party() {
      self.f += c;
    }
  }
  // ... other methods ...
}
```

```rust
struct AuthShare<F> {
  val: AddShare<F>,
  mac: AddShare<F>,
}

impl<F: Field> FieldShare<F>
 for AuthShare<F> {
  fn shift(&mut self, c: F) {
    self.val.shift(c);
    self.mac.f +=
      mac_key_share() * c;
  }
  // ... other methods ...
}
```

```rust
struct GszShare<F> {
  y: F,
  // x is party number
}

impl<F: Field> FieldShare<F>
 for GszShare<F> {
  fn shift(&mut self, c: F) {
    self.y += c;
  }


  // ... other methods ...
}
```

Figure 5: Fragments of the definitions of additive, authenticated, and GSZ shares

```rust
enum MpcField<F, S> {
  Public(F),
  Shared(S),
}

impl<F: Field, S: FieldShare<F>>
  Field for MpcField<F, S> {
  // ...
}
```

```rust
fn foo<F: Field>() { /* .. */ }
use ark_bls12_377::Bls12_377::Fr as OurField;

fn run_local() {
  foo::<OurField>();
}
fn run_mpc() {
  net::init(); // reads config file
  foo::<MpcField<OurField, GszShare<OurField>>>();
}
```

Figure 6: Single and multi-party computations in our framework. `foo` is defined generically over a field `F`. Instantiating `F` with BLS 12-377's scalar field yields a local computation. Instantiating `F` with `MpcField` and `GszShare` yields a multi-party computation backed by the GSZ protocol.

To avoid this cost, one could replace the vector commitment to evaluations of a polynomial f with N vector commitments to evaluations of shares of f: $f^{(1)}, \ldots, f^{(N)}$. Then, when $\mathcal{V}$ requests an evaluation, each $\mathcal{P}_i$ opens the corresponding evaluation of $f^{(i)}$; $\mathcal{V}$ checks all N opening proofs and computes the sum. This minimizes prover cost, but the proof size grows by a factor of N, as does $\mathcal{V}$'s work. When an application requires short proofs and fast verification, this might be unacceptable.

6 Implementation

We implement collaborative proofs based on Groth16, Marlin, and Plonk. Our starting point is `arkworks` [43]: a collection of Rust libraries for implementing cryptographic proofs, used in a few recent works [32, 33, 35]. It includes interfaces and implementations for finite fields, pairing-friendly curves, and polynomial commitments. It also includes SNARKs implemented generically over their primitives; e.g., Groth16 and Marlin implementations applicable to any pairing e.

Collaborative SNARKs through MPC lifting Our implementation pursues three goals. First, to support different secret-sharing schemes, proof systems, and MPC protocols. Second, to achieve proving time concretely competitive with state-of-the-art conventional SNARKs. Third, to focus on the multi-party aspects of the implementation and optimization.

Given these considerations, we adopt a design that lifts an existing SNARK prover implementation into a multi-prover protocol by instantiating the primitives that the prover uses with secure multi-party protocols. At a high level, we:

(1) define interfaces for field and curve sharing schemes
(2) provide implementations based on the SPDZ and GSZ protocols
(3) define wrapper types for field and curve elements which can be either public or shared,
(4) implement `arkworks` field and curve interfaces for the wrapper types,
(5) instantiate `arkworks`-based SNARK provers with the wrapper types (yielding multi-prover protocols!), and
(6) optimize the protocols.

Building multi-party primitives We first define interfaces for shares of both field elements and groups of prime order (like pairing-friendly elliptic curves). From here on, we discuss only field support; group support is similar. The field share interface includes functions for scaling by public values, shifting by public values, adding shares, multiplying shares, and opening shares. We implement three sharing schemes generically over their underlying field. They use an network library to communicate and access protocol metadata (e.g., their party number). Figure 5 shows the definitions, and how to shift each type of share by a public value. For GSZ shares,[3] all parties shift their value; for additive shares, only one party does; for authenticated shares, the base share is updated by a

[3] Identical to Shamir shares, save the verification that runs before any share is opened.

single party, but the MAC is updated by everyone using their share of the MAC key.

To represent values in an MPC, we introduce a union type `MpcField` for either public or shared field elements. `MpcField` is generic over both the field *and* the sharing scheme. We implement the `Field` interface for `MpcField`, allowing it to be used in any computation that expects to operate on field elements. Figure 6 sketches the code for this. By instantiating the function `foo` (perhaps not intended as an MPC) with an `MpcField` based on `GszShare`, we execute it as an honest-majority secure multi-party computation.

Lifting SNARKs into MPCs and optimizing Similarly, we can obtain collaborative proofs from prover implementations that are generic over the `arkworks` interfaces. For Marlin and Groth16 we can use implementations from `arkworks`; Plonk has no `arkworks`-based implementation yet, so we build one.

The resulting protocols (with a few small tweaks) are correct but perform poorly. The problem is that some operations are not optimized for an MPC. One example is computing the partial products of a list (discussed in Section 5.4). Without optimization, this entails a sequence of calls to the `Field` interface's multiplication method; each call requires a round of communication, which is expensive. To allow for an MPC-optimized protocol here (i.e., the one from Section 5.4), we add a partial product method to the `Field` interface. The default implementation is based on serial multiplication, but we also provide an overriding implementation for a sequence of shared values. We repeat this pattern (find a problematic operation, identify a better MPC protocol, add the operation to the relevant algebraic interface, and provide an MPC-optimized implementation) for batch multiplication, polynomial division, batch inversion, multi-scalar multiplication, and the fourier transform. These new methods and their default implementations are our primary changes to the `arkworks` algebraic interfaces. Calls to these methods are our primary changes to the SNARK prover implementations.

Limitations Our approach is effective for collaborative proofs, but cannot be applied to *all* computations based on `arkworks`. Critically, it does not support programs which branch on shared data or access data at shared locations. The limitation is fundamental: MPC protocols support *circuit evaluation*, not *RAM machine execution*—which the Rust programming language and `arkworks` target. While a program *might* perform circuit-incompatible operations, our approach works because the provers we study do not do that.[4] Techniques for translating RAM programs into circuits [6, 28, 29, 69, 70, 81, 93] could yield more general support.

