Fast Zero-Knowledge Argument System with Short Polynomial Using Direct Computation

Frank Y.C. Lu

FSHKT, YinYao

Abstract. We introduce a new transparent zero-knowledge argument system that will significantly reduce the size of the polynomial commitment scheme (PCS) that needs to be evaluated relative to the circuit size.
In our protocol, committed input parameters are transformed into a format that the verifier can process directly, so the output of the circuit polynomial at some evaluation point can be directly computed by the verifier instead of asking the prover to run the expensive PCS protocol on circuit polynomial(s) as big as the circuit size. In the default setting, the prover runtime cost for group exponentiation operations is only the square root of the degree ($\sqrt{p_d}$) of the polynomial the circuit generates.
This direct computation approach brings many additional benefits. For example, it allows inputs and outputs of a circuit to be in the same commitment format. This will open up new use cases such as creating an openly accessible database with data perfectly hidden using commitments, but still allowing anyone to validate updates to the data with zero-knowledge.
Our benchmark result shows our approach can significantly improve both proving and verification speed over the state-of-the-art by one order of magnitude for all types of circuits of any depth, while keeping the communication cost competitive. (code is included in the attachment file).
Our approach also allows our protocol to be made memory-efficient while being non-interactive. The theoretical memory cost of our protocol is $O(b + s)$, where $s = b = \sqrt{p_d}$ in the default setting.

1 Introduction

Ever since the discoveries of interactive proofs (IPs) [27] and probabilistically checkable proofs (PCPs) [6] [5] [4] [3] in the late last century, there has been a tremendous amount of research in the area of proof systems. More recently, the rise of blockchain and Web3 has finally triggered real-world interest in zero-knowledge systems.

Due to the expensive computation cost in the setup phase of earlier SNARKs (Succinct Non-Interactive Argument of Knowledge), the industry developed protocols that have the structured reference string (SRS) be constructible in a "universal and updatable" fashion. The first such universal SNARK was in Groth et al. [29], and Maller et al. improved the SRS size from quadratic to linear in Sonic [32]. More recently developed protocols such as PLONK [25], MARLIN [18] are universal fully-succinct SNARK with significantly improved prover runtime compared to the fully-succinct Sonic. However, many of these universal succinct SNARKs systems require trusted setup, and the prover run-time of these protocols is prohibitively expensive even with the latest improvements such as HyperPlonk [17], usually takes over 100 seconds on a single-threaded CPU for a circuit with over 2^{20} constraints.

Other classes of protocols including the Goldwasser, Kalai, and Rothblum (GKR) class such as Hyrax [37], Virgo [42]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [26] and Ligero/Ligero++ [1] [10] offer efficient prover runtimes that are at least one order of magnitude more efficient than pairing-based SNARKs. However, these protocols are largely ignored by the industry (e.g., the blockchain community) due to their expensive verifier runtime and high communication cost (hundreds of KBs) compared to fully succinct SNARK and STARK [9] protocols.

NIZKs such as SpartanNIZK [35] and later Lakonia [36] seem to offer a much more balanced approach, where they offer efficient prover runtime (6-18 seconds single thread) and competitive communication costs for large circuits while not being layer dependent. However, the downside of these protocols is that their verifier performance is still expensive, usually in the 400+ ms range on a single-threaded CPU.

Recent advances in multivariate polynomial commitment schemes (PCS) such as Binius [21] can, in theory, improve the prover runtime performance for protocols using multivariate PCS even better with the right hardware. However, it does not bridge the gap in communication cost with popular ECC-based PCS and the efficiency gains fades for general arithmetic circuit.

Another category of protocols emphasizes on memory-efficiency such as garbled circuits [31] [24] [30] and Vector Oblivious Linear Evaluation (VOLE) protocols [13] [34] [15] [14] [41] [38] [8] [40] generally offer better prover performance. However, their verifier runtimes are just as expensive and generally require a designated verifier with very expensive communication cost. We can, however, use VOLE-in-the-head approach to construct NIZK [7], but that again requires linear verifier.

Our aim is to create a new transparent zero-knowledge protocol that is easy to code for both developers and business people and, at the same time, offers the best overall performance while free of significant performance shortcoming in any area. Specifically, we want to keep the communication cost comparable to those of the state-of-the-art and greatly improve both the prover runtime and the verifier runtime costs. Finally, we also want our new system to be memory-efficient without requiring a designated verifier like that of VOLE protocols.

Specifically, when processing a high-depth circuit of 2^{20} sequential multiplication gates ($p_d = n$) with 10 inputs on a single CPU thread, the performance of our protocol is as follows: 0.22 seconds for the prover time; 4.5 milliseconds for the verifiertime; and 16 kilobytes for the communication cost.

We have also used an unoptimized setup to estimate the cost of proving SHA256 hash function, which we also demonstrated significant improvement in the prover and the verifier time (detailed in the supplemental material **??**). The conservative estimate is that we can prove a SHA256 in 3 milliseconds and verify it in 0.72 milliseconds with

To the best of our knowledge, our protocol offers the best prover/verifier runtime speed in the literature by a large margin for circuits of any depth.

2 Summary of Contributions

Our approach is to design a new class of protocols that allows verifiers to compute the evaluation of polynomial commitments representing the circuit directly. In our protocol, circuit inputs in the commitment form are converted to linear polynomials in $\mathbb{Z}_q$ so that verifiers can use standard arithmetic operations in $\mathbb{Z}_q$ to just "execute" the evaluating circuit to get its output. Our protocol does not require trusted setup and depends only on discrete logarithm assumption.

There were past attempts that somewhat enabled verifiers to "validate" each multiplication gate on its own, such as Cramer and Damgård [20] and more recent designated-verifier (which is a limitation itself) VOLE (LPZK in particular) [23] [40] [22] [39] based protocols. In these older strategies, each multiplication gate computation is actually not computed but "confirmed" by the verifier using transcripts tied to each multiplication gate. As a result, the communication/verifier costs of these earlier protocols are still linear in the number of multiplication gates in a circuit.

On the other hand, the input transformation technique introduced by our protocol allows verifiers to use transformed inputs to directly (one operation for each operation, like we do with clear text data) compute the circuit output that is still bound to the challenge x. This is a first and brings us two direct benefits: 1) After "computing" the whole circuit using transformed inputs, the verifier does not need to validate polynomial(s) as big as the circuit size. 2) we can trivially break a large circuit into several smaller sub-circuits, minimizing the memory footprint to that of one sub-circuit.

In our protocol, committed input parameters in $\mathbb{G}$ are transformed to linear polynomials in $\mathbb{Z}_q$. For example, we have a circuit ${a_1}^d + {a_2}^d + {a_3}^d = r$ that takes inputs a_1, a_2, a_3 and outputs r. In our

protocol, the prover provides the transformed inputs a_1, a_2, a_3 in linear polynomial form $a'_1, a'_2, a'_3 \in \mathbb{Z}_q$ s.t. $a'_i = a_i + x\alpha_i \in \mathbb{Z}_q$ (α_i is its blinding key and x is the challenge generated during runtime). Since the transformed inputs are in field values, the verifier can plug these values directly into a program (circuit) and just "execute" them to get output $o = a'^d_1 + a'^d_2 + a'^d_3 = o$, which is also the evaluation at point x of a degree d polynomial s.t. $f(x) = o$. The constant term of this polynomial is the circuit output r and all other coefficients terms are equivalent to blinding keys. If the prover can prove 1) it knows all coefficients of the output polynomial before the evaluation point is given (e.g., using a polynomial commitment) and 2) all transformed inputs are legitimate, then we know the proof is valid.

The output polynomial in the example above has degree d because all transformed inputs (linear polynomials) have degree 1. Taking them to their d-th power and adding them together will also get a polynomial of degree d. So if the circuit is something like $a'^3_1 + a'^3_2 + a'^3_3 + , ..., + a'^3_n = o$, the degree of the output polynomial is still 4 regardless of how many addition and multiplication operations are in that circuit. Throughout our paper, we use the symbol p_d (short for "polynomial degree") to denote the degree of the output polynomial of a circuit. Please note that p_d is different from the common understanding of the "multiplication depth" or the "total number of multiplications" of a circuit. For example, the p_d value of circuit $a'^3_1 \cdot a'^5_2 + a'^6_3$ is 9, which is bigger than the multiplication depth (6) but smaller than the total number of multiplications (12). This implies addition/subtraction operations are not only cheap computation-wise but also have the added benefit of shrinking the size of the output polynomial that we need to evaluate.

Because the output of the polynomial of a circuit is directly computed by the verifier using transformed inputs, the verifier can just execute the circuit to get the correct polynomial output $f(x) = o$ for some evaluation point x, allowing us to bypass the need to use an expensive polynomial commitment scheme to evaluate the output polynomial as a whole.

Achieving Higher Performance By Evaluating Significantly Shorter Polynomial Unlike other zero-knowledge protocols that depend on polynomial commitment, the evaluation result at random point x is directly computed by the verifier itself in our protocol (through direct computation of transformed inputs in $\mathbb{Z}_q$), so it has to be correct. This allows us to bypass the expensive PCS protocols and only evaluate a small polynomial of $O(b)$ degrees, where b is the bootstrapping parameter set by the user that tells the protocol where to "reset" a degree d polynomial back to a linear polynomial (degree 1), a technique used to slow the growth of polynomial degree. In the default setting, we set $b = \sqrt{p_d}$.

In the memory-efficient setting, the theoretical memory cost of our protocol is $O(b + s)$. This makes our protocol extremely attractive because designated verifier VOLE-based memory-efficient protocols generally require one round of interaction and are extremely expensive in terms of verifier runtime cost and communication cost. Techniques to make VOLE protocols publicly verifiable such as VOLE-in-the-head approach will add even more overheads.

Same Format for Circuit Inputs and Outputs Having both inputs and output(s) in the same format (e.g. commitment scheme) allows the output(s) of one circuit to be directly reused as input(s) to another circuit. This is a very useful feature in practice when verifying data in a publicly accessible/verifiable data store (including but not limited to blockchain). e.g., allows many participants to continuously manage/update a shared datastore as long as they can prove these updates were correctly computed from existing data. While other zero-knowledge protocols may also be able to support such a feature in theory by mapping witnesses to some publicly accessible committed or encrypted data, it does not come naturally and will require additional cost that is not accounted for.

Expanded Use Cases We believe the biggest benefit of our input transformation based approach is that developers can just wire business logics (computer programs in their readable form) in a

much more straightforward fashion than the expensive arithmetization process required in most other protocols. We can do this because our system not only supports arithmetic (+, - ×, ÷) operators but also comparison operators ($<, >, \geq, \leq, =$) by natively embedding range proof (Section 6.3).

We introduce our protocol in an interactive setting where all verifier challenges are random field elements. In practice, we assume the Fiat-Shamir heuristic is applied to our protocol to obtain a non-interactive zero-knowledge argument in the random oracle model. While we introduce our protocol in the discrete log setting, it can theoretically be adjusted to the post-quantum setting by switching to plausible post-quantum PCS commitment schemes.

3 Preliminaries

3.1 Assumption

Definition 1. (Discrete Logarithmic Relation) For all PPT adversaries $\mathcal{A}$ and for all $n \geq 2$ there exists a negligible function $\boldsymbol{negl}(\lambda)$ s.t.

$$Pr\left[\begin{array}{c|c} \mathbb{G} = Setup(1^\lambda), g_0, .., g_{n-1} \xleftarrow{\$} \mathbb{G} & \exists\, a_i \neq 0 \\ a_0, .., a_{n-1} \in \mathbb{Z}_p & \wedge \prod_{i=0}^{n-1} g_i^{a_i} = 1 \\ \leftarrow \mathcal{A}(\mathbb{G}, g_0, ..., g_{n-1}) & \end{array}\right] \leq \boldsymbol{negl}(\lambda)$$

The Discrete Logarithmic Relation assumption states that an adversary can't find a non-trivial relation between the randomly chosen group elements $g_0, ..., g_{n-1} \in \mathbb{G}^n$, and that $\prod_{i=0}^{n-1} g_i^{a_i} = 1$ is a non-trivial discrete log relation among $g_0, ..., g_{n-1}$. Please note the generators we use in this paper are $g, h, u \in \mathbb{G}$.

3.2 Zero-Knowledge Argument of Knowledge

Interactive arguments are interactive proofs in which security holds only against computationally bounded provers. In an interactive argument of knowledge for a relation $\mathcal{R}$, a prover convinces a verifier that it knows a witness w for a statement x s.t. $(x, w) \in \mathcal{R}$ without revealing the witness itself to the verifier.

Let $(\mathcal{P}, \mathcal{V})$ denote a pair of PPT interactive algorithms, and $\boldsymbol{Setup}$ denotes a non-interactive setup algorithm that outputs public parameters pp given a security parameter λ. Let $\langle \mathcal{P}(pp, x, w), \mathcal{V}(pp, x) \rangle$ denote the output of $\mathcal{V}$ on input x after its interaction with $\mathcal{P}$, who has knowledge of witness w. The triple $(\boldsymbol{Setup}, \mathcal{P}, \mathcal{V})$ is called an argument for relation $\mathcal{R}$ if for all non-uniform PPT adversaries $\mathcal{A}$ it satisfies completeness, soundness, and zero-knowledge definitions defined below:

Definition 2. (Perfect Completeness) The triple $(\boldsymbol{Setup}, \mathcal{P}, \mathcal{V})$ satisfies perfect completeness if for all PPT $\mathcal{A}$:

$$Pr\left[\begin{array}{c|c} (pp, x, w) \notin \mathcal{R} \text{ or} & pp \leftarrow \boldsymbol{Setup}(1^\lambda) \\ \langle \mathcal{P}(pp, x, w), \mathcal{V}(pp, x) \rangle = 1 & (x, w) \leftarrow \mathcal{A}(pp) \end{array}\right] = 1$$

The soundness notion we consider in this work is computational witness-extended emulation.