[4]There are a few exceptions. Some basic operations (polynomial division, multi-scalar multiplication) branch on shared data—we provide MPC-optimized replacements anyway, mooting the issue. Some provers also check intermediate (shared) proof material for consistency—we defer all checks until the final proof is revealed.

Source code In sum, our implementation comprises a network library (≈700 lines), interfaces and implementations of sharing schemes (≈3000 lines), MPC implementations of `arkworks` algebraic interfaces (≈2000 lines), an `arkworks` Plonk implementation (≈1200 lines), and tests and benchmarking scripts (≈3000 lines). We also slightly modify a number of `arkworks` libraries. Our implementation is available at https://github.com/alex-ozdemir/multiprover-snark.

7 Evaluation

We evaluate our collaborative SNARKs against conventional SNARKs. Our metric is single-threaded prover runtime; for multiple parties, this is their average runtime. For all configurations, we report the median of three trials. We vary these parameters:

- collaborative proof: based on Groth16, Marlin, or Plonk
- number of rank-1 constraints: from 2 to 2^{20}. We discuss the number of constraints needed by one application at the end of the section.
- MPC algorithm: GSZ (honest majority) or SPDZ (dishonest majority)
- link capacity: from 1Mb/s to 3Gb/s

As we will see, collaborative proofs are concretely fast: over gigabit links, multi-prover protocols are nearly as fast as a single prover.

Preprocessing We do not evaluate pre-processing. Omitting preprocessing is acceptable for two reasons. First, preprocessing can be performed when spare compute and bandwidth are available. Second, preprocessing is concretely faster than proof generation. The most expensive preprocessing requires by either of our base protocols (SPDZ and GSZ) is the *multiplication triple generation* phase of SPDZ. Yet, with recent protocols [8, 64, 86], this requires only a few microseconds per triple (e.g., [8] generates one million triples in in ≈ 5s). Meanwhile, even the fastest prover (Groth16)—which uses one triple per constraint—requires over 200 µs per constraint.

High capacity links Our first experiment measures proving time for varying MPC algorithms, numbers of constraints, and proof systems over a high-capacity link. We consider 2 and 3-party proofs, omitting honest majority 2-party proofs, which provide no security. We consider constraints systems of size 2 through 2^{20}, and collaborative proofs based on Groth16, Marlin, and Plonk. For each proof system, we compare with the single-prover time. Google Cloud Platform (GCP) `n2-standard-2` machines are our testbed; these have 1 hardware core (Cascade lake), 8GB of memory, and 3Gb/s links. We run Debian 10 with hyperthreading disabled.

Figure 7 shows the results. Missing data points indicate memory exhaustion. For small constraint systems the multi-prover protocols are slower than a single prover, because network round-trips (which take milliseconds) dominate. How-

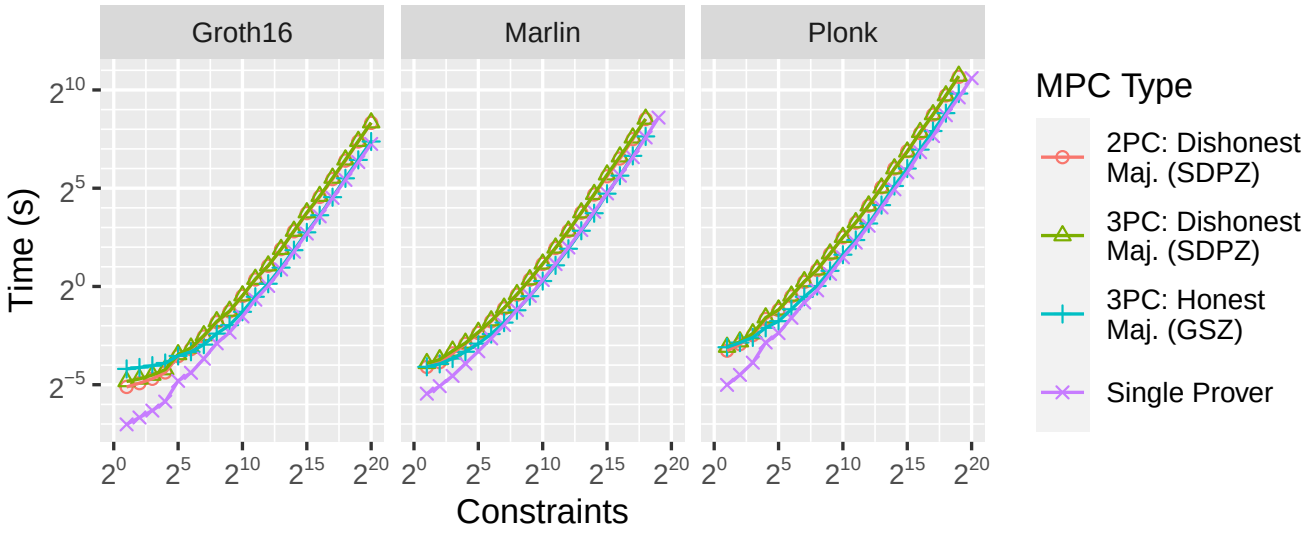


Figure 7: Proving time for varying numbers of parties and protocols, over a 3 gigabit link.

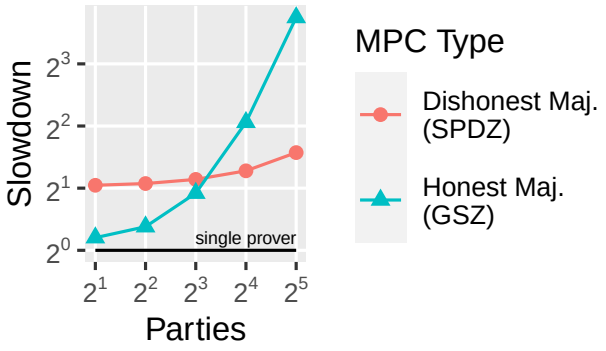


Figure 8: Proving time versus number of parties, normalized by single-prover time.

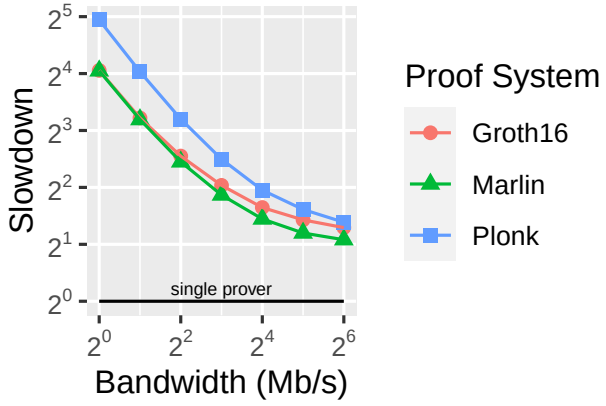


Figure 9: Proving time versus link capacity, normalized by single-prover time.

ever, the number of round-trips is sub-linear in constraints,[5] and for large constraint systems ($\gtrsim 2^{10}$) the protocols perform quite well. The honest majority protocol (3PC GSZ) runs in essentially the same time as a single prover; the dishonest majority protocols (SPDZ) run in essentially twice the time. This is because the computational costs of the protocols dominate the bandwidth costs. SPDZ is twice as slow as a single prover because it applies the computation to the data *and* its MAC: duplicating the computational work.

Many parties To show how the different MPC implementations scale with party count, we write Groth16 proofs for 2^{10} constraints, with varying numbers of parties. We use the same testbed as the previous experiment, and report *slowdown*: the time to write the collaborative proof, divided by the time of a single prover. Figure 8 shows the results. As the number of parties increases, the performance of SPDZ slowly degrades, while the performance of GSZ degrades faster. The crucial difference is in the amount of inter-prover communciation. For SPDZ, there is only a constant number of rounds of communication, since the Groth16 circuit has low multiplicative depth. However, the multiplication checking protocol of GSZ requires a number of rounds logarithmic in the number of constraints. This means that for large numbers of parties, one should use SPDZ instead of GSZ, even if one assumes a majority of provers are honest.

Low capacity links While gigabit links are common in LANs, within datacenters, and between data centers, lower capacity links are common elsewhere. Our second experiment measures the relationship between proving time and

[5] SPDZ is constant-round, while GSZ requires a logarithmic number of rounds to check multiplication triples.

Type	Size	Compute	Traffic	Total Cost
2PC	2^{10}	1.9×10^{-5}	1.0×10^{-5}	2.9×10^{-5}
1PC	2^{10}	4.6×10^{-6}	0	4.6×10^{-6}
2PC	2^{20}	0.019	0.011	0.030
1PC	2^{20}	0.0048	0	0.0048

Figure 10: Price estimates (in US$) for collaborative (2PC) and conventional (1PC) proofs. The "size" column refers to the number of constraints.

link capacity. We cohost two provers on an Arch Linux (kernel 5.12.15) machine with an AMD Ryzen-2700 CPU and 32GB of RAM. We use the Mahimahi network shell suite [78] to emulate low-capacity links. We report slowdown for SPDZ evaluating varying provers on 2^{10} constraints with varying network capacity.

Figure 9 shows the results. At low bandwidths, data transfer dominates proving time, so slowdown grows linearly with the reciprocal of link capacity. At higher bandwidths, computational costs dominate, so slowdown approaches 2 (recall that SPDZ duplicates the computation). Plonk's slowdown is higher than the other proof systems. This is due to the batch multiplications and divisions in its ProdCheck; while our optimization (Section 5.4) keeps round complexity constant, the communication volume is still substantial.

Price We also estimate the *price* of writing collaborative proofs. We consider 2-prover Groth16 proofs using the SPDZ protocol. Communication is dominated by 256-bit field elements: $4n$ are transferred. Figure 7 shows that the computation requires $\approx 488\,\mu s$ per constraint. Currently, GCP `e2-standard-2` VMs cost \$0.067/hr [84], and cross-continent traffic between VMs costs \$0.08/GB [83]. At these rates, Figure 10 summarizes net costs for proofs of varying size and type. Collaborative proofs (2PC) have quadruple the compute costs of conventional ones (1PC), because they involve twice the wall-clock time and two machines. Ultimately, net costs are tiny: million-constraint collaborative proofs cost only 3 cents.

Proofs about net assets In Section 2 we discussed a settings where a client of multiple banks wants a publicly verifiable proof that the net of its debits and credits across *all* its banks exceeds a threshold T. Here, we discuss the number of constraints required for such a proof.

To use our collaborative proofs, that claim must be written as a rank-1 constraint system (R1CS). How does the number of constraints depend on the number of transactions (credits and debits)? The constraints must (a) open the bank's commitments to all the customer's transactions, (b) check that each transaction is well-formed, (c) sum their values, and (d) compare that sum to T.

Let there be n transactions split among N banks. Each transaction is a 64-bit signed integer. Let bank i have n_i transactions in a Merkle tree of $n_i + 1$ leaves—the extra leaf is fully random to make the commitment hiding. Opening bank i's commitment uses n_i hash evaluations: one for each internal node in the binary Merkle tree. Thus, opening all the commitments uses $n = \sum_i n_i$ hash evaluations. The Poseidon [58] hash function requires ≈ 300 constraints per evaluation.

Furthermore, checking that each committed transaction has a value representable as a 64-bit signed integer requires ≈ 64 constraints. In R1CS, computing the sum requires essentially no constraints, and comparing with T requires a small constant number of constraints: $\approx 64 + \log_2 n$. In sum, a proof about n transactions requires $\approx 364n$ constraints. Thus, 2^{20} constraints suffices for proofs about ≈ 2900 customer transactions. This means that three banks could write a proof about ≈ 2900 transactions in $\approx 165\,s$. This time could be further reduced with parallelism (§9).