Definition 3. (Computational Witness-Extended Emulation or CWEE) Given a public-coin interactive argument tuple $(\mathbf{Setup}, \mathcal{P}, \mathcal{V})$ and arbitrary prover algorithm $\mathcal{P}^*$, let $\boldsymbol{Recorder}\ (\mathcal{P}^*, pp, x, s)$ denote the message transcript between $\mathcal{P}^*$ and $\mathcal{V}$ on shared input x, initial prover state s, and pp generated by $\boldsymbol{Setup}$. Furthermore, let $\mathcal{E}\ \boldsymbol{Recorder}\ (\mathcal{P}^*, pp, x, s)$ denote a machine $\mathcal{E}$ with a transcript oracle for this interaction that can rewind to any round and run again with fresh verifier randomness. The tuple $(\boldsymbol{Setup}, \mathcal{P}, \mathcal{V})$ has CWEE if for every deterministic polynomial time $\mathcal{P}^*$ there exists

an expected polynomial time emulator $\mathcal{E}$ s.t. for all non-uniform polynomial time adversaries $\mathcal{A}$ the following holds:

$$\left| Pr\left[\mathcal{A}(tr) = 1 \;\middle|\; \begin{array}{c} pp \leftarrow Setup(1^\lambda) \\ (x,s) \leftarrow \mathcal{A}(pp) \\ tr \leftarrow \boldsymbol{Recorder}(\mathcal{P}^*, pp, x, s) \end{array} \right] - \right.$$

$$\left. Pr\left[\begin{array}{c} \mathcal{A}(tr) = 1 \wedge \\ tr \;\text{ accepting} \\ \implies (x,w) \in \mathcal{R} \end{array} \middle| \begin{array}{c} pp \leftarrow Setup(1^\lambda) \\ (x,s) \leftarrow \mathcal{A}(pp) \\ (tr,w) \leftarrow \mathcal{E}^{\boldsymbol{Recorder}(\mathcal{P}^*, pp, x, s)}(pp,x) \end{array} \right] \right| \leq \boldsymbol{negl}(\lambda)$$

Informally, if an adversary can produce an argument that satisfies the verifier with some probability, then there exists an emulator producing an identically distributed argument with the same probability, as well as a witness. The zero-knowledge property requires that the verifier doesn't learn anything about the witness from its interaction with an honest prover.

Definition 4. (Perfect Special Honest Verifier Zero Knowledge for Interactive Arguments) An interactive proof is $(\boldsymbol{Setup}, \mathcal{P}, \mathcal{V})$ is a perfect special honest verifier zero knowledge (PHVZK) argument of knowledge for $\mathcal{R}$ if there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that all interactive adversaries $\mathcal{A}$ have the following property for every $(x, w, \sigma) \leftarrow \mathcal{A}(pp) \wedge (pp, x, w) \in \mathcal{R}$, where σ stands for verifier's public coin randomness for challenges

$$Pr\left[\mathcal{A}(tr) = 1 \;\middle|\; \begin{array}{c} pp \leftarrow Setup(1^\lambda), \\ tr \leftarrow \langle \mathcal{P}(pp,x,w), \mathcal{V}(pp,x) \rangle \end{array} \right] =$$

$$Pr\left[\mathcal{A}(tr) = 1 \;\middle|\; \begin{array}{c} pp \leftarrow Setup(1^\lambda), \\ tr \leftarrow \mathcal{S}(pp,x,\sigma) \end{array} \right]$$

Above property states that the adversary chooses a distribution over statements x and witnesses w but is not able to distinguish between the simulated transcripts and the honestly generated transcripts for a valid statement/witnesses pair, and that the simulator has access to the randomness used by the verifier.

Definition 5. (Public Coin) All messages sent from $\mathcal{V}$ to $\mathcal{P}$ are chosen uniformly at random and independently of $\mathcal{P}$'s messages.

3.3 Zero Knowledge Proof of Discrete Logarithm

For a prover to prove it has the knowledge of a discrete logarithmic κ of some group element $s = g^\kappa \in \mathbb{G}$. We define the relation for this protocol as $\mathcal{R}_{PoD} = \{(h, s; \kappa) : s = g^\kappa\}$. We also define two functions $(\boldsymbol{ProveDL}, \boldsymbol{VerifyDL})$ for provers and verifiers to create and verify proof transcripts:

- $\boldsymbol{ProveDL}(g, \kappa) \rightarrow tr_\kappa$ generates the proof transcript tr_κ, where κ is the witness
- $\boldsymbol{VerifyDL}(g, s, tr_\kappa) \rightarrow b \in \{0, 1\}$ takes a proof transcript tr_κ and a pair of group elements with discrete log relation $(g, s \in \mathbb{G} \wedge s = h^\kappa)$, and outputs $true$ if the knowledge of the relation is verified, $false$ otherwise

In this paper, we assume the underlying implementation of the proof of discrete logarithm protocol is Schnorr's protocol [33]. We know for a fact that Schnorr's protocol has perfect completeness, special honest verifier zero knowledge, and computational witness-extended emulation.

3.4 Notations

Let $\mathbb{G}$ denote any type of secure cyclic group of prime order p, and let $\mathbb{Z}_p$ denote an integer field modulo p. Group elements other than generators are denoted by capital letters. e.g., $C = u_1^{a_1} u_2^{a_2} ... u_n^{a_n} \in \mathbb{G}$ is a commitment committed to a vector $\vec{a}$ denoted by a capital letter, and $B \in \mathbb{G}$ is a random group element also denoted by a capital letter. For generators used as base points to compute other group elements in our protocol, such as $\vec{g}, h \in \mathbb{G}$, we use lower case letters to denote them. Greek letters are used to label hidden key values. e.g., υ is the blinding key for Pedersen commitment P on generator $h \in \mathbb{G}$ s.t. $P = g^a h^\upsilon$. Finally, we use standard vector notation $\vec{v}$ to denote vectors. i.e., $\vec{a} \in \mathbb{Z}_p^n$ is a list of n values a_i for $i = \{1, 2, ..., n\}$ in $\mathbb{Z}_p$.

We write $\mathcal{R} = \{(Public\ Inputs\ ; Witnesses) : Relation\}$ to denote the relation $\mathcal{R}$ using the specified public inputs and witnesses.

4 High Level Idea

While post-quantum setting is possible, we explain our protocol in the discrete logarithm setting because it is more straight forward and likely more efficient for benchmark testing.

The relation for the base version of our protocol is as follows: for l input parameters, let $\mathcal{C}_\mathbb{F}$ represent the set of arbitrary arithmetic circuits in $\mathbb{F}$, there exists a zero knowledge argument for the relation:

$$\begin{aligned} \{(g, h, \vec{P}, R \in \mathbb{G}, E \in \mathcal{C}_\mathbb{F}\ ;\ \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_p) :\ & E(\vec{a}) = r \\ \wedge\ & P_i = g^{a_i} h^{\upsilon_i} \wedge\ R = g^r h^\phi\} \end{aligned} \tag{1}$$

Instead of just prove the knowledge of witnesses going into a circuit, our approach allows the verifier to validate the mapping of input and output witnesses to predefined commitments s.t. both inputs and outputs of a circuits are of the same format.

g, h are initial public parameters pp generated during setup. The above relation states that each input parameter to a circuit is represented by a commitment P_i in $\mathbb{G}$, which hides each input value a_i with a blinding key υ_i. r is the output of circuit E computed from inputs $\vec{a}$, which is also committed in $R \in \mathbb{G}$ with blinding key ϕ.

The main idea behind the "input transformation" concept is the process of transforming committed inputs in $\mathbb{G}$ to linear polynomials in $\mathbb{Z}_q$ so that the verifier can perform addition and multiplication operations "as is". For an input commitment P_i s.t. $P_i = g^{a_i} h^{\upsilon_i} \in \mathbb{G}$ where a_i is the input value and υ_i is its blinding key, its transformed value linear polynomial is $a_i{}' \in \mathbb{Z}_q$ is :

$$a_i{}' = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \tag{2}$$

x is the challenge provided by the verifier during runtime, and the blinding key of each input is replaced by a random α_i s.t. $\alpha_i \neq \upsilon_i$. Likewise, the circuit output commitment $R = g^r h^\phi \in \mathbb{G}$ also has a matching linear polynomial in $\mathbb{Z}_q$ with blinding key ϵ.

$$r' = r + x \cdot \epsilon \in \mathbb{Z}_q \tag{3}$$

Where r' is "directly computed" from a list of inputs a'_i ($i = \{1, ..., l\}$) by the verifier. Since inputs are in $\mathbb{Z}_q$, verifiers can perform arithmetic operations on them just as they do on decrypted numbers. The output value of a circuit evaluation is now a polynomial with p_d degrees evaluated at point x. The constant term of the output polynomial is the circuit output r, and the coefficient of the degree one term is the blinding key ϵ of the circuit output.

In the next two sections, we explain our protocol in two steps: In the first step, we introduce a sub-protocol (Protocol InputMapping) that allows the prover to prove each input in $\mathbb{G}$ is correctly transformed to that in $\mathbb{Z}_q$; In the second step, we introduce the full protocol (Protocol AriCircuit) that uses the aforementioned sub-protocol to validate transformed inputs and proves the circuit output is correctly computed from circuit inputs as relation 15 states. Before jumping into details, there are three key techniques that our protocol leverages that are important to go over:

Use of two primes We use two prime domains p, q in our protocol. All transformed inputs are in field $\mathbb{Z}_q$, where the verifier can perform circuit-related computations, and commitments are in $\mathbb{G}$ of order p. There are two simple reasons for using $\mathbb{Z}_q$ in circuit computation: 1) Prime p is not Number Theoretic Transform (NTT) friendly. 2) It is cheaper to work in a smaller field (we default q to a 61-bit prime).

Hiding data in $\mathbb{Z}_q$ with a significantly larger value in $\mathbb{F}$ (no modulo operation) We use significantly larger numbers in $\mathbb{F}$ to hide data in $\mathbb{Z}_q$. For example, if we want to hide some 61-bit value $r \in \mathbb{Z}_q$. The prover can use a 141-bit random value ω to hide it s.t. $r + \omega \in \mathbb{F}$ will perfectly hide r except for a negligible chance of at least 2^{-80} (e.g., when ω is too large or too small, the prover just picks another ω).

The reason we use this hack is because we often perform operations on committed values in $\mathbb{G}$ with order p. If the committed value overflows p after some operation (e.g., for $r, x \in \mathbb{Z}_p$, the exponent of g^{rx} will almost certain overflow p), then we cannot use mod q on the opening of the commitment to get $rx \in \mathbb{Z}_q$ without changing r to something else.

Polynomial degree of a circuit As mentioned earlier, the degree size of the polynomial generated by our protocol is not necessarily tied to the number of multiplications. In most real-world use cases, we believe the p_d value generated by our protocol is smaller than the total number of multiplications.

This is because every time an "addition or subtraction" operation is performed, the p_d value of two input wires merges. For example, if the first input wire ${a'_1}^7$ has $p_d = 7$ and the second input wire ${a'_1}^6$ has $p_d = 6$, their sum is an output wire with merged $p_d = 7$. The more addition/subtraction operations we have in a circuit, the smaller the p_d value compared to the total gate count. The sum-of-products circuit explained earlier only has a multiplication depth of 1.

Another technique we use to keep the p_d value small is to allow the protocol designer to "bootstrap" the p_d value by breaking a larger circuit into multiple smaller sub-circuits, a concept we will detail in Section 6.

5 The Sub-Protocol for Linear Polynomial to Pedersen Commitment Mapping Validation

We define a sub-protocol that validates committed inputs in $\mathbb{G}$ is correctly mapped to transformed inputs in $\mathbb{Z}_q$, which is defined by the following relation:

$$\{(g, h, \vec{P} \in \mathbb{G}, \vec{a}' \in \mathbb{Z}_q \,;\, \vec{a}, \vec{\alpha} \in \mathbb{Z}_q, \vec{\upsilon} \in \mathbb{Z}_p) : P_i = g^{a_i} h^{\upsilon_i} \ \wedge \ a'_i = a_i + X\alpha_i\} \tag{4}$$

X is the challenge generated during runtime, so $\vec{a}'$ is only available during runtime. The relation above says that for any commitment P_i of value a_i and blinding key υ_i, the prover will provide a'_i during runtime that s.t. $a'_i = a_i + x\alpha_i$ for some random public coin challenge x.

In this relation, we assume the committed inputs are in Pedersen commitment format because it has simple representation and wide adaptability. However, this is not required. For example, one can develop a commitment-to-linear polynomial mapping validation algorithm for a post-quantum commitment. One may also develop an input-mapping validation algorithm optimized to achieve quasi-scalability relative to witness size (see supplemental material - Appendix D, which is also post-quantum friendly). Since there are many ways to build this input-mapping validation protocol and we are just giving one such example, you may skip this section entirely and jump to section 6.

We make sure the domain of transformed input in $\mathbb{Z}_q$ is friendly to NTT. When multiplying two polynomials of degree p_d, the trivial approach to compute the output polynomial's coefficients would require a runtime cost of $O({p_d}^2)$, whereas the NTT-based approach would reduce that to $O(p_d \log p_d)$. Prime q is also expected to be a lot shorter than p in bits, therefore improving the runtime cost while at the same time lowering the communication cost at the same time.

Setup Phase Before the random challenge x is available, the prover must commit to all witnesses for each ith input before the challenge point x is known.

S_i commits to the randomly generated blinding keys $\omega_i \in \mathbb{Z}_p$ used in transformation validation, and η_{si} is its blinding key.

$$S_i = g^{\omega_i \cdot q} h^{\eta_{si}} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{5}$$

The second commitment T_i commits to the new blinding key α_i used to hide data in transformed input $a'_i \in \mathbb{Z}_q$, and η_{ti} is its blinding key.

$$T_i = g^{\alpha_i} h^{\eta_{ti}} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{6}$$

The setup phase of the protocol is detailed below. This part is called before the random challenge x is generated.

Input : $(; \vec{a}, \vec{\alpha}, \vec{\omega} \in \mathbb{Z}_q)$

$\mathcal{P}$ *compute* :

$\eta_{s_i}, \eta_{t_i} \xleftarrow{\$} \mathbb{Z}_p$	$i = \{1, ..., l\}$
$S_i = g^{\omega_i \cdot q} h^{\eta_{si}} \in \mathbb{G}$	$i = \{1, ..., l\}$
$T_i = g^{\alpha_i} h^{\eta_{ti}} \in \mathbb{G}$	$i = \{1, ..., l\}$

$\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}, \vec{\eta_t}, \vec{\eta_s}$

Protocol InputMapping - Setup

Once the setup phase completes, the prover then sends S_i, T_i for $i = \{1, .., l\}$ to the verifier.

Validation Phase After the random challenge x is generated, the prover computes $\vec{a}'$ and sends them to the verifier. Next, the protocol checks the mapping between transformed inputs in $\mathbb{Z}_q$ to those in group $\mathbb{G}$.