8 Related work

The most closely related work is [65] (building on [94]), which is concurrent and independent. Their focus is on enabling *MPC-as-a-service*, by building a PA-MPC protocol (as defined in [7]) from Marlin [35]. They specially integrate the PA-MPC commitments that each party posts to its input with the Marlin prover, via *polynomial evaluation commitments*: a primitive they introduce. In contrast, our focus is on collaborative proofs: we explore the MPC friendliness of different zk-SNARK systems. Any of our constructions yields PA-MPC (§4.1).

Another line of work [1, 25, 26], based on Prio [38], considers proofs over a witness shared among multiple *verifiers*. In our setting, the witness is shared among multiple *provers*.

Still other works distribute witness material among multiple provers for efficiency [100] or delegating computation [88]. We consider a witness that is already distributed; each prover wishes to conceal their part of the witness.

Other research [15, 27, 67] uses MPC to securely sample zk-SNARK parameters (e.g., to ensure that KZG's α parameter is discarded). This is orthogonal to our approach, which is securely distribute the proving algorithm of a zk-SNARK.

9 Discussion

A partial barrier to reducing communication Since communication is a bottleneck over low-capacity networks, one might try to reduce it. However, we show a partial barrier: $\Omega(n)$ communication is required if the R1CS witness is additively shared. We show this, in Appendix B, through a reduction from *disjoint*: a known communication-hard problem.

Parallelization For simplicity, we evaluate single-threaded provers. However, our techniques should yield multi-threaded multi-prover protocols with similarly good performance compared to a multi-threaded single prover. In most provers, par-

allelism accelerates Fourier transforms and multi-scalar multiplications. Our protocols perform these operations locally, so the same benefits should be achievable. Similarly, distributed computing [100] and hardware acceleration [103] could be used. However, parallelism would be less helpful over low-capacity links, where bandwidth becomes the bottleneck (§7).

Latency Our evaluation considered the effect of link capacity—but not latency—on proving-time. All protocols have round-complexity sub-linear in the constraint count, so latency has an insignificant effect as constraint count grows.

Leaking relation membership We emphasize that a collaborative proof for (x, w) reveals whether (x, w) is in $\mathfrak{R}$. This is unavoidable: completeness and soundness require it. For some relations, this enables malicious provers to glean information about another prover's witness. As a simple example, the relation that checks whether two witnesses agree in the first bit lets one prover learn the first bit of another prover's witness. When using a collaborative proof, one must consider the worst-case leakage from the relation itself.

Significance We've shown that collaborative proofs can be constructed with little to no computational overhead compared to single-prover proofs. Communication costs are asymptotically $\Theta(\lambda n)$, but concretely small over commodity high-capacity links. In sum, most server-side applications that can tolerate the (considerable) cost of a single-prover proof should also be able to tolerate the cost of a collaborative proof. We hope this makes succinct, public verifiability available to more applications—even when the secrets of multiple parties are involved.

Acknowledgements

We thank Benedikt Bünz, Mark Braverman, Veronica Rivera, Li-Yang Tan, and Riad Wahby for helpful comments and advice, and Weikeng Chen, Pratyush Mishra, Dev Ohja, and others for developing and maintaining `arkworks`.

This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, and a Google Cloud Platforms grant. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares, 2021. https://ia.cr/2021/576.

[2] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, 2016.

[3] T. Ashur and S. Dhooghe. MARVELlous: a STARK-friendly family of cryptographic primitives. Cryptology ePrint Archive, Report 2018/1098, 2018. https://eprint.iacr.org/2018/1098.

[4] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *8th ACM PODC*, 1989.

[5] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, 2005.

[7] C. Baum, I. Damgård, and C. Orlandi. Publicly auditable secure multi-party computation. In *SCN*, 2014.

[8] C. Baum, D. Escudero, A. Pedrouzo-Ulloa, P. Scholl, and J. R. Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-LWE. In *SCN*, 2020.

[9] D. Beaver. Multiparty protocols tolerating half faulty processors. In *CRYPTO*, 1990.

[10] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, 2008.

[11] A. Beimel. Secure schemes for secret sharing and key distribution, 1996.

[12] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *CRYPTO*, 1993.

[13] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *ICALP 2018*, 2018.

[14] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO*, 2019.

[15] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *IEEE S&P*, 2015.

[16] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, 2019.

[17] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *TCC*, 2016.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.

[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, 2014.

[20] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, 2012.

[21] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf. DEEP-FRI: Sampling outside the box improves soundness. In *ITCS*, 2020.

[22] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. The hunting of the SNARK. *Journal of Cryptology*, 30(4):989–1066, 2017.

[23] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. Verifiable computation using multiple provers, 2014. https://eprint.iacr.org/2014/846.

[24] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *FC*, 2009.

[25] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-

knowledge proofs on secret-shared data via fully linear PCPs. In *CRYPTO*, 2019.

[26] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[27] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In *FC Workshops*, 2019.

[28] B. Braun. Compiling computations to constraints for verified computation. *UT Austin Honors Thesis HR-12-10*, 2012.

[29] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles*, 2013.

[30] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT*, 2005.

[31] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In *EUROCRYPT*, 2020.

[32] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. Proof-carrying data without succinct arguments, 2020. https://ia.cr/2020/1618.

[33] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for inner pairing products and applications, 2019. https://eprint.iacr.org/2019/1177.

[34] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, 2018.

[35] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT*, 2020.

[36] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, 2020.

[37] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.

[38] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017.

[39] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multiparty computations secure against an adaptive adversary. In *EUROCRYPT*, 1999.

[40] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

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

[42] N. Daswani and M. Elbayadi. *Big Breaches: Cybersecurity Lessons for Everyone*. Apress, 2021.

[43] A. Developers. Arkworks, 2020. https://github.com/arkworks-rs.

[44] Z. Developers. What is JubJub? https://z.cash/technology/jubjub/. Retrieved 30 July 2021.

[45] D. Evans, V. Kolesnikov, and M. Rosulek. *A pragmatic introduction to secure multi-party computation*. 2017.

[46] M. Evans, A. W. Mathews, and M. Tom. Hospitals often charge uninsured people the highest prices, new data show. *The Wall Street Journal*, 2021.