For each a'_i, the prover provides transcript e_i, which is used to convert the blinding element $x\alpha \in \mathbb{Z}_q$ of each a'_i to its raw form $x\alpha \in \mathbb{F}$ (without mod q).

$$e_i = (x\alpha_i - (x\alpha_i \bmod q)) \cdot x + \omega_i \cdot q \in \mathbb{F} \qquad i = \{1, ..., l\} \tag{7}$$

The part of e_i on the left of the addition sign $((x\ \alpha_i - (x\ \alpha_i \bmod q)) \cdot x)$ is around $\approx$ 183-bits, which is small enough s.t. an adversary can use brutal force attack to extract α_i. We can rewrite this left part to $s_i\ \cdot q$ for some $|s_i| \leq$ 122-bits and equation 7 to $e_i = (s_i + \omega_i) \cdot q$. The prover needs to use a blinding key ω_i in $\mathbb{Z}_p$ (130-bits larger than s_i) to perfectly hide it, except for a negligible probability of at most 2^{-130}. This makes $|e_i| \leq |\omega| + |q| \approx$ 313-bits.

With transcripts S_i, T_i, e_i and the transformed input a'_i, we can compute a "public key" PK s.t. its witness is the exponent of the generator h:

$$h^{\eta_{si} + \upsilon_i x + \eta_{si} x^2} = \frac{S_i \cdot P_i^x \cdot T_i^{x^2}}{g^{a'_i \cdot x + e_i}} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{8}$$

Where $\eta_{si} + \upsilon_i x + \eta_{si} x^2$ is the secret key. Using another challenge $k \in \mathbb{Z}_p$, we can create a public key for validating all P_i to a'_i mappings at once:

$$PK = \prod_{i=1}^{l} \left(\frac{S_i \cdot P_i^x \cdot T_i^{x^2}}{g^{a'_i \cdot x + e_i}} \right)^{k^i} \in \mathbb{G} \tag{9}$$

The verifier can confirm the correctness of all transformations if the prover can prove the knowledge of $\upsilon_t = \sum_{i=1}^{l}(\eta_{si} + \upsilon_i x + \eta_{si}x^2) \cdot k^i$ on generator $h \in \mathbb{G}$.

We use Shnorr's protocol to perform the check on the knowledge of witness υ_t. To ensure soundness, note that only elements carrying input data (P_i and a'_i) are taking to the first power x, and all other committed values S_i, T_i are either taking to the second power x^2 or not at all.

Finally, the verifier validates that e_i doesn't alter the value of a_i. This can be done by taking the modulus q of e_i which must return 0. This is trivial to understand since a'_i is in $\mathbb{Z}_q$ so e_i must be a multiple of q.

$$\textbf{if } (e_i \bmod q) \stackrel{?}{=} 0, \textbf{ then } \textit{continue} \tag{10}$$

We have so far skipped the overflow problem. If $a_i + (x\ \alpha_i \bmod q)\ > q$, then we will have an overflow problem in equation **??** 8 when computing $a'_i \cdot x + e_i$. To get around this, the prover simply needs to check if $a_i + (x\ \alpha_i) \bmod q$ overflows q and adds $q \cdot x$ from e_i if that's the case.

$$\textbf{if } a_i + (x\alpha_i \bmod q) \geq q, \textbf{then } e_i = e_i + x \cdot q \qquad i = \{1, ..., l\} \tag{11}$$

This adjustment does not break the zero-knowledgeness of e_i since x (61-bits) is significantly smaller than e_i's blinding key ω_i (253-bits), so subtracting $x \cdot q$ from e_i's blinding term $\omega_i \cdot q$ is perfectly unnoticeable except for a negligible chance of 2^{-122}. The validation part of the input-mapping sub-protocol is defined as follows:

$Input : (\vec{P}, \vec{S}, \vec{T} \in \mathbb{G},\ \vec{a}' \in \mathbb{Z}_q; \vec{a}, \vec{\alpha}, \vec{\omega} \in \mathbb{Z}_q,\ \vec{\upsilon}, \vec{\eta_s}, \vec{\eta_t} \in \mathbb{Z}_p)$

$\mathcal{P}'s\ input : (\vec{P},\ \vec{S}, \vec{T}; \vec{a}, \vec{\alpha}, \vec{\omega}, \vec{\upsilon}, \vec{\eta_s}, \vec{\eta_t});\ \mathcal{V}'s\ input : (\vec{P}, \vec{S}, \vec{T})$

$\mathcal{P}$ *compute* :

$e_i = \ (x\ \alpha_i - (x\ \alpha_i \bmod q))x\ + \omega_i\ q; \qquad i = \{1, ..., l\}$

if $a_i + (x\ \alpha_i \bmod q)\ > q,$

then $e_i = e_i + q \cdot x \qquad i = \{1, ..., l\}$

$\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}'$

$\mathcal{V} \rightarrow \mathcal{P} : k \stackrel{\$}{\leftarrow} \mathbb{Z}_p$

$\mathcal{P}$ *compute* :

$$\upsilon_t = \sum_{i=1}^{l}\ (\eta_{si} + \upsilon_i x + \eta_{si}x^2) \cdot k^i\ \in \mathbb{Z}_p$$

$tr = ProveDL(; \upsilon_t)$

$\mathcal{P} \rightarrow \mathcal{V} : tr$

$\mathcal{V}$ *verify inputs* :

if $(e_i\ \bmod q) \stackrel{?}{=} 0,$ **then** *continue*; $\qquad i = \{1, ..., l\}$

else *reject*

$$PK = \prod_{i=1}^{l}\left(\frac{S_i \cdot P_i^x \cdot T_i^{x^2}}{g^{a'_i \cdot x - e_i}}\right)^{k^i} \in \mathbb{G}$$

if $VerifyDL(PK, tr),$ **then** *accept*

else *reject*

Protocol for InputMapping - Verify

Theorem 1. *(The Input-Mapping Protocol). The proof system presented in this section has perfect completeness, PHVZK, and CWEE.*

Proof. The perfect completeness of protocol InputMapping Validation is trivial to observe. Specifically, equation 8 says the exponent of generator g must equal to zero so that the prover can express the output of this equation using just generator h. So ignoring the exponents of h and replace $S = g^{\omega_i \cdot q}$, $T = g_i^{\alpha}$, $a_i' = a + x \cdot \alpha$, and e_i to that defined in equation 7, we can rewrite the equation 8 to the following and shows the exponent of g computes 0:

$$(\omega_i \cdot q + (a_i)x + (\alpha_i)x^2) - ((a + x \cdot \alpha)x + ((x\alpha_i - (x\alpha_i \bmod q))) \cdot x + \omega_i \cdot q) = 0 \tag{12}$$

With generator g and its exponents gone, we can trivially see the exponents of generator h in equation 8 matches that defined on the left hand side of equation 8, and batch validation with challenge k wouldn't change this result. We therefore conclude that perfect completeness holds for protocol InputMapping Validation.

To prove PHVZK for relation 115, we define a simulator $\mathcal{S}_{input}$, which calls on simulator $\mathcal{S}_{schnorr}$ to simulate transcripts in the Schnorr protocol, which we know exist for a fact. During the setup phase, $\vec{S}, \vec{T}$ are committed values, so simulation starts after the setup phase. After receiving challenge x and matching linear polynomials $\vec{a}'$ from the verifier, it simulates all proof transcripts needed to prove the mapping between committed values $\vec{P}$ and whatever $\vec{a}'$ it received.

To begin, the simulator $\mathcal{S}_{input}$ randomly generates and sends $\vec{e}^*$ (e_i^* is generated by first randomly generating a value $v_i \in \mathbb{Z}_p$ and multiplying it by q s.t. $e_i^* = v_i \cdot q$ s.t. equality 7 will pass) to the verifier. When challenge k is received, the simulator $\mathcal{S}_{input}$ calls simulator $\mathcal{S}_{schnorr}$ to generate transcript tr^* and send it to the verifier.

The verifier then follows the protocol to compute PK using $\vec{S}, \vec{T}, \vec{a}'$ and transcripts $\vec{e}^*$, and calls the $SchnorrVerify$ function to test it against the input transcript tr^*, We already know for a fact that there exists a simulator $\mathcal{S}_{schnorr}$ that can simulate transcripts for such discrete-log relations, so tr^* will pass the validation test to prove $\vec{P}$ maps to $\vec{a}'$. Given that simulators $\mathcal{S}_{input}$ and $\mathcal{S}_{schnorr}$ choose all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or compute them directly as described in the protocol, we can conclude that protocol InputMapping is PHVZK.

To prove CWEE, we construct an extractor $\mathcal{X}$ that uses extractor $\mathcal{X}_{schnorr}$ to extract witnesses from proof of knowledge transcripts (which we know exist). To start, the extractor $\mathcal{X}$ interacts with the prover and receives $\vec{S}$, $\vec{T}$ from the prover. The extractor $\mathcal{X}$ then generates a challenge x_1 and forwards it to the prover.

After receiving transformed inputs $\vec{a}'$ and transformation transcripts $\vec{e}_1, \vec{a}_1'$ from the prover, the extractor generates k_1 and then follows the protocol to get tr_1 (from the prover) and PK_1. The extractor $\mathcal{X}$ then calls the extractor $\mathcal{X}_{schnorr}$ to retrieve υ_{t1} from generator h. The extractor then rewinds and repeats this step l times to retrieve $\upsilon_{t2}, ..., \upsilon_{tl+1}$. Through interpolation, the extractor retrieves witnesses $(\eta_{si} + \upsilon_i x + \eta_{si} x^2)$ for all i in $\{1, ..., l\}$.

The extractor rewinds 2 times to repeat with challenges x_2, x_3, and then uses interpolation again to extract witnesses $\upsilon_i, \eta_{si}, \eta_{ti}$ from $(\eta_{si} + \upsilon_i x + \eta_{si} x^2)$ for i in $\{1, ..., l\}$. Using any two different challenges x_i, x_{i+1}, the extractor receives $\vec{a}_1'$ and $\vec{a}_2'$ from the prover, from which we can trivially retrieve $\vec{\alpha}$ for all $i = \{1, ..., l\}$ using the equation below.

$$a'_{1_i} - a'_{2_i} = \alpha_i(x_1 - x_2) \tag{13}$$

With $\vec{a}, \vec{\alpha}$ extracted, we can also extract ω from equation 7. Plugging witnesses $\vec{a}, \vec{\alpha}, \vec{\upsilon}, \vec{\omega}$ into generators g, h, we can re-write the left and right sides of equation 8 to:

$$h^{\sum_i^l (\eta_{si} + \upsilon_i x + \eta_{si} x^2) k_i} = \prod_{i=1}^{l} \left(\frac{g^{a_i \cdot x + (\alpha_i) \cdot x^2 + \omega_i \cdot q} h^{\eta_{si} + \upsilon_i x + \eta_{si} x^2}}{g^{a_i' \cdot x - e_i}} \right)^{k^i} \in \mathbb{G} \tag{14}$$

The equality above can only be true if exponent of g computes to 0. Take out challenge k_i and exponents of h, for each ith input we have the right hand side of the equation above can be re-written

to:

$$g^{a_i \cdot x + (\alpha_i) \cdot x^2 + \omega_i \cdot q} = g^{a_i' \cdot x - e_i} \in \mathbb{G}$$

The equality above must be true for any challenge x_i except for a negligible probability, or else we find a non-trivial discrete log problem between generatorsg, h. We therefore conclude this protocol satisfies our CWEE definition.

Another way to confirm the soundness is to just observe the equality above. Since we know $e_i \bmod q = 0$ for a fact because our code explicitly checks that, we can trivially observe that no other value besides a_i in g's exponent ($a_i' \cdot x = a_i \cdot x + (\alpha_i x) \cdot x$) is multiplied by the single power of x. This implies the equality above must be true for a computationally bounded prover except for a negligible probability (the adversary guessed x correctly), or we find a non-trivial relationship between generators g, h.

6 Protocol for Arbitrary Circuits

We first define the relation for the base version of our protocol. For l input parameters, let $\mathcal{C}_{\mathbb{F}}$ represent the set of arbitrary arithmetic circuits in $\mathbb{F}$, there exists a zero knowledge argument for the relation:

$$\begin{aligned} \{(g, h, \vec{P}, R \in \mathbb{G}, E \in \mathcal{C}_{\mathbb{F}} \,;\, \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_p) :\ & E(\vec{a}) = r \\ \wedge\ & P_i = g^{a_i} h^{v_i} \wedge\ R = g^r h^{\phi}\} \end{aligned} \tag{15}$$

g, h are initial public parameters pp generated during setup. The above relation states that each input parameter to a circuit is represented by a commitment P_i in $\mathbb{G}$, which hides each input value a_i with a blinding key v_i. r is the output of circuit E computed from inputs $\vec{a}$, which is also committed in $R \in \mathbb{G}$ with blinding key ϕ.

The main idea behind the "input transformation" concept is the process of transforming committed inputs in $\mathbb{G}$ to linear polynomials in $\mathbb{Z}_q$ so that the verifier can perform addition and multiplication operations "as is". For an input commitment P_i s.t. $P_i = g^{a_i} h^{v_i} \in \mathbb{G}$ where a_i is the input value and v_i is its blinding key, its transformed value linear polynomial is ${a_i}' \in \mathbb{Z}_q$ is :

$${a_i}' = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \tag{16}$$

x is the challenge provided by the verifier during runtime, and the blinding key of each input is replaced by a random α_i s.t. $\alpha_i \neq v_i$. Likewise, the circuit output commitment $R = g^r h^{\phi} \in \mathbb{G}$ also has a matching linear polynomial in $\mathbb{Z}_q$ with blinding key ϵ.

$$r' = r + x \cdot \epsilon \in \mathbb{Z}_q \tag{17}$$

Where r' is "directly computed" from a list of inputs a_i' ($i = \{1, ..., l\}$) by the verifier. Since inputs are in $\mathbb{Z}_q$, verifiers can perform arithmetic operations on them just as they do on plain-text values. The output of the circuit execution is a polynomial with p_d degrees evaluated at point x. The constant term of the output polynomial is the circuit output r, and the coefficient of the degree one term is the blinding key ϵ of the circuit output. It is the job of our protocol to subtract out all other degree terms from the output in a verifiable way.