[47] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.

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

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

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

[51] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[52] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *17th ACM PODC*, 1998.

[53] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM STOC*, 1987.

[54] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *ACM STOC*, 2008.

[55] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song. Atlas: Efficient and scalable mpc in the honest majority setting, 2021. https://ia.cr/2021/833.

[56] V. Goyal and Y. Song. Malicious security comes free in honest-majority mpc, 2020. https://eprint.iacr.org/2020/134.

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

[58] L. Grassi, D. Kales, D. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. Starkad and Poseidon: New hash functions for zero knowledge proof systems. Cryptology ePrint Archive, Report 2019/458, 2019. https://eprint.iacr.org/2019/458.

[59] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[60] J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *CRYPTO*, 2017.

[61] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[62] J. Håstad and A. Wigderson. The randomized communication complexity of set disjointness. *Theory of Computing*, 3(1):211–219, 2007.

[63] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, 2008.

[64] W. jie Lu and J. Sakuma. Faster multiplication triplet generation from homomorphic encryption for practical privacy-preserving machine learning under a narrow bandwidth. Cryptology ePrint Archive, Report 2018/139, 2018. https://eprint.iacr.org/2018/139.

[65] S. Kanjalkar, Y. Zhang, S. Gandlur, and A. Miller. Publicly auditable mpc-as-a-service with succinct verification and universal setup. https://arxiv.org/abs/2107.04248, 2021.

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

[67] M. Kohlweiss, M. Maller, J. Siim, and M. Volkhov. Snarky ceremonies, 2021. https://ia.cr/2021/219.

[68] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. C∅c∅: A framework for building composable zero-knowledge proofs, 2015. https://ia.cr/2015/1093.

[69] A. E. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *IEEE S&P*, 2018.

[70] K. R. M. Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[71] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, 2013.

[72] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://eprint.iacr.org/2016/046.

[73] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, 2017.

[74] Y. Lindell, E. Oxman, and B. Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In *CRYPTO*, 2011.

[75] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[76] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-

choose oblivious transfer. In *TCC*, 2011.

[77] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[78] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, 2015.

[79] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, 2009.

[80] P. S. Nordholt and M. Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *ACNS*, 2018.

[81] A. Ozdemir, F. Brown, and R. S. Wahby. Unifying compilers for snarks, smt, and more, 2020. https://eprint.iacr.org/2020/1586.

[82] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, 2013.

[83] G. C. Platform. All networking pricing. Webpage, retrieved 21 July 2021. https://cloud.google.com/vpc/network-pricing.

[84] G. C. Platform. Pricing, compute engine. Webpage, retrieved 28 July 2021. https://cloud.google.com/compute/all-pricing.

[85] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *ACM STOC*, 1989.

[86] D. Rathee, T. Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS 19*, 2019.

[87] A. Sangers, M. van Heesch, T. Attema, T. Veugen, M. Wiggerman, J. Veldsink, O. Bloemen, and D. Worm. Secure multiparty PageRank algorithm for collaborative fraud detection. In *FC*, 2019.

[88] B. Schoenmakers, M. Veeningen, and N. de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *ACNS*, 2016.

[89] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. 2013.

[90] A. Shamir. How to share a secret. *Communications of the ACM*, 1979.

[91] N. P. Smart and Y. T. Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, 2019.

[92] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, 2013.

[93] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[94] M. Veeningen. Pinocchio-based adaptive zk-SNARKs and secure/correct adaptive function evaluation. In *AFRICACRYPT*, 2017.

[95] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P*, 2013.

[96] R. S. Wahby, M. Howald, S. J. Garg, a. shelat, and M. Walfish. Verifiable ASICs. In *IEEE S&P*, 2016.

[97] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[98] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE S&P*, 2018.

[99] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, 2015.

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

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

[102] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE S&P*, 2018.

[103] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *ISCA 2021*, 2021.

A Honest majority MPC for elliptic curve circuits

In this section we adapt the multiplication protocols of [40] and [57] to products of scalars and group elements. We follow the exposition of [56], which includes the protocols from both works.

We adopt a few notational conventions. Bolded quantities are vectors: $\cdot$ denotes inner product and $\circ$ denotes Hadamard product,. For $\mathbf{x}$, a vector, $\mathbf{x}_L$ and $\mathbf{x}_R$ denote its left and right halves and x_i denotes an element. We assume all vector lengths are a power of two; generalizing to other lengths is straightforward. $[v]_d$ denotes that v is distributed among the parties as degree-d Shamir shares. If the degree is omitted, it is assumed to be $t = \lfloor N/2 \rfloor$. Lowercase letters denote scalars and uppercase letters denote group elements. f and its subcripts denote scalar polynomials, F and its subscripts denote polynomials with group coefficients and a scalar variable. Vectors of polynomials are bolded, and polynomials (and vectors thereof) can be shared coefficient-wise. $[\mathbf{f}(x)]$ denotes shares of each polynomial in $\mathbf{f}$ evaluated at public x.

Protocols G-DOUBLE-RAND, G-DOUBLE-RAND, and G-COIN are *exactly* like RAND, DOUBLE-RAND, and COIN, save that the operate on Shamir shares of group elements rather than field elements.

The rest of the protocols are adapted to scalar-group products as follows. DE-LINEARIZATION (Protocol 13, [56]) reduces checking a sequence of multiplication tuples (a Hadamard product); it is adapted to G-HADAMARD-CK, which reduces checking a sequence of scalar-point multiplication triples to checking a scalar-point inner product.

G-HADAMARD-CK($[\mathbf{a}] \in \mathbb{F}^n, [\mathbf{A}], [\mathbf{B}] \in \mathbb{G}^n$)

Shows: $\mathbf{a} \circ \mathbf{A} = \mathbf{B}$

$\alpha \leftarrow$ COIN()

$\boldsymbol{\alpha} \leftarrow (\alpha^i)_{i=1}^n$

G-IP-CK($\boldsymbol{\alpha} \circ [\mathbf{a}], [\mathbf{A}], \boldsymbol{\alpha} \cdot [\mathbf{B}]$)

G-IP-COMPUTE adapts EXTEND-MULT (Protocol 10, [56]) from computing a field inner product to computing a scalar-point inner product. EXTEND-MULT is itself a generalization of the multiplication and degree reduction protocols of [40].

G-IP-COMPUTE($[\mathbf{a}] \in \mathbb{F}^n, [\mathbf{A}] \in \mathbb{G}^n$)

Computes $\mathbf{a} \circ \mathbf{A} = B$

Everyone computes the inner product on their shares

Interpret the results as $[B]_{2t}$

$[X]_t, [X]_{2t} \leftarrow$ G-DOUBLE-RAND()

All parties send $[B-X]_{2t}$ *to Party 1*

Party 1 opens $B-X$ *and reshares it as* $[B-X]_t$

return $[B] \leftarrow [B-X]_t + [X]_t$

G-IP-FOLD adapts EXTEND-COMPRESS (Protocol 12, [56]). The original protocol compresses N different inner product checks into one. The new one compresses 2 point-scalar inner product checks into one.

G-IP-FOLD($[\mathbf{x}] \in \mathbb{F}^n, [\mathbf{X}] \in \mathbb{G}^n, [P] \in \mathbb{G}$, $[\mathbf{y}] \in \mathbb{F}^n, [\mathbf{Y}] \in \mathbb{G}^n, [Q] \in \mathbb{G}$)

Reduces $\mathbf{x} \cdot \mathbf{X} = P \wedge \mathbf{y} \cdot \mathbf{Y} = Q$ *to* $\mathbf{a} \cdot \mathbf{A} = R$

Interpolate shares of lines $[\mathbf{f}]$ *such that*

$f_i(1) = x_i$ *and* $f_i(2) = y_i$ *for* $i \in [n]$

Interpolate shares of lines $[\mathbf{F}]$ *such that*

$F_i(1) = X_i$ *and* $F_i(2) = Y_i$ *for* $i \in [n]$

$[\mathbf{z}] \leftarrow [\mathbf{f}(3)]$; $[\mathbf{Z}] \leftarrow [\mathbf{F}(3)]$

$[S] \leftarrow$ G-IP-COMPUTE($[\mathbf{z}], [\mathbf{Z}]$)

Interpolate shares of a parabola $[F_p]$ *such that*

$F_p(1) = P$, $F_p(2) = Q$, *and* $F_p(3) = S$

$\alpha \leftarrow$ COIN()

$[\mathbf{a}] \leftarrow [\mathbf{f}(\alpha)]$; $[\mathbf{A}] \leftarrow [\mathbf{F}(\alpha)]$; $[R] \leftarrow [F_p(\alpha)]$

return $([\mathbf{a}], [\mathbf{A}], [R])$

Finally, G-IP-CK adapts MULT-VERIFICATION (Protocol 17, [56]), from a recursive protocol for product checking to a recursive protocol for point-scalar inner product checking. G-PROD-CK is the adapted base case (Protocol 15, [56]).

G-IP-CK($[\mathbf{x}] \in \mathbb{F}^n, [\mathbf{X}] \in \mathbb{G}^n, [P] \in \mathbb{G}$)

Shows: $\mathbf{x} \cdot \mathbf{X} = P$

while $n > 1$*:*

$[L] \leftarrow$ G-IP-COMPUTE($[\mathbf{x}_L], [\mathbf{X}_L]$)

$[\mathbf{x}], [\mathbf{X}], P \leftarrow$ G-IP-FOLD($[\mathbf{x}_L], [\mathbf{X}_L], [L]$, $[\mathbf{x}_R], [\mathbf{X}_R], [P-L]$)

$n \leftarrow n/2$

G-PROD-CK($[x_1], [X_1], [P]$)

G-PROD-CK($[x] \in \mathbb{F}, [X] \in \mathbb{G}, [P] \in \mathbb{G}$)

Shows: $x \cdot X = P$

$[r] \leftarrow$ RAND()

$[R] \leftarrow$ RAND()

$[S] \leftarrow$ G-IP-COMPUTE($[r], [R]$)

$[y], [Y], [Q] \leftarrow$ G-IP-FOLD($[(x,r)], [(X,R)], [(P,S)]$)

All parties open y, Y, *and* Q *and check* $y \cdot Y = Q$

B Communication Lower Bound

In this section, we show a $\Omega(n)$ lower bound on the communication needed to compute a collaborative proof when the witness is additively shared among two parties.

Our bound applies to proof systems that are *honestly sound*: a prover following the proving protocol with an invalid witness produces a valid proof with negligible probability.

Definition 1. *A proof* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *is* ***honestly sound*** *if for all* $(x,w) \notin \mathfrak{R}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, x, \pi) = 1 : \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(\mathfrak{R}, 1^\lambda) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, x, w) \end{array}\right] = \mathsf{negl}(\lambda)$$

Surprisingly, knowledge soundness does not imply honest soundness. While the former guarantees that a verifier can *extract* a valid witness from an acceptable proof, it makes no guarantees about the witness that the prover used.

As an example, consider the (trivial) relation

$$\mathfrak{R}_= = \{(x \in \mathbb{F}, w \in \mathbb{F}) : x = w\}$$

For this relation, define the proof system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ where Setup is a no-op, Prove always returns a null proof $\pi = \perp$, and Verify always return 1. This proof system is knowledge-sound, but not honestly sound.

Regardless, it's easy to show that the SNARKs we study are honestly sound.

It suffices to show a $\Omega(n)$ communication bound for checking a shared R1CS witness. If there were a low-communication protocol for generating a proof, parties could just check a shared witness by constructing and checking the proof. Per completeness and honest soundness, the proof is valid if and only if the witness was valid, except with negligible probability.