6.1 The Main Protocol for Circuit Evaluation

To make the circuit output binding, the prover commits to all coefficients of the output polynomial before challenge X is available. For example, for a simple circuit that computes the sum of two inputs, the prover proves the constant term r and the coefficient of the degree 1 term ϵ of the output polynomial matches the committed values:

$$o = {a_1}' + {a_2}' = r + x \cdot \epsilon \tag{18}$$

Computing the output polynomial of the addition operation is the same as adding two polynomials, where $r = (a_1 + a_2)$ and its blinding key $\epsilon = (\alpha_1 + \alpha_2)$. Likewise, multiplying two inputs $a_1{}', a_2{}'$ is the same as multiplying two polynomials:

$$o = a_1{}' \cdot a_2{}' = r + x \cdot \epsilon + x^2 \cdot \tau \tag{19}$$

Where $r = a_1 \cdot a_2$, $\epsilon = a_2\alpha_1 + a_1\alpha_2$, and $\tau = \alpha_1 \cdot \alpha_2$. We use the label "$o$" to represent the circuit output, which is equivalent to the output polynomial evaluated at point x. To make our protocol binding, the prover commits to coefficients r, ϵ, τ.

To get the linear polynomial r' from the raw output o, the verifier needs to subtract out terms with degrees higher than one. In the multiplication circuit above, the verifier needs to eliminate the degree 2 term. To do so, the prover sends $y = x^2\tau$ to the verifier when challenge x is available. The verifier can then compute the output in the linear polynomial form defined in equation 17:

$$r' = o - y \tag{20}$$

Since r, ϵ, τ are all committed values, the verifier can validate that r', y, o satisfies the committed values at any challenge point x and that o is the multiplicative product of transformed inputs a'_1, a'_2.

We call each value y a "breaker." Breaker(s) subtracts all "noises" (polynomial terms of degree higher than one) from the raw circuit output o.

In most arithmetic circuits with both addition/subtraction and multiplication/division operations, its output's polynomial degree may be a lot smaller than the total number of multiplication operations. This is because every time we add two output wires, their polynomials get merged (e.g., circuit $o = a_1'^7 + a_2'^3 \cdot a_3'^5 + a_4'^6$ only outputs a polynomial of degree 8, but 21 multiplications are performed). However, there are special cases where the increase of the p_d value exceeds the increase in the total number of multiplications.

What we need is 1) a mechanism to allow the prover to repeatedly reset the polynomial degree back to 1 so that the penalty of high-degree polynomials can be avoided, and 2) a mechanism to commit to only a sub-linear number of the total p_d coefficients generated by a circuit.

Breaking a large circuit into "s" smaller circuits We break the circuit we are evaluating into "s" sub-circuits and then batch evaluate them in one run. So when the polynomial degree of a sub-circuit i reaches degree $b + 1$, we reset it to a linear polynomial by evaluating the sub-circuit. The output of each sub-circuit is then used as the input to the next sub-circuit (Figure 1).

$$\begin{matrix} o_1 \\ o_2 \\ . \\ . \\ o_s \end{matrix} = \begin{pmatrix} r_1 & \epsilon_1 & \tau_{1,1} & \tau_{1,2} & . & . & \tau_{1,b} \\ r_2 & \epsilon_2 & \tau_{2,1} & \tau_{2,2} & . & . & \tau_{2,b} \\ . & . & . & . & . & & \\ . & . & . & . & & . & \\ r_s & \epsilon_s x & \tau_{s,1} & \tau_{s,2} & . & . & \tau_{s,b} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

Figure 1

We evaluate each sub-circuit $i = \{1, ..., s\}$ in the same way we evaluate the full circuit. The raw output of each sub-circuit o_i is reduced to its linear polynomial form $r'_i = r_i + x \cdot \epsilon_i$ using breaker $y_i = \sum_{j=1}^{b} \tau_{i,j} \cdot x^{j+1}$.

$$r'_i = o_i - y_i \qquad \text{for} \qquad i = \{1, ..., s\} \tag{21}$$

Each o_i is the output of each sub-circuit at evaluation point x, and each breaker y_i is the evaluation output of that sub-polynomial minus the constant term (r, sub-circuit output) and the degree 1 term (ϵ, the blinding key of they sub-circuit output), see Figure 2. Since the output of each sub-circuit is in

the same linear polynomial format as inputs to any circuit, they can be reused as inputs to subsequent sub-circuits.

$$\begin{matrix} o_1 \\ o_2 \\ . \\ . \\ o_s \end{matrix} = \underbrace{\begin{pmatrix} r_1 & \epsilon_1 \\ r_2 & \epsilon_2 \\ . & . \\ . & . \\ r_s & \epsilon_s \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix}}_{outputs} + \underbrace{\begin{pmatrix} \tau_{1,1} & \tau_{1,2} & . & . & \tau_{1,b} \\ \tau_{2,1} & \tau_{2,2} & . & . & \tau_{2,b} \\ . & . & . & & . \\ . & . & & . & . \\ \tau_{s,1} & \tau_{s,2} & . & . & \tau_{s,b} \end{pmatrix} \begin{pmatrix} x^2 \\ x^3 \\ . \\ . \\ x^{b+1} \end{pmatrix}}_{breakers}$$

Figure 2

By proving prior knowledge of all coefficients of each sub-circuit (e.g., each row of Figure 2), we know the output r'_i of each sub-circuit is correct. Sub-circuits are arranged according to the computation order of the full circuit. If outputs of sub-circuits are correct, then the output of the final sub-circuit must also be the output of the full circuit.

It is worth noticing that the prover can use breaker(s) to bootstrap polynomial degree anywhere on a circuit depending on the application design. For simplicity, we assume all breaker are set at $b+1$ degrees, where $b = p_d/s$.

Make group exponentiation operations sub-linear to the polynomial degree of the full circuit Using polynomial commitment schemes to evaluate all sub-circuits would be expensive. In our case, the verifiers already know the evaluation output o_i of each sub-polynomial (sub-circuit) by directly computing them with transformed inputs. Knowing this we can efficiently batch evaluate polynomials by 1) committing and evaluating the output (r_i) of each i-th sub-circuit/sub-polynomial, 2) using an intermediary challenge "w" to batch commit the rest of the coefficients used for bootstrapping.

First, we commit and evaluate the outputs of each sub-circuit using a PCS scheme. Since the prover and verifier runtime cost for running PCS in our protocol is insignificant compared to the circuit size, we pick a group based PCS to minimizing the communication cost. The one we use in our protocol is based on the one defined by Bootle et al. [11]. We define two functions that our protocol will use to commit and evaluate coefficients $\vec{z}$ at evaluation point x.

- ***polyCommit***$(\vec{c}) \rightarrow \vec{C}$: commit coefficients $\vec{c}$ using generators $\vec{g}$, returns commitments $\vec{C} \in \mathbb{G}$ s.t. $|\vec{C}| = \sqrt{|\vec{c}|}$.
- ***PolyEval***$(\vec{C}, y, x; \vec{c}) \rightarrow b \in \{0, 1\}$: $\vec{c}$ are the coefficients of univariate polynomial $f(X)$ committed in $\vec{C}$. At evaluation point x, this function verifies if $f(x) = y$.

Instead of using Pedersen commitments to commit each output as we do with the circuit output R, we use polynomial commitment to batch commit all sub-circuit outputs $\vec{r}$ and their blinding keys $\vec{\epsilon}$.

$$r_\beta, \epsilon_\beta \xleftarrow{\$} \mathbb{Z}_q \tag{22}$$

$$\text{polyCommit}(r_\beta || \vec{r}) \rightarrow : \vec{R_s}; \quad \text{polyCommit}(\epsilon_\beta || \vec{\epsilon}) \rightarrow : \vec{E_s} \tag{23}$$

To avoid possible leaks of data (in boolean circuits, polynomial evaluation output may leak information since $r_i \in \{0, 1\}$), we create a blinding output $r'_\beta = r_\beta + x \cdot \epsilon_\beta$ where r_β, r_ϵ are blinding keys, and let it be the output of the 0th sub-circuit.

We will later use a challenge point z as the evaluation point to evaluate the polynomial commitment. However, this is the last step of our protocol because we need the prover to commit (by sending)

all other transcripts before we can evaluate these two polynomials.

$$y_r = r_\beta + \sum_{i=1}^{s} r_i \cdot z^i \in \mathbb{Z}_q; \qquad y_\epsilon = \epsilon_\beta + \sum_{i=1}^{s} \epsilon_i \cdot z^i \in \mathbb{Z}_q \tag{24}$$

Specifically, the prover sends sub-circuit outputs $r'_1, ..., r'_s$ to the verifier before evaluation point z is known, which allows the verifier to use the following equalities to check if $\vec{r}'$ match committed values.

$$\text{polyEval}(\vec{R_s}, y_r, z; \vec{r}) \wedge \tag{25}$$

$$0 = (y_r + y_\epsilon \cdot x - (r'_\beta + \sum_{i=1}^{s} (r'_i \cdot z^i))) \bmod q \tag{26}$$

Second, we use another intermediary challenge w to allow the prover to send the polynomial evaluation at point w for the j-th term coefficient set $\vec{\tau_j}$ s.t. $c_j = f(\vec{\tau_j})$. The "batched" j-th coefficient is defined as:

$$\tau_{i,0} \xleftarrow{\$} \mathbb{Z}_q, \qquad i = \{1, ..., s\} \tag{27}$$

$$c_j = \sum_{i=1}^{s} \tau_{i,j} \cdot w^i \in \mathbb{Z}_q \qquad j = \{0, ..., b\} \tag{28}$$

The first element c_0 $(j = 0)$ is computed from a set of blinding keys $\tau_{1,0}, ..., \tau_{s,0}$ we randomly generated to update each blinding key ϵ_i to cover special cases when $\epsilon_i = 0$ (which may leak data).

$$\epsilon_i = \epsilon_i - \tau_{i,0} \in \mathbb{Z}_q \tag{29}$$

Both challenge w and evaluations $\vec{c_j}$ are made public before challenges x, z are available. The updated matrix with blinding elements $\tau_{1,0}, ..., \tau_{s,0}$ is presented in Figure 3 below.

$$\begin{matrix} y_1 w \\ y_2 w^2 \\ . \\ . \\ y_s w^s \end{matrix} = \begin{pmatrix} \tau_{1,0} w & \tau_{1,1} w & \tau_{1,2} w & . & . & \tau_{1,b} w \\ \tau_{2,0} w^2 & \tau_{2,1} w^2 & \tau_{2,2} w^2 & . & . & \tau_{2,b} w^2 \\ . & . & . & . & & . \\ . & . & . & & . & . \\ \tau_{s,0} w^s & \tau_{s,1} w^s & \tau_{s,2} w^s & . & . & \tau_{s,b} w^s \end{pmatrix} \begin{pmatrix} x \\ x^2 \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

Figure 3

When challenge x is available, the verifier receives $\vec{r}'$ from the prover and computes $\vec{y}$:

$$y_i = o_i - r'_i \qquad \text{for} \qquad i = \{1, ..., s\} \tag{30}$$

Since $\vec{o}$ is directly computed by the verifier using transformed inputs and $\vec{r}'$ gets validated with PCS schemes when the evaluation point z is available, we can assume $\vec{y}$ is correct if transformed inputs and $\vec{r}'$ are validated.

We can then use $\vec{y}$ to validate $\vec{c}$ by multiplying each y_i by w^i and each c_j by x^{j+1}. The equality below must be true except for a negligible probability.

$$\sum_{j=0}^{b} c_j \cdot x^{j+1} = \sum_{i=0}^{s} y_i \cdot w^i \in \mathbb{Z}_q \tag{31}$$

To show why this is sound for all sub-circuit outputs. Let's say $r_i^* = r_i - \delta_i$ for some arbitrary δ_i, the dishonest prover needs to find a set $\vec{\Delta}$ before challenge x is known s.t.

$$\sum_{j=0}^{b}(c_j + \Delta_j) \cdot x^{j+1} = \sum_{i=0}^{s}(y_i + \delta_i) \cdot w^i \in \mathbb{Z}_q$$

Taking out the "correct part", the equality above can be rewritten as the equality below, which clearly shows such $\vec{\Delta}$ cannot be found without prior knowledge of challenge x.

$$\sum_{j=0}^{b}\Delta_j \cdot x^{j+1} = \sum_{i=0}^{s}\delta_i \cdot w^i \in \mathbb{Z}_q$$

Lastly, sending $\vec{c}$ to the verifier may not be safe, the more secure and communication-cost-efficient way is to use a polynomial commitment scheme to commit and evaluate $\vec{c}$ at evaluation point x.

$$\text{polyCommit}(\vec{c}) \rightarrow: \vec{C} \tag{32}$$

Let $y_c \in \mathbb{Z}_q$ be the evaluation output at point x for a univariate polynomial with coefficients $\vec{c}$ committed in $\vec{C}$. The equation 31 is now updated to:

$$\text{polyEval}(\vec{C}, y_c, x; \vec{c}) \ \wedge \ y_c = (\sum_{i=1}^{s} y_i \cdot w^i) \bmod q \tag{33}$$

We now have all the necessary pieces to describe our main protocol for arithmetic circuits.

6.2 The Complete Protocol For Arithmetic Circuits

We define two more functions for our protocol definition. function
computeSubCircuitKeys is used by the prover to compute the keys of each sub-circuit or "row" in Figure 3, and function
computeSubCircuit is used by the verifier to compute the value of a sub-circuit at the evaluation point x:
1. function $\boldsymbol{computeSubCircuitKeys}_i$("input values", "input keys"): for $i = \{1, .., s\}$, it takes input values $\vec{a}$ and keys $\vec{\alpha}$ to evaluate the sub-circuit and outputs $r_i, \epsilon_i, \vec{\tau}_i$ (coefficients of o_i).
2. function $\boldsymbol{computeSubCircuit}_i$ ("inputs in linear polynomial form", "output from the previous computeSubCircuit function"): for $i = \{1, .., s\}$, it trivially computes the result o_i from the inputs to the sub-circuit.

For example, if the logic of the ith sub-circuit is to return the product of l inputs, then the $computeSubCircuit_i$ function simply performs $o_i = a'_1 \times a'_2 \times, ..., \times a'_l$. Since $a'_1, ..., a'_l$ are linear polynomials evaluated at point x, o_i is the evaluation of the output polynomial at point x, and the $computeSubCircuitKeys_i$ function computes all coefficients of the output polynomial. We are now ready to introduce the complete version of our protocol - Protocol AriCircuit - as follows:

$$Input : (\vec{P}, R \in \mathbb{G};\ \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_p) \tag{34}$$

$$\mathcal{P}'s\ input : (\vec{P}, R;\ \vec{a}, \vec{v}, r, \phi);\ \ \mathcal{V}'s\ input : (\vec{P}, R) \tag{35}$$

$$\mathcal{P}\ compute : \tag{36}$$

$$\alpha_i \xleftarrow{\$} \mathbb{Z}_q, \qquad i = \{1, ..., l\} \tag{37}$$

$$\tau_{i,0} \xleftarrow{\$} \mathbb{Z}_q, \qquad i = \{1, ..., s\} \tag{38}$$

$r_\beta, \epsilon_\beta \xleftarrow{\$} \mathbb{Z}_q$ (39)

for $i = 1, ..., s$ { (40)

$r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}', r_i, \epsilon_i, \vec{\tau}_i);$ (41)

$\epsilon_i = (\epsilon_i - \tau_{i,0}) \bmod q$ } (42)

$\text{polyCommit}(r_\beta || \vec{r}) \rightarrow: \vec{R}_s;\ \text{polyCommit}(\epsilon_\beta || \vec{\epsilon}) \rightarrow: \vec{E}_s$ (43)

$\text{InputMapping-Setup}(; \vec{a} || r_s, \vec{\alpha} || \epsilon_s, \vec{v} || \phi) \rightarrow: \vec{S}, \vec{T}, \vec{\eta_s}, \vec{\eta_t}$ (44)

$\mathcal{P} \rightarrow \mathcal{V} : \vec{R}_s, \vec{E}_s, \vec{S}, \vec{T}$ (45)

$\mathcal{V} \rightarrow \mathcal{P} : \quad w \xleftarrow{\$} \mathbb{Z}_p$ (46)

$\mathcal{P}$ *compute* : (47)

$$c_j = \sum_{i=1}^{s} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{0, ..., b\} \tag{48}$$

$\text{polyCommit}(\vec{c}) \rightarrow: \vec{C}$ (49)

$\mathcal{P} \rightarrow \mathcal{V} : \vec{C},$ (50)

$\mathcal{V} \rightarrow \mathcal{P} : \quad x \xleftarrow{\$} \mathbb{Z}_p$ (51)

$\mathcal{P}$ *compute* : (52)

$a'_i = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$ (53)

$r'_i = r_i + x \cdot \epsilon_i \in \mathbb{Z}_q \qquad i = \{\beta, 1, ..., s\}$ (54)

$$y_c = \sum_{j=0}^{b} c_j \cdot x^j \in \mathbb{Z}_q \tag{55}$$

$\mathcal{P} \rightarrow \mathcal{V} : \vec{a}', \vec{r}', y_c$ (56)

$\mathcal{V} \rightarrow \mathcal{P} : \quad z \xleftarrow{\$} \mathbb{Z}_q$ (57)

$\mathcal{P}$ *compute* : (58)

$$y_r = r_\beta + \sum_{i=1}^{s} r_i \cdot z^i \in \mathbb{Z}_q; \qquad y_\epsilon = \epsilon_\beta + \sum_{i=1}^{s} \epsilon_i \cdot z^i \in \mathbb{Z}_q \tag{59}$$

$\mathcal{P} \rightarrow \mathcal{V} : y_r, y_\epsilon$ (60)

$\mathcal{V}$ *verify final output* : (61)

for $i = 1, ..., s$ { (62)

$o_i = computeSubCircuit_i(\vec{a}' || \vec{r}') \in \mathbb{Z}_p$ (63)

$y_i = o_i - r'_i \in \mathbb{Z}_q$ } (64)

if $(\text{polyEval}(\vec{R}_s, y_r, z; \vec{r}) \ \wedge \ \text{polyEval}(\vec{E}_s, y_\epsilon, z; \vec{\epsilon})$ (65)

$$\wedge\ 0 = (y_r + y_\epsilon \cdot x - (r'_\beta + \sum_{i=1}^{s} (r'_i \cdot z^i))) \bmod q \tag{66}$$

$\wedge\ \text{polyEval}(\vec{C}, y_c, x; \vec{c})$ (67)

$$\wedge\ y_c = (\sum_{i=1}^{s} y_i \cdot w^i) \bmod q) \textbf{ then } continue \tag{68}$$

else *reject* (69)

if $\text{InputMapping-Verify}(\vec{P} || R, \vec{S}, \vec{T}, \vec{a}' || r'_s;$ (70)

$$\vec{\eta_s}, \vec{\eta_t}, \vec{a}||r_s, \vec{\alpha}||\epsilon_s, \vec{v}||\phi) \tag{71}$$

then *continue* (72)

else *reject* (73)

Protocol AriCircuit

Theorem 2. *(Protocol AriCircuit). The proof system presented in this section has perfect completeness, PHVZK, and CWEE.*

Proof. The perfect completeness of protocol AriCircuit is trivial to observe.

To prove PHVZK for relation 15, we define a simulator $\mathcal{S}$. Simulator $\mathcal{S}$ calls on simulators $\mathcal{S}_{input}$ defined earlier to simulate interactions in the InputMapping sub-protocol used in our AriCircuit protocol, and uses Simulator $\mathcal{S}_P$, which we know exist for the PCS scheme we use, to simulate interactions in the PCS scheme our protocol calls.

We have already shown that $\mathcal{S}_{input}$ can simulate all interactions needed in sub-protocol InputMapping and there exists a simulator $\mathcal{S}_P$ that can simulate all transcripts for PCS schemes. We now show how $\mathcal{S}$ uses simulators $\mathcal{S}_{input}$ and $\mathcal{S}_P$ to generate all transcripts according to the randomness supplied to the simulator from their respective domains or computes them directly as described in the protocol.

To start, the simulator $\mathcal{S}$ randomly generates input witnesses $\vec{a}^*, \vec{\alpha}^*$ which computes to random linear polynomials $\vec{a}'^*$. The simulator $\mathcal{S}$ then calls simulator $\mathcal{S}_{input}$ to simulate transcripts needed to prove the mapping between committed inputs $\vec{P}$ and generated linear polynomial inputs $\vec{a}'^*$.

For the circuit transcripts, the simulator $\mathcal{S}$ randomly generates PCS commitments $\vec{R}_s^*, \vec{E}_s^*, \vec{C}^*$ and sends them to the verifier. The simulator then follows the protocol to compute witnesses $\vec{r}^*, \vec{\epsilon}^*, \vec{\tau}^*$, which in turn computes to simulated transcripts $y_c^*, \vec{r}'^*, y_r^*, y_\epsilon^*$ with challenges w, x, z. The verifier just needs to follow the protocol specification to run and pass equality tests 26 and 33. For PCS evaluations, the simulator $\mathcal{S}$ calls on simulator $\mathcal{S}_P$ (which will use rewinding technique) to simulate all transcripts needed in proving these PCS commitments at their evaluation points (x, z).

All rewindings only take place inside the simulator $\mathcal{S}_{input}$ (for simulating transcripts used in the InputMapping-Verify function) and simulator $\mathcal{S}_P$ (for simulating transcripts in the polyEval function). This implies that if the input-mapping process and PCS scheme are PHVZK, then our protocol is PHVZK.

Since all elements in proof transcripts are either independently randomly distributed or their relationship is fully defined by the verification equations, we can conclude that the protocol AriCircuit is PHVZK.

To prove CWEE, we define an extractor $\mathcal{X}$ that calls on extractors $\mathcal{X}_{input}$ defined earlier to extract witnesses for the Input-Mapping sub-protocol used in AriCircuit. We also define an extractor $\mathcal{X}_P$ that extracts witnesses from a polynomial commitment, which is defined for the PCS scheme we use.

We already know $\mathcal{X}$ can extract $\vec{a}, \vec{\alpha}$ and $\vec{r}, \vec{\epsilon}$ using extractor $\mathcal{X}_{input}$ from committed transcripts $\vec{S}, \vec{T}$. If the prover is honest, it will use these input witnesses to compute circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$, so we need to show these circuit witnesses match what we we can extract from circuit transcripts $(\vec{R}_s, \vec{E}_s, y_r, y_\epsilon, \vec{C}, \vec{r}', y_c)$. If they match, we can claim CWEE for protocol AriCircuit.

First, we extract $\vec{\tau_j}$ for $j = \{1, .., b\}$ and then validate their soundness. To begin, the extractor $\mathcal{X}$ calls $\mathcal{X}_P$ to extract coefficients $\vec{c}$ from polynomial commitments $\vec{C}$ that satisfy its evaluation output y_c. With $\vec{c}$, the extractor $\mathcal{X}$ rewinds one more time to line 45 and then rewinds another s times generates $s + 1$ challenges $w_1, w_2, ..., w_{s+1}$, using which the extractor can extract witnesses $\vec{\tau_j}$ for $j \in \{0, .., b\}$ using interpolation. Although witnesses $\vec{\tau_j}$ are not committed using a PCS scheme here, its evaluation c_j is presented before challenge x is known so the set $\vec{\tau_j}$ is still "committed".

Equality 31 implies that if $y_1, ..., y_s$ on the right hand side the equality is correct and are "committed" also before both challenges w, x are known, then the equality above can only be true for any

challenge pair x, w except for a negligible probability.

$$\sum_{j=1}^{b} c_j \cdot x^{j+1} = \sum_{i}^{s} y_i \cdot w^i \in \mathbb{Z}_q$$

To check if $y_1, ..., y_s$ are correct, we check the soundness of $r'_\beta, r'_1, ..., r'_s$. The extractor $\mathcal{X}$ calls $\mathcal{X}_P$ to extract coefficients $\vec{r}, \vec{\epsilon}$ from polynomial commitments $\vec{R}_s, \vec{E}_s$ that evaluates to y_r, y_ϵ at a challenge point z. With $\vec{r}, \vec{\epsilon}$ extracted, the equality of equation 26 can be rewritten to:

$$r'_\beta + \sum_{i=1}^{s} r'_i \cdot w^i = (r_\beta + \epsilon_\beta \cdot x) + \sum_{i=1}^{s} (r_i + \epsilon_i \cdot x) \cdot w^i \in \mathbb{Z}_q \tag{74}$$

The equality above can only be true for any challenge pair x, w if blinding sub-circuit output r'_β and sub-circuit outputs $\vec{r}'$ (left hand side of the equality) are computed according to equations 16, 17 using committed witnesses $\vec{r}, \vec{\epsilon}$ shown on the right hand side of the equality above. We now updates the equality 31 to the following:

$$\sum_{j=1}^{b} c_j \cdot x^{j+1} = \sum_{i}^{s} (o_i - (r_i + x \cdot \epsilon_i)) \cdot w^i \in \mathbb{Z}_q \tag{75}$$

Knowing that $\vec{r}, \vec{\epsilon}$ are committed witnesses and that $\vec{o}$ must be correct since they are directly computed by the verifier, the equality above can only be true for any trio of challenges w, x, z if circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau_0}, ..., \vec{\tau_b}$ we extracted are correct except for a negligible probability. We therefore claim CWEE for protocol AriCircuit.

6.3 Embedding Range Proof for Comparison Operations

One of the primary reasons for using a boolean circuit over an arithmetic circuit in the real world (there are no real-world applications of trying to prove a hash) is the ability to perform comparison operations $(>, <, \geq, \leq, =)$. Our design allows the use of customized circuit(s) to embed range-proof protocols to evaluate comparison operations inside the arithmetic circuit being processed. This way, there will be fewer needs for expensive boolean circuits and/or the expensive process of decomposing/recomposing integers to bits within a circuit.

The idea is similar to that of "custom gates" found in SNARKs protocols in principle but very different in implementation. In general, the design goal of a customized circuit is to utilize existing algorithms/protocols to handle operations that would otherwise be expensive in our protocol (or any other protocol).

In particular, we can use range proof inside an arithmetic circuit to handle all comparison operations and decimal point reductions. This is huge in practice because either using a boolean circuit directly or converting to/from a boolean circuit inside an arithmetic circuit is expensive.

For example, to prove $a_1 > a_2$ (or $P_1 > P_2$), the prover can do the following:

1. Commit to their difference $C = g^c h^\upsilon$ s.t. $c = a_1 - a_2$ (or compute C from P_1, P_2 using additive homomorphism e.g. $C = P_1/P_2$).
2. Call protocol InputMapping to check $c' = a'_1 - a'_2 \in \mathbb{Z}_q$ maps $C \in \mathbb{G}$;
3. Use a range-proof protocol to prove $C > 0$. If returns true, then we know $a_1 > a_2$.

An example usage is as follows:

computeSubCircuit : $(\vec{a}' \in \mathbb{Z}^q, \vec{C} \in \mathbb{G})$

$c'_1 = a'_1 - a'_2 \in \mathbb{Z}_q$

if *Protocol RangeProof*$(C_1, 0)$

$c'_2 = a'_3 - a'_4 \in \mathbb{Z}_q$

if *Protocol RangeProof*$(C_2, 0)$

$o =$ do something

else $o =$ do something else

else $o =$ do something

if *Protocol InputMapping*$(C_1||C_2, c'_1||c'_2)$ **return** o

else *reject*

Function computeSubCircuit (Customized)

In the computeSubCircuit function defined above, the circuit first tests if $a'_1 > a'_2$ and then tests if $a'_3 > a'_4$. Before the function returns o, it batch checks the mapping between each c'_i and C_i pair. In practice, all calls to range proof should also be batch verified at the end of the function.

We can bypass the "inactive" part of the circuit (similar to that of suBlonk [19]). For example, if the first range proof returns false (e.g., $a_1 < a_2$), then both the prover and the verifier can bypass the two "else" parts of the circuit above. However, it is worth noticing that using a customized sub-circuit bypassing parts of the circuit may leak information about data to attackers, so one must use such a strategy with extreme caution.

We believe combining the arithmetic circuit and range-proof protocols is the most efficient way to run a zero-knowledge test in the real world. This is much more powerful than it seems. Besides handling comparison operators, one can also use embedded range proof to verify floating point operations (perform multiplication/division operations as full integer operations, then remove decimal places by proving their range). We believe it will allow us to use arithmetic circuits to handle almost all types of business logic that would otherwise require boolean circuits.

6.4 Memory Efficiency

The memory consumption cost of our protocol is $O(p_d)$. However, our design approach allows us to improve the memory consumption cost of our protocol to $O(b+s)$. We only need to do two things to achieve a memory cost of $O(b+s)$: First, delete $\vec{\tau}$ from memory in line 44; Second, recompute $\vec{\tau}_i$ after challenge x is available so that y_t can be computed (in line 98).

for $i = 1, ..., m$ {

$r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}')$;

$\epsilon_i = (\epsilon_i - \mu_i) \bmod q \in \mathbb{Z}_q$;

$\vec{a}' = \vec{a} \,||\, r_i, \quad \vec{\alpha}' = \vec{\alpha} \,||\, \epsilon_i$;

$z_j = z_j + \sum_{i=1}^{s} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{1, ..., b\}$; }

Since we have $b = s = \sqrt{p_d}$ in the default setting, the asymptotic memory cost is $O(\sqrt{p_d})$.