Unfortunately, the communication-hard problem DISJ^n reduces to checking additively shared R1CS witnesses. DISJ^n asks whether the two length-n bit-strings do not share a one at any index; i.e., $\mathrm{DISJ}^n(a,b) = \bigwedge_{i=1}^n \neg(a_i \wedge b_i)$. DISJ^n has been shown to have $\Omega(n)$ randomized 2-party communication complexity [5, 62]. $\mathrm{DISJ}^n(a,b)$ reduces to R1CS checking as follows. $\mathcal{P}_1$, who has a, interprets it as a zero-one vector in $\mathbb{F}^n$ and sets $x_0 \leftarrow a; y_0 \leftarrow \vec{0}; z_0 \leftarrow \vec{0}$. Simultaneously, $\mathcal{P}_2$, interprets b as a zero-one vector in $\mathbb{F}^n$ and sets $x_1 \leftarrow \vec{0}; y_1 \leftarrow b; z_1 \leftarrow \vec{0}$. With this reduction, a and b are disjoint if and only if (x_0, x_1), (y_0, y_1), and (z_0, z_1) are sharings of x, y, z such that $x \circ y = z$, an R1CS relation. □

Note that this is only a limited bound. First, it requires $\Omega(n)$—not $\Omega(n \log |\mathbb{F}|)$—communication. We suspect the stronger bound holds as well, but do not know of an existing communication complexity result that it immediately follows from. Second, the bound applies only to additively shared witnesses.

C KZG for Shared Polynomials

In this section we prove that our alternate KZG commitment and proof protocols ($\mathsf{PC.Commit}'$ and $\mathsf{PC.Prove}'$) can be used within an arithmetic MPC without compromising security.

We show this for all arithmetic MPCs which build on a *homomorphic secret sharing scheme*. This includes both SPDZ and GSZ.

Definition 2. *Homomorphic secret-sharing:*

Let SSS *be a secret sharing scheme for messages from* $\mathcal{X}$*, an* $\mathbb{F}$*-module, and with shares in* $\mathcal{S}$*, also an* $\mathbb{F}$*-module.*

SSS *is homomorphic if for all messages* $x, x' \in \mathcal{X}$ *shared as* $(s_1, \ldots, s_n) \in \mathcal{S}^n$ *and* $(s'_1, \ldots, s'_n) \in \mathcal{S}^n$ *respectively, and all scalars* $\alpha, \alpha' \in \mathbb{F}$*, the tuple* $(\alpha s_1 + \alpha' s'_1, \ldots \alpha s_n + \alpha' s'_n) \in \mathcal{S}^n$ *is is a sharing of* $\alpha x + \alpha' x'$*.*

The homomorphism of such a scheme is a natural communication-free protocol for adding shared values or scaling them by public constants.

Lemma 4. *Let* SSS *be a homomorphic secret sharing scheme. Let* C *be an arithmetic circuit with wire values in* $\mathcal{X}$*. Let* Π *be a secure MPC protocol that evaluates* C *by computing secret shares of its wire values in topological order—using* SSS*'s homomorphism to add or scale shares—and revealing the outputs. Let* C' *be a sub-circuit that computes a output wires* $y_1, \ldots, y_m$ *from input wires* $x_1, \ldots, x_n$*, according to a linear function* $f : \mathcal{X}^n \to \mathcal{X}^m$*. Let* Π' *be a protocol which is exactly like* Π*, save how the parties evaluate* C'*. Each party i, for inputs* x_j *to* C'*, lets* $x_j^{(i)}$ *denote its share of* x_j*'s value. Then, each party computes* $y_1^{(i)}, \ldots, y_m^{(i)} \leftarrow f(x_1^{(i)}, \ldots, x_n^{(i)})$*, and takes each* $y_j^{(i)}$ *to be its share of the value of* y_j *in evaluating the rest of the circuit. Then* Π' *is a secure MPC protocol for evaluating* C*.*

Proof. First, we show that Π and Π' differ only in the local computation performed by each party: not in the messages they send. Then, we show Π' is secure.

Since $y = f(x)$ is a linear transformation, there exists a matrix $M \in \mathbb{F}^{m \times n}$, such that $y = Mx$. Without loss of generality, let the sub-circuit C' compute Mx naively, that is, let it compute y_j as $\sum_{i=1}^{n} M_{ij} x_i$ (n scalar multiplication gates and $n-1$ addition gates). Since Π evaluates addition and scalar multiplication gates in C according to the homomorphism of SSS, evaluating C' involves no communication. Each party i locally computes a share shares $y_j^{(i)}$ of wire y_j from its shares $x_i^{(j)}$ of wires x_i using the formula

$$y_j^{(i)} = \sum_{k=1}^{n} M_{kj} x_k^{(i)}$$

But this is equivalent to computing $(y_1^{(i)}, \ldots, y_m^{(i)}) = f(x_1^{(i)}, \ldots x_n^{(m)})$, as done in protocol Π'. Thus, Π and Π' differ only in the local computation performed by each party when evaluating C'—parties following Π and Π' send identical messages.

Since Π is secure, for any adversary $\mathcal{A}$, there is an efficient algorithm Sim, which can simulate its view when interacting with honest parties following Π. Since honest parties following Π and Π' send the same messages, Sim is also a simulator for $\mathcal{A}$ when it interacts with parties following Π'. Thus, the view of any adversary participating in Π' can be simulated, so it is secure. □

Both of the MPC protocols we build on (GSZ and SPDZ) meet the conditions of Lemma 4. SPDZ uses authenticated secret sharing, and GSZ uses Shamir secret sharing. Both schemes are homomorphic.

Lemma 4 concerns functions $f : \mathcal{X}^n \to \mathcal{X}^m$ with domain and codomain elements from the same space. However, it's easy to generalize to lists of inputs and outputs from different spaces—that is, to $f : (I_i)_{i=1}^n \to (O_i)_{i=1}^m$ where each I_i or O_i may be a different space—so long as each space is an $\mathbb{F}$-module with a homomorphic sharing scheme. Below, we use a generalization to functions which operate on a mix of field and curve elements.