6.5 Cost Analysis

The prover runtime of our protocol is dominated by $O(\iota\, s \log s + l)$ field operations and $O(b+l)$ group exponentiations. We set $b = \sqrt{s}$ in our benchmark testing, so the cost of group exponentiations grow sub-linearly to the total polynomial degree generated by the circuit. The value of ι depends on how the

circuit is wired. For the sequential multiplication test case that we benchmark against, $\iota = m \sum_{i=1}^{\log b} i$; the verifier runtime is dominated by $O(n+l)$ field operations and $O(m+b+l)$ group exponentiations; and the communication cost is dominated by $O(b+l)$ group elements and $O(m+l)$ field elements.

Our protocol is also natively faster than its asymptotic cost indicates because group exp. operations of our protocol operate mostly in q (61-bits), which is significantly smaller than p in ECC. This gives us approximately 2.5X performance gains when performing multi-exponentiations over standard ECC multiplications. Although the verifier runtime is technically linear, it is so efficient to the point that it is close to SNARKs with trusted setup. This is because all the asymptotically slow operations are performed at field level (many papers consider this free).

It is important to note that $p_d \neq n$. For some arithmetic circuits, the total polynomial degree (p_d) can be smaller than the number of multiplications because every time we perform the "addition" operation on two output wires. On the other hand, if p_d is bigger than the total number of multiplications, then we may need to set breakers more aggressively to cut the p_d value down to an acceptable level, preferably lower than n.

It is also worth noting that it is not necessary to set all breakers at some fixed point b. For example, if we know several output wires with high degrees are going to get merged through addition operations, we may want to save the breaker until they are merged together.

7 Performance Comparison

Our test runs are performed on an Apple M4 processor (base version) . All tests are run on a single CPU thread. Our test code is a non-interactive implementation (using Fiat-Shamir heuristic). For group operations, we use the curve25519-dalek implementation, and Pippenger acceleration is applied to all sum-of-product group operations. For field operations, we use the Montgomery algorithm to accelerate modular multiplications on the prime q.

We compare our protocol against Spartan-NIZK, which is considered the fastest transparent zero knowledge protocol in the literature when using a comparable polynomial commitment scheme.

Other noticeable protocols such as Brakedown [28] that are comparable to SpartanNIZK in the prover and verifier speed generally have significantly larger proof sizes, often by more than one order of magnitude more than that of SpartanNIZK. This is primarily due to the polynomial commitment scheme (PCS) these protocols used, which offers a faster prover and verifier speed at the expense of a much larger communication cost. Had our protocol use a PCS scheme like these based on Reed-Soloman code, our prover and verifier speed may be even better in exchange for a larger communication cost.

We do not consider protocols that can't handle high depth circuits, such as GKR-based protocols, while they are efficient for shallow circuits speed-wise, they cannot handle circuits with high depth. Although some test circuits can be transformed into shallow circuits because test circuit are usually structurally simple, it is hard to achieve without significantly increasing the size of the circuit.

We set the number of inputs to our protocol to $l = 10$ integers and perform input-mapping verification test on each of them on them, this is a feature offered by our protocol that Spartan and other comparable protocols do not perform in their benchmarking as they only need to prove knowledge of witnesses. The circuit we run our test performs n sequential multiplications on l inputs. We do this by first stretching inputs $a_1, ..., a_l$ to $a_1, ..., a_n$ using copy and past.

To maximize the advantage of the NTT algorithm in computing sequential multiplications, we process each segment ($subCircuit_i$) of our circuit in binary tree format to represent layers we would see in the real world. So the total depth = Log b (layers in a sub-circuit) $\times$ s (number of sub-circuits) (e.g., total depth $\approx 10,000$ for $n = 2^{20}$ sequential multiplications).

7.1 Benchmark Results

By default, we set the bootstrapping parameter $b = s = \sqrt{p_d}$ to get a more balanced result. Alternatively, one can bootstrap more aggressively (by setting $b < \sqrt{p_d}$ value) to get better prover runtime

speed in exchange for (slightly) more expensive verifier speed and communication costs (Table 4). This is because doing so will bypass the expensive NTT computations at high degrees in exchange for more breakers ($\vec{y}$).

Table 1: Prover performance comparison (seconds)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
SpartanNIZK	0.012	0.028	0.088	0.28	0.98	3.48
This work	0.002	0.003	0.005	0.0014	0.05	0.22

Table 1 shows that as the circuit size gets bigger, the prover performance of our protocol is becoming increasingly more efficient than all the other protocols we are comparing against. This is because the cost associated with the number of inputs to the circuit is fixed (20 inputs), and its impact relative to the cost of evaluating the whole circuit gradually declines as the circuit size gets bigger (the same effect will also apply to verifier runtime and proof size benchmarks below). To the best of our knowledge, our protocol offers the best prover performance in the non-interactive setting in the literature.

Table 2: Proof size comparison (kilobytes)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
SpartanNIZK	9	12	15	21	30	48
This work	3.3	4	5.3	7	10.5	16.2

Table 2 shows that the communication cost of our protocol dominates that of Aurora, while being largely comparable to SpartanNIZK and Hyrax. The fixed cost of one additional input is 112 bytes, and other protocols also incur comparable costs when they map more witnesses to some pre-committed/encrypted value in the public data store.

Table 3: Verifier performance comparison (milliseconds)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
SpartanNIZK	4	5.6	9.8	23	74	275
This work	0.67	0.72	0.83	1	1.8	4.5

Table 3 demonstrates that our protocol achieves a significant improvement of over one order of magnitude in verifier runtime compared to other protocols.

The cost of one million sequential multiplications is approximately 2 milliseconds. It can be faster if the compiler has native support (assembly code) for Montgomery acceleration s.t. that we don't have to code that ourselves, which is not optimal. It will be much faster (probably by one order of magnitude) if CPU can natively support 64-bit modular arithmetic

The NTT friendly prime number q we used for our benchmark testing is 1945555039024054273, a 61-bit prime that implies the soundness error will be at most $\frac{b}{q} = 2^{-51}$ in our test case ($b = s = \sqrt{p_d}$), which is good enough for many applications and ones where one interaction is allowed.

When a soundness error of 2^{-51} is not enough, the straight-forward way is to run the whole protocol twice to get a soundness error of at least $(\frac{b}{q})^2 = 2^{-102}$. This will almost double the cost of everything (the prover time cost will increase by 50-100% depending on the bootstrapping parameter, while the verifier time cost and the communication cost will double), but our protocol will still claim the title of the fastest prover and verifier runtime in the literature by a wide margin. The more advanced way is to use a bigger q prime. For example, a 90-bit q prime will comfortably increase the soundness error to at least 2^{-80}, just that there would be a lot of engineering work to get an efficient NTT and modular arithmetic implementation at a higher bit value.

7.2 Benchmark at different bootstrapping parameter (b value)

Boolean circuits (Appendix B) present a special challenge since output wires of two gates is often the input wires of another in the next layer, effectively doubling the degree count for one multiplication operation! Fortunately, a boolean circuit usually operates in hundreds of bits simultaneously (e.g the output wire of a multiplication gate will be the input wire for tens if not hundreds of gates of the next layer), so this doubling effect gets flatten out across gates by a large extent.

Table 4 shows the performance of our protocol when running a circuit with $p_d = 2^{20}$ multiplication operations at different bootstrapping parameter. When $b = 2^5$, the total depth of our test circuit is 3 (Log b) $\times$ 2^{15} (s) $\approx$ 49.15 thouand.

Table 4: Comparison at different 'b' value ($n = 2^{20}$)

b	Prover	Verifier	Communication Cost
2^9	0.19 s	4.6 ms	26 KB
2^8	0.18 s	4.7 ms	43 KB
2^7	0.17 s	5.0 ms	79 KB
2^6	0.19 s	5.8 ms	147 KB
2^5	0.25 s	6.8 ms	283 KB

The prover runtime performance peaks at around 0.17 seconds when $b = 2^7$, then it starts to increase again afterwards. This is because the cost for computing the polynomial commitments for $\vec{r}, \vec{\epsilon}$ gets increasingly expensive as the number of sub-circuits (s) increases relative to the shrinking cost of field operations computing $\vec{r}, \vec{\epsilon}, \vec{\tau}$ in NTT.

Note that for a boolean circuit with at least 64-bits (2^6) output, p_d will get flatten out across 64 outputs so it is probably unlikely we need to set b lower than 2^6.

We are still able to achieve significant improvement in speed over the state-of-the-art at very high depth because 1) the PCS we are evaluating is still many times smaller than the size of the circuit polynomial. 2) All other costs (mainly NTT) besides PCS scheme is approaching 0. We may also achieve even better performance by switching to faster PCS schemes since the communication cost will be dominated "breakers" at lower "b" value anyway.

8 The Post-Quantum Option

We can transform our protocol to a plausible post-quantum design in two steps. First, switch from a group-based PCS scheme to a plausible post-quantum PCS scheme. Second, replace the input-mapping phase from Pedersen commitment to a post-quantum commitment/encryption scheme and design a post-quantum input-mapping verification protocol to confirm the transformation (Appendix D offers one such design for using post-quantum PCS).

We suspect the post-quantum version of our protocol offers even better prover and verifier runtime performances compared to that of the current version of our protocol. The communication cost, however, will likely be higher in the post-quantum setting in the default setting but flattens out when we move to a smaller "b" value for a more aggressive bootstrapping approach.

9 Final Remarks and SHA256 benchmark

Please see Appendix B for our performance benchmark in the memory efficient setting. Another notable benefit our approach offers is we can achieve quasi-scalability relative to the witness size, which is achieved in the boolean input test shown in Appendix C if the witnesses count is smaller than q, and fully achieved in Appendix D. Many have asked us to provide a benchmark proving the pre-image of SHA-256, you can find that analysis in Appendix F.

10 Estimated Benchmark for SHA256

We have been asked to provide a benchmark for proving SHA256 because our benchmark looks too good to be true. Full implementation will take time, so we give our best effort to make a conservative effort (for now).

We first accept that it is true that many techniques invented by our protocol have little value for shallow circuits. This is because we cannot use the placement of breakers to aggressively shrink the size of PCS relative to the circuit size. Unfortunately, a shallow circuit is exactly what we get after removing all full adders from the SHA256 circuit, where we will use a customized circuit to implement as follows:

1. Recompose 32 bit-representing polynomials $b_1, ..., b_{32}$ into one polynomial representing one 32-bit integer: $a' = \sum_{i=1}^{3} 2b_i \cdot 2^{i-1}$
2. Perform integer adds on $a'_1 + a'_2, ..., a'_n = c_\prime$
3. Decompose the sums computed from step 2 with 32 bit-representing polynomials. The prover has to send these new polynomials to the verifier.

There are several places to put breakers. In our experiment we place a breaker whenever a 32-bit modular sum is performed on inputs. This setup will make both of our estimation and implementation easier but likely not the most efficient way to construct the circuit.

The obvious downside of using such recomposition-modular add-decompositon strategy is it will significantly increase the communication cost. Regardless, we should still be able to prove and verify an SHA256 hash with high performance because:

1. There are 512 bits in each block, which spreads out the growth of the p_d value, which also limits the PCS size
2. In the scheduling phase of SHA-256, each block is spread out to 64×32 bits to further lower the p_d value relative to the total number of gates
3. The output of SHA256 is 256 bit-representing polynomials regardless of how much larger the intermediate state gets, creating a lot of merges on the coefficients s.t. further reduces the PCS size

For our unoptimized setup, we estimated the prover runtime cost is approximately 3 milliseconds, the verifier runtime cost is approximately 0.72 milliseconds, and the communication cost is 46.5 kilobytes.

Message Scheduling Rounds

The message scheduling phase computes message blocks $W_{16}, ..., W_{64}$. For each message block W_t, we compute $W'_t = W_{t-16} + \sigma_0(W_{t-15}) + W_{t-7} + \sigma_1(W_{t-2})$.

For the sigma functions σ_0, σ_1, we perform bit rotations and shifts on each transformed input in the same way as any SHA-256 implementation does on clear text bit values.

The sigma function on each bit is $W_{t_{ia}}$ XOR $W_{t_{ib}}$ XOR $W_{t_{ic}}$, where a, b, c represent shifted and rotated bits. There are 32 bits in each message block, so there should be a total of 64 XOR functions performed. However, since bit rotations will create many 0 bits, in practice only 61 XOR functions are needed for the σ_0 function and 54 XOR functions are needed for the σ_1 function.

$$XOR(A, B) = (A + B) - 2 * (A * B) \tag{76}$$

Each XOR gate can be implemented using two multiplication and two addition operations. Using linear polynomials as inputs for A and B requires seven multiplications and six additions to compute new coefficients and outputs a new bit with $p_d = 2$. Applying an XOR gate on the output of the previous XOR gate and a new input will require 10 multiplications and 8 additions to compute coefficients for the output of the second XOR gate, which is a new polynomial with $p_d = 3$.

At each message block, our circuit performs bit recomposition on $W_{t-16}, \sigma_0(W_{t-15}), W_{t-7}, \sigma_1(W_{t-2})$ and then adds them together using simple arithmetic in $\mathbb{Z}_q$. We use one breaker to reset p_d to 1 and then perform bit decomposition to break the updated message block W'_t into 32 linear polynomials.

We do not need to perform the "full" booleanity check on decomposed bits (linear polynomials) because their witnesses are used and committed in the later section of the protocol, so they are still statistically bounded by challenge x. Instead, we just need to check if the sum-of-product of decomposed linear polynomials can compute back to its 32-bit representation ($a' = \sum_{i=0}^{31} b'_i \cdot 2^i$)

In total, the message scheduling phase requires 11,232 multiplications, 11,232 additions/subtractions, 192 recompositions, 144 full adds, and 48 decompositions. For the prover to compute the coefficients for each output, that translates to at most 54,624 multiplication operations in $\mathbb{Z}_q$ and 46,272 addition operations in $\mathbb{Z}_q$ for our circuit. In practice, there will be fewer multiplication operations needed to compute coefficients for the output polynomial because there will be many 0 bits that will output 0 directly.

61-bit modular multiplication operations cost approximately 2 nanoseconds (already in Montgomery format) to 4 nanoseconds (including the cost of entering Montgomery format, which only needs to be performed once on the inputs) on our CPU. To make our estimate conservative, we assume 3 nanoseconds are required for all 61-bit modular multiplications and 1 nanosecond for addition/subtraction operations. In reality, each addition costs less than 1nm on most CPUs, we want to make our estimate conservative to cover other hidden costs.

Excluding PCS cost, the prover cost in the message scheduling rounds is approximately 0.3 milliseconds, and the verifier cost is approximately 0.11 milliseconds.

In terms of communication cost, the message scheduling rounds will add $48 \times 32 = 1536$ linear polynomial inputs (for bit decomposition) and 48 breakers for a total of ≈ 94.4 KBs

Compression Rounds

The compression function rounds updates message blocks a, b, c, d, e, f, g, h using scheduled message blocks $W_1, ..., W_{63}$, constant parameters $\vec{K}$, and messages blocks of the earlier round.

Like that for the message scheduling rounds, our protocol performs the sigma functions Σ_0, Σ_1 on linear polynomial inputs in the same way as any SHA-256 implementation does on clear text bits. There are a total of 128 XOR operations in Σ_0, Σ_1 executed in two layers in each round, which translate to 256 multiplications and 256 additions using the arithmetic equation 76.

$$Choice(x, y, z) = x * y + (1 - x) * z$$

$$Majority(a, b, c) = a * (b + c - 2bc) + bc$$

We implement each "Choice" function with 2 multiplication and 2 addition operations, which translates to 8 multiplications and 6 additions/subtractions to compute coefficients for the output of each "Choice" function that a polynomial degree $p_d = 2$. We also implement each "Majority" function with 3 multiplication and 3 addition operations, which translates to 13 multiplications and 8 additions/subtractions to compute coefficients for the output of each "Majority" function with $p_d = 3$.

At each round t, our circuit performs bit recomposition on outputs of $\Sigma_0, \Sigma_1, Choice, Majority$ functions and corresponding h_{t-1}, K_t, W_t so that the full adder operation to update e, a can be performed at negligible cost. We only need to apply breakers twice after computing e, a (or T_1, T_2) at the end of each round before decomposing each int32-representing linear polynomial into 32 single-bit representation linear polynomials.

Like that in the Message Scheduling Rounds, we do not need to perform the "full" booleanity check on decomposed bits (linear polynomials) because their witnesses, or coefficients computed from these witnesses, are committed in the later sections of the protocol, so they are still bounded by challenge x. Instead, we just need to check if the sum-of-product of decomposed linear polynomials can compute back to its 32-bit representation ($a' = \sum_{i=0}^{31} b'_i \cdot 2^i$).

The final output of the hash function is 8 linear polynomials $h'_0, ...h'_7$, and we use 8 full adds to compute them before decomposing each with 32 linear polynomials. We only need to perform booleanity check on the final output bit-representing polynomials using the protocol defined in Appendix C.

In total, the compression phase requires 26,624 multiplication and 26,624 addition/subtraction operations, 512 recompositions, 456 full-adds, and 128 decompositions. For our protocol, that translates to at most 133,376 multiplication operations in $\mathbb{Z}_q$ and 108,720 addition/subtraction operations in $\mathbb{Z}_q$ for our circuit to compute the coefficients for the 256 linear polynomials for the final output. Again, there will be fewer multiplication operations in practice because there will be many 0 bits in the circuit that will output 0 directly.

By conservatively assuming 3 nanoseconds are required for all 61-bit modular multiplications and 1 nanosecond for addition/subtraction (both are conservative estimates), the compression rounds will need approximately 0.5 milliseconds for the prover cost and 0.18 milliseconds for the verifier cost, excluding PCS cost (for applying and verifying breakers).

In terms of communication cost, the message compression rounds will add $128 \times 32 = 4096$ linear polynomial inputs (for bit decomposition) and 128 breakers for a total of ≈ 251.6 kilobytes.

Polynomial Commitment Scheme Evaluation

We have so far ignored the cost for PCS evaluation in both message scheduling rounds and compression rounds because we can evaluate them together. We only need to apply breakers at int32 representation linear polynomials s.t. there are a total of 176 linear polynomials with $p_d \leq 3$ s.t. each is computed with a sub-circuit. In our protocol, 3 PCSs (for commitments $\vec{R}_s, \vec{E}_s, \vec{C}$) are needed for each sub-circuit. In practice, we combine them into one PCS with size equal to $176 + 176 + 3 = 355$. This is significantly less than the number of constraints required by an R1CS circuit.

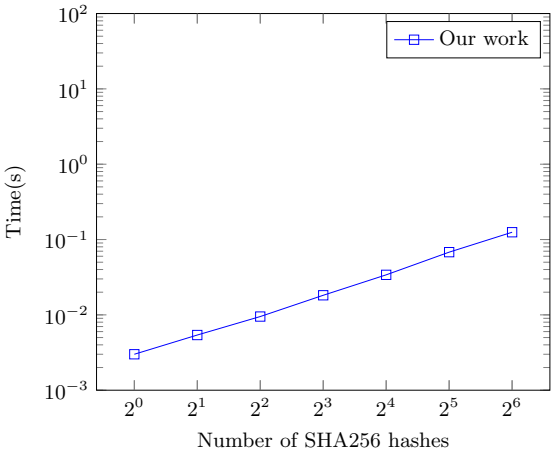
In the PCS scheme we are using (based on Bootle et. al with slight modification to adjust to 61-bit inputs), the prover cost (committing plus evaluation) is approximately 2.2 milliseconds, the verifier cost is 0.46 milliseconds, and the proof size is approximately 1 KB.

The Final Estimate

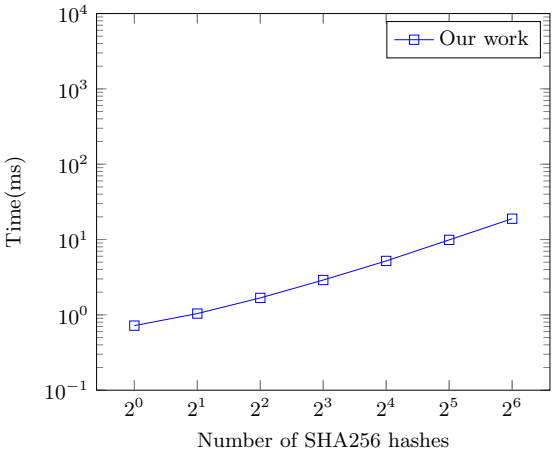
The output of the SHA-256 function is 256 bits. In our protocol that translates to 256 bit-representing linear polynomials. To make these bits "committed" and bound by challenge "x", we run the proof of booleanity protocol as specified in Appendix C.

Based on the detailed estimate we listed above, our conservative estimate the total cost of proving the knowledge of witness of a SHA-256 hash is approximately 3 milliseconds for the prover time, 0.72 milliseconds for the verifier time, and 46 KB for the communication cost.

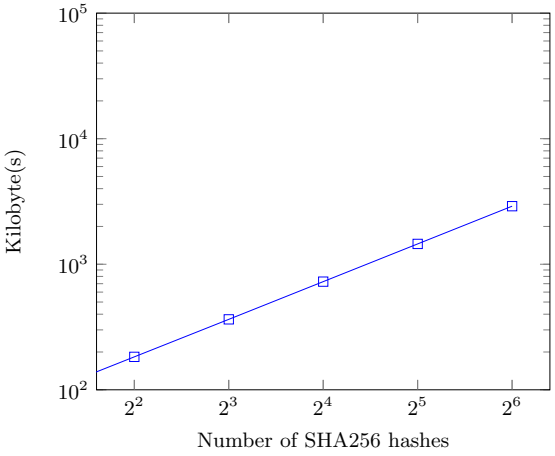
The simple way to estimate cost for proving n number of hashes is to add them together. However in practice there are at least two optimizations we can do: first, we only need to run the proof for booleanity once for the final output in a merkle tree proof because output linear polynomials of each child node is automatically inputs to the next layer of the tree. second, we can combine all PCSs into one big PCS for all n SHA256 proofs.



(a) Prover Runtime



(b) Verifier Runtime



(c) Communication Cost

The cost of proving one SHA256 hash is approximately only 3 milliseconds, and proving 64 consecutive hashes (equivalent to proving a Merkle tree with 32 leafs) can be completed in just 0.12 second, more than 50X improvement over the state-of-the-art transparent NIZK protocols and comparable to

the top of the line VOLE protocols (estimate based on the actual number of arithmetic operations in $\mathbb{Z}_q$ is called and the size of the PCS needs to be evaluated). [42]

The cost of verifying one SHA256 hash is approximately 0.72 millisecond, and verifying 64 consecutive hashes is around 18.9 milliseconds, still ahead of the state-of-the-art to the best of our knowledge. However, as the cost of direct computation (linear time, about 0.26 milliseconds per hash) starts to dominate the verifier cost, our protocol will get increasingly less efficient in terms of verifier cost. However, the prover runtime cost of alternative protocols will also grow too high till that point.

The simple construction we used for cost estimation (possible implementation later) is not friendly for managing communication costs. The communication cost for a single SHA256 circuit is already at 46.5KBs, and grows to 2.8MBs for 64 hashes.

The high communication cost is mainly caused by us aggressively applying breakers after every arithmetic sum, and each breaker then translates to 33 decomposed linear polynomials (1 for breaker, 32 for decomposed bits). This contributes to 45KBs out of the 46.5KBs of communication cost required for each hash function.

However, this is not to be confused with designated verifier VOLE protocols where there is linear a per-gate communication cost, the growth shown in the graph is actually aligns to the per-bit-decomposition action. Had we more carefully designed our protocol and use breakers less aggressively, it is highly likely we can get a much better benchmark on communication cost with some additional expense on prover and verifier runtime costs depending on the circuit design.

11 The Post-Quantum Option

It is possible to transform our protocol to a plausible post-quantum design in two steps. First, we can switch from a group-based PCS scheme to a plausible post-quantum PCS scheme. Second, replace the input-mapping phase from Pedersen commitment to a post-quantum commitment/encryption scheme and design a post-quantum input-mapping verification protocol to confirm the transformation (Appendix D offers one such design for using post-quantum PCS).

We suspect the post-quantum version of our protocol (e.g., using Reed-Soloman code based PCS scheme) offers even better prover and verifier runtime performances compared to that of the current version of our protocol. The communication cost, however, will likely be higher in the post-quantum setting in the default setting but flattens out when we move to a smaller "b" value for a more aggressive bootstrapping approach.

12 Final Remarks

Please see Appendix B for our performance benchmark in the memory efficient setting. Another notable benefit our approach offers is we can achieve quasi-scalability relative to the witness size, which is achieved in the boolean input test shown in Appendix C if the witnesses count is smaller than q, and fully achieved in Appendix D.

References

1. Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: Lightweight sublinear arguments without a trusted setup. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 2087–2104. ACM Press, Dallas, TX, USA (Oct 31 – Nov 2, 2017). https://doi.org/10.1145/3133956.3134104
2. Aranha, D.F., Bennedsen, E.M., Campanelli, M., Ganesh, C., Orlandi, C., Takahashi, A.: ECLIPSE: Enhanced compiling method for pedersen-committed zkSNARK engines. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds.) PKC 2022, Part I. LNCS, vol. 13177, pp. 584–614. Springer, Cham, Switzerland, Virtual Event (Mar 8–11, 2022). https://doi.org/10.1007/978-3-030-97121-2$_2$1

3. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and hardness of approximation problems. In: 33rd FOCS. pp. 14–23. IEEE Computer Society Press, Pittsburgh, PA, USA (Oct 24–27, 1992). https://doi.org/10.1109/SFCS.1992.267823
4. Arora, S., Safra, S.: Probabilistic checking of proofs; A new characterization of NP. In: 33rd FOCS. pp. 2–13. IEEE Computer Society Press, Pittsburgh, PA, USA (Oct 24–27, 1992). https://doi.org/10.1109/SFCS.1992.267824
5. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: 23rd ACM STOC. pp. 21–31. ACM Press, New Orleans, LA, USA (May 6–8, 1991). https://doi.org/10.1145/103418.103428
6. Babai, L., Fortnow, L., Lund, C.: Non-deterministic exponential time has two-prover interactive protocols. In: 31st FOCS. pp. 16–25. IEEE Computer Society Press, St. Louis, MO, USA (Oct 22–24, 1990). https://doi.org/10.1109/FSCS.1990.89520
7. Baum, C., Braun, L., Delpech de Saint Guilhem, C., Klooß, M., Orsini, E., Roy, L., Scholl, P.: Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part V. LNCS, vol. 14085, pp. 581–615. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 20–24, 2023). https://doi.org/10.1007/978-3-031-38554-4$_1$9
8. Baum, C., Malozemoff, A.J., Rosen, M.B., Scholl, P.: Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 92–122. Springer, Cham, Switzerland, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84259-8$_4$
9. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 701–732. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_2$3
10. Bhadauria, R., Fang, Z., Hazay, C., Venkitasubramaniam, M., Xie, T., Zhang, Y.: Ligero++: A new optimized sublinear IOP. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2025–2038. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3417893
11. Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 327–357. Springer, Berlin, Heidelberg, Germany, Vienna, Austria (May 8–12, 2016). https://doi.org/10.1007/978-3-662-49896-5$_1$2
12. Bootle, J., Chiesa, A., Hu, Y., Orrù, M.: Gemini: Elastic SNARKs for diverse environments. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 427–457. Springer, Cham, Switzerland, Trondheim, Norway (May 30 – Jun 3, 2022). https://doi.org/10.1007/978-3-031-07085-3$_1$5
13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243868
14. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354255
15. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_1$6
16. Campanelli, M., Faonio, A., Fiore, D., Querol, A., Rodríguez, H.: Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part III. LNCS, vol. 13092, pp. 3–33. Springer, Cham, Switzerland, Singapore (Dec 6–10, 2021). https://doi.org/10.1007/978-3-030-92078-4$_1$
17. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part II. LNCS, vol. 14005, pp. 499–530. Springer, Cham, Switzerland, Lyon, France (Apr 23–27, 2023). https://doi.org/10.1007/978-3-031-30617-4$_1$7
18. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.P.: Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I.