Corollary 1. *Let* Π *be a secure protocol which computes a KZG commitment by evaluating an arithmetic circuit. Let* Π' *be the same protocol, save that the commitment is computed using* $\mathsf{PC.Commit}'$*. Then* Π' *is secure.*

Proof. First, consider the function $C' : \mathbb{F}^{d+1} \to \mathbb{G}_1$ that map polynomials of degree up to d to their commitments. C' maps any vector of coefficients $f_0, \ldots, f_d$, to $\sum_i f_i \cdot h_i$ where each $h_i = \alpha_i \cdot g_i$ and is part of the public parameters. Observe that C' is linear, for polynomials $f, g \in \mathbb{F}^{\leq d}[X]$, and scalars $x, y \in \mathbb{F}$, $C'(xf + yg) = \sum_i (xf_i + yg_i) \cdot h_i = \sum_i (xf_i + yg_i) \cdot h_i = x \cdot \sum_i (f_i \cdot h_i) + y \cdot \sum_i (g_i \cdot h_i) = x \cdot C'(f) + y \cdot C'(g)$.

Thus, Lemma 4 implies that Π' is secure. □

Corollary 2. *Let* Π *be a secure protocol which computes a KZG opening proof by evaluating an arithmetic circuit. Let* Π' *be the same protocol, save that the commitment is computed using* $\mathsf{PC.Prove}'$*. Then* Π' *is secure.*

Proof. A KZG proof that $f(z) = y$ is computed in two steps. First, compute $q(X) = \frac{f(X) - y}{X - z}$. Second, commit to q. $\mathsf{PC.Prove}'$ uses $\mathsf{PC.Commit}'$ for the second step, which Corollary 1 shows does not compromise security. We consider the first step. It computes shares of the quotient from shares of f, i.e. it computes a function $C' : \mathbb{F}^{d+1} \to \mathbb{F}^d$. We show that this function is linear. Let f and g be polynomials, a and b be scalars, and let $C'(f) = q_f$ and $C'(g) = q_g$. Then there exist remainder constants r_f and r_g such that $f = q_f \cdot (X - z) + r_f$ and $g = q_g \cdot (X - z) + r_f$. Scaling the first equation by a, scaling the second by b, and summing, gives

$$af + bg = (aq_f + bq_g) \cdot (X - z) + (ar_f + br_g)$$

Definition 2. *A secure PA-MPC for t malicious parties: A PA-MPC for function $f : \mathcal{X}^N \to \mathcal{Y}$ and a commitment scheme* Commit *is a tuple* $(\mathsf{Setup}, \Pi, \mathsf{Verify})$ *such that:*

- $\mathsf{Setup}(f) \to \mathsf{pp}$*: setup public parameters for f*
- $\Pi(\mathsf{pp}, \vec{x}) \to (y, \pi)$*: evaluate f on $\vec{x}$ and create a proof.*
- $\mathsf{Verify}(\mathsf{pp}, \vec{c}, y, \pi) \to \{0,1\}$*: check output with respect to input commitments.*

A PA-MPC is secure against t malicious parties if

- Π_y *(the restriction of Π to its y output) is a secure-with-abort MPC for function f, against t malicious parties, and*
- $(\mathsf{Setup}, \Pi_\pi, \mathsf{Verify})$*, where Π_π is the restriction of Π to its π output, is a collaborative zk-SNARK for the relation*

$$\mathfrak{R}_{f,\mathsf{Commit}} := \Big\{ (\vec{c}, y\ ;\ \vec{x}, \vec{r}) : f(\vec{x}) = y \wedge \bigwedge_{i \in [N]} c_i = \mathsf{Commit}(x_i, r_i) \Big\}$$

that is secure against t malicious provers.

Since $a_r f + b r_g$ is a constant, and polynomial division is unique, $a q_f + b q_g$ is the quotient when $af + bg$ is divided by $X - z$. Thus $C'(af + bg) = a q_f + b q_g$, so C' is linear.

Thus, Lemma 4 implies that Π' is secure. □

D PA-MPC from collaborative proofs

Publicly-auditable MPC (PA-MPC) was originally defined in terms of simulation [7], with the auditor as an additional party. Definition 2 gives an alternative formulation based on collaborative proofs. With this definition, it's straightforward to construct PA-MPC from a collaborative proof. We give the construction below.

For a multivariate function $f(\vec{x}) \to y$ and a commitment scheme $\mathsf{Commit}(x, r) \to c$, let $\mathfrak{R}_{f,\mathsf{Commit}}$ be as in Definition 2.

Theorem 5. *Let $f : \mathcal{X}^N \to \mathcal{Y}$ be a function, and let* Commit *be a binding and hiding commitment scheme. Let Π_f be an MPC for f that is secure-with-aborts against t malicious parties, and let $(\mathsf{Setup}_\pi, \Pi_\pi, \mathsf{Verify}_\pi)$ be a collaborative proof for $\mathfrak{R}_{f,\mathsf{Commit}}$ that is secure against t malicious provers. Then, there is a PA-MPC protocol for f and* Commit*, for t malicious parties, with the same proof size and verification time as the collaborative proof.*

Proof. We construct the PA-MPC protocol, $(\mathsf{Setup}, \Pi, \mathsf{Verify})$:

- $\mathsf{Setup}(1^\lambda)$: output $\mathsf{Setup}_\pi(1^\lambda)$.
- $\Pi(\mathsf{pp}, \vec{x}, \vec{r}, \vec{c})$: compute $y \leftarrow \Pi_f(\vec{x})$ and $\pi \leftarrow \Pi_\pi(\mathsf{pp}, (\vec{c}, y), (x_i, r_i)_{i=1}^N)$; output (y, π).
- $\mathsf{Verify}(\mathsf{pp}, \vec{c}, y, \pi)$: output $\mathsf{Verify}_\pi(\mathsf{pp}, (\vec{c}, y), \pi)$

The security of this PA-MPC protocol follows immediately from its construction and the definition of PA-MPC security. □

Note that in this approach, Commit must be expressible within a relation that the collaborative proof system supports. This paper constructs collaborative proofs for R1CS. Efficient rank-1 constraint systems have been designed for commitments based on discrete-log [44, 68], SHA-2 [2, 68], algebraic hash functions [2, 3, 58], and more.