LNCS, vol. 12105, pp. 738–768. Springer, Cham, Switzerland, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-1$_2$6
19. Choudhuri, A.R., Garg, S., Goel, A., Sekar, S., Sinha, R.: SublonK: Sublinear prover PlonK. Cryptology ePrint Archive, Report 2023/902 (2023), https://eprint.iacr.org/2023/902
20. Cramer, R., Damgård, I.: Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In: Krawczyk, H. (ed.) CRYPTO'98. LNCS, vol. 1462, pp. 424–441. Springer, Berlin, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 23–27, 1998). https://doi.org/10.1007/BFb0055745
21. Diamond, B.E., Posen, J.: Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Report 2024/504 (2024), https://eprint.iacr.org/2024/504
22. Dittmer, S., Ishai, Y., Lu, S., Ostrovsky, R.: Improving line-point zero knowledge: Two multiplications for the price of one. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 829–841. ACM Press, Los Angeles, CA, USA (Nov 7–11, 2022). https://doi.org/10.1145/3548606.3559385
23. Dittmer, S., Ishai, Y., Ostrovsky, R.: Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446 (2020), https://eprint.iacr.org/2020/1446
24. Frederiksen, T.K., Nielsen, J.B., Orlandi, C.: Privacy-free garbled circuits with applications to efficient zero-knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 191–219. Springer, Berlin, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46803-6$_7$
25. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2019), https://eprint.iacr.org/2019/953
26. Giacomelli, I., Madsen, J., Orlandi, C.: ZKBoo: Faster zero-knowledge for Boolean circuits. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 1069–1083. USENIX Association, Austin, TX, USA (Aug 10–12, 2016)
27. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: 17th ACM STOC. pp. 291–304. ACM Press, Providence, RI, USA (May 6–8, 1985). https://doi.org/10.1145/22145.22178
28. Golovnev, A., Lee, J., Setty, S.T.V., Thaler, J., Wahby, R.S.: Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part II. LNCS, vol. 14082, pp. 193–226. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 20–24, 2023). https://doi.org/10.1007/978-3-031-38545-2$_7$
29. Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-SNARKs. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 698–728. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 19–23, 2018). https://doi.org/10.1007/978-3-319-96878-0$_2$4
30. Heath, D., Kolesnikov, V.: Stacked garbling for disjunctive zero-knowledge proofs. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 569–598. Springer, Cham, Switzerland, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45727-3$_1$9
31. Kiayias, A., Tang, Q.: How to keep a secret: leakage deterring public-key cryptosystems. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 943–954. ACM Press, Berlin, Germany (Nov 4–8, 2013). https://doi.org/10.1145/2508859.2516691
32. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2111–2128. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3339817
33. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 239–252. Springer, New York, USA, Santa Barbara, CA, USA (Aug 20–24, 1990). https://doi.org/10.1007/0-387-34805-0$_2$2
34. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-OLE: Improved constructions and implementation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1055–1072. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3363228
35. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 704–737. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 17–21, 2020). https://doi.org/10.1007/978-3-030-56877-1$_2$5
36. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), https://eprint.iacr.org/2020/1275

37. Wahby, R.S., Tzialla, I., shelat, a., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. Cryptology ePrint Archive, Report 2017/1132 (2017), https://eprint.iacr.org/2017/1132
38. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 2021 IEEE Symposium on Security and Privacy. pp. 1074–1091. IEEE Computer Society Press, San Francisco, CA, USA (May 24–27, 2021). https://doi.org/10.1109/SP40001.2021.00056
39. Weng, C., Yang, K., Yang, Z., Xie, X., Wang, X.: AntMan: Interactive zero-knowledge proofs with sublinear communication. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 2901–2914. ACM Press, Los Angeles, CA, USA (Nov 7–11, 2022). https://doi.org/10.1145/3548606.3560667
40. Yang, K., Sarkar, P., Weng, C., Wang, X.: QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 2986–3001. ACM Press, Virtual Event, Republic of Korea (Nov 15–19, 2021). https://doi.org/10.1145/3460120.3484556
41. Yang, K., Weng, C., Lan, X., Zhang, J., Wang, X.: Ferret: Fast extension for correlated OT with small communication. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1607–1626. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3417276
42. Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: 2020 IEEE Symposium on Security and Privacy. pp. 859–876. IEEE Computer Society Press, San Francisco, CA, USA (May 18–21, 2020). https://doi.org/10.1109/SP40000.2020.00052

APPENDIX

A. Univariate Polynomial Commitment Scheme

When a lot of inputs have value 0, it is possible that some z_i (except z_0, the sum of product of blinding keys) also computes value "0", which may leak information about inputs.

The reason we didn't pick the more popular FRI-based PCS scheme is because the polynomial our protocol needs to evaluate is expected to be small, and the prover and verifier runtime performance is automatically good. Instead, we use the PCS defined by Bootle et. al. which is depicted below with our terminology:

$$f(X) = (1\ X^n ...\ X^{(m-1)n}) \begin{pmatrix} z_{0,0} & .. & z_{0,n-1} \\ z_{1,0} & .. & z_{1,n-1} \\ . & . & . \\ . & . & . \\ z_{m-1,0} & .. & z_{m-1,n-1} \end{pmatrix} \begin{pmatrix} 1 \\ X \\ . \\ . \\ X^{n-1} \end{pmatrix}$$

Figure 4

$\vec{z}$ are coefficients of the polynomial and are arranged to an $m \times n$ matrix. The idea behind Bootle et al's protocol is that the prover commits to the rows of this matrix using commitments $C_0, ..., C_{m-1}$ s.t. $C_i = \sum_{j=0}^{n-1} z_{i,j} \cdot x^j$. When given an evaluation point x, we use the homomorphic property of the commitment scheme to compute the commitment $\prod_{i=0}^{m-1} C_i^{i \cdot n}$ to the vector:

$$\vec{c} = (1\ x^n ...\ x^{(m-1)n}) \begin{pmatrix} z_{0,0} & . & . & z_{0,n-1} \\ z_{1,0} & . & . & z_{1,n-1} \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ z_{m-1,0} & . & . & z_{m-1,n-1} \end{pmatrix}$$

Figure 5

The prover opens this later commitment so that computing $y = f(x)$ becomes trivial. To avoid leaking partial information about the coefficients of $f(x)$ through these openings, Bootle et al.'s protocol inserts blinding values $u_1, ..., u_{n-1}$ to hide the weighted sum of the coefficients in each column of the matrix, and make sure these blinding values cancel each other out when the polynomial gets evaluated at point x.

$$\begin{pmatrix} z_{0,0} & z_{0,1} + u_1 & . & . & z_{0,n-2} + u_{n-2} & z_{0,n-1} + u_{n-1} \\ z_{1,0} & z_{1,1} & . & . & z_{1,n-2} & z_{1,n-1} \\ . & . & & & . & . \\ . & . & & & . & . \\ . & . & & & . & . \\ z_{m-1,0} & z_{n-1,1} & . & . & z_{m-1,n-2} & z_{m-1,n-1} \\ -u_{q\,1} & -u_{q\,2} & . & . & -u_{q\,n-1} & 0 \end{pmatrix}$$

Figure 6

We do not need to make any modifications to the protocol itself. However, we need to redefine the domain of each variable in the matrix above since all coefficients are in $\mathbb{Z}_q$ and cannot overflow p when computing commitments $C_0, ..., C_{m-1}$. When given an evaluation point x, we use the homomorphic property of the commitment scheme to compute the commitment $\prod_{i=0}^{m-1} C_i^{i \cdot n}$.

Since coefficients $\vec{z}$ are in $\mathbb{Z}_q$ and the exponentiation operations performed in group cannot overflow the order of $\mathbb{G}$, the verifier must compute powers of x (e.g. $x, x^2, ...x^s$) in $\mathbb{Z}_q$ before applying them to C_i.

Each blinding key u_j is randomly sampled from $\mathbb{Z}_p$ and $u_{qj} = u_j \bmod q$. We set each $|u_j|$ to a $|\sum_{i=0}^{m-1} z_{i,j} \cdot x^{j \cdot n}| + 80$-bits value, which is approximately ≈ 207-bits in our case for $n = m = 2^5$, enough to hide coefficients of T_i. For the first column $j = 0$, we expand z_0 from a 61-bit value to an equivalent 207-bit value s.t. $z_0 = z_0 + s \cdot q$ for some 146-bit random s.

$u_j \in \mathbb{Z}_p$ will get cancelled by $-u_{qj} \in \mathbb{Z}_q$ once the domain of the final evaluation is reduced to domain $\mathbb{Z}_q$.

B. Benchmark In The Memory Efficient Setting

One of the biggest advantages of our protocol is that we can build a memory-efficient, non-interactive zero-knowledge version out of it with a theoretical memory cost of $O(b+s)$. Using the memory-efficient implementation mentioned in Section 6.4, we get the results listed in Table 5.

Table 5: Performance comparison in memory efficient setting ($n = 2^{20}$)

b	Prover	Verifier	Communication Cost
2^{10}	0.22 s	4.5 ms	16 KB
2^9	0.19 s	4.6 ms	26 KB
2^8	0.18 s	4.7 ms	43 KB
2^7	0.17 s	5.0 ms	79 KB
2^6	0.19 s	5.8 ms	147 KB
2^5	0.25 s	6.8 ms	283 KB
2^4	0.34 s	7.7 ms	547 KB
2^3	0.58 s	9.9 ms	1075 KB

The memory efficient mode peaks at around 3 million multiplications per second when $b = 2^6$. Since the number of NTT operations is doubles, the performance is noticeably worse than that shown in Table 4 running in the non-memory efficient mode.

Regardless, the benchmark numbers shown in Table 4 compare well against top-of-the line VOLE-based protocols (shown in Table 5; reported numbers are copied directly from paper [40], so CPU differences are not counted for.), especially consider our protocol we benchmarked is non-interactive (not impacted by network latency), publicly verifiable, and offers a significantly smaller proof size without requiring pre-setup like that of Ant-Man.

Table 6: Performance of VOLE protocols (Arithmetic Circuit)

Protocol	Size	Speed	Non-Interactive
Wolverine	4	0.66 M	No
Mac'n'Cheese	3	0.4 M	No
QuickSilver	1	4.8 M	No
This work	$\frac{1}{b}$	$\geq$ 3 M	Yes

There are other techniques for improving prover memory: Commit-and-prove to glue sub-circuits together (Lunar/Eclipse [16] [2]), streaming SNARKs (Gemini [12]). However, the reported constructions of these protocols require trusted setup (non-transparent), and the prover runtime cost of these protocols is magnitudes more expensive.

C. The sub-protocol for boolean circuit validation

A common requirement in proving boolean circuits is to enforce input data $b_i \in \{0, 1\}$ for $i \in \{1, ..., l\}$ (this is not new, but we need this sub-protocol defined to make our main protocol easier to parse), such relation is defined as:

$$\{(\vec{b'}, \in \mathbb{Z}_q^l \,;\, \vec{b}, \vec{\beta} \in \mathbb{Z}_q^l) : b'_i = b_i + X\beta_i \ \wedge \ \forall_{b_i} \in [0, 1] \ \wedge \ \forall_i \in [1, .., l]\} \tag{77}$$

In practice, it is useful to decompose l full integer inputs into $l \cdot 32$ bits (assuming we use 32 bits to represent a full integer). If a committed value b_i is in $[0, 1]$, then its linear polynomial form b_i must have the following property:

$$d_i = (b'_i \cdot b'_i - b'_i) = \delta_{1,i}x + \delta_{2,i}x^2 \tag{78}$$

Where $\delta_{2,i} = \beta_i^2$, and $\delta_{1,i} = \beta_i$ when b_i is 1 and $\delta_{1,i} = -\beta$ when b_i is 0. If $b_i \in [0, 1]$, then the constant term can only be 0.

We can reduce the degree of d_i by dividing it by x to get a degree 1 linear polynomial:

$$d'_i = (b'_i \cdot b'_i - b'_i)/x = \delta_{1,i} + \delta_{2,i}x \in \mathbb{Z}_q \quad for \quad i = \{1, ..., n\}$$

We take $\vec{d'}$ as inputs to a sub-circuit that simply multiplies them all sequentially:

$$o_i = \prod_{i=1}^{n} d'_i \in \mathbb{Z}_q$$

If the constant term is 0 for all d_i and $d_i \neq 0$, then we can compute o_i using committed coefficients s.t. the degree of each term is a positive value. Otherwise at least one of the terms has a negative power of x^{-m} for some m. This allows us to reuse most of the protocol AriCircuit to perform the booleanity test on linear polynomial inputs. There are two differences:

- First, there is no need to test the mapping of linear polynomial inputs and output representing boolean values to commitments
- Second, instead of evaluating just one single output for the whole protocol, the updated protocol evaluates n output polynomials at once, and each output o_i represents the evaluation of l boolean inputs representing one l-bit input

We define the protocol BooleanityTest by modifying the main protocol AriCircuit as follows:

$$Input : (\vec{b}, \vec{\beta} \in \mathbb{Z}_p) \tag{79}$$
$$\mathcal{P}\ compute : \tag{80}$$
$$\delta_{1i} = b_i\beta_i + b_i\beta_i - \beta_i \in \mathbb{Z}_q \qquad i = \{1, ..., l \times n\} \tag{81}$$
$$\delta_{2i} = \beta_i^2 \in \mathbb{Z}_q \qquad i = \{1, ..., l \times n\} \tag{82}$$
$$r_\beta, \epsilon_\beta, c_\beta \xleftarrow{\$} \mathbb{Z}_q \tag{83}$$
$$\textbf{for} \quad i = 1, ..., n \quad \{ \tag{84}$$
$$r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{\delta_1}', \vec{\delta_2}'); \tag{85}$$
$$\epsilon_i = (\epsilon_i - \tau_{i,0}) \bmod q \quad \} \tag{86}$$
$$\text{polyCommit}(r_\beta || \vec{r}) \rightarrow: \vec{R_s};\ \text{polyCommit}(\epsilon_\beta || \vec{\epsilon}) \rightarrow: \vec{E_s} \tag{87}$$
$$\mathcal{P} \rightarrow \mathcal{V} : \vec{R_s}, \vec{E_s} \tag{88}$$
$$\mathcal{V} \rightarrow \mathcal{P} : \quad w \xleftarrow{\$} \mathbb{Z}_p \tag{89}$$
$$\mathcal{P}\ compute : \tag{90}$$
$$c_j = \sum_{i=1}^{n} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{0, ..., l-2\} \tag{91}$$
$$\text{polyCommit}(\vec{c}) \rightarrow: \vec{C} \tag{92}$$
$$\mathcal{P} \rightarrow \mathcal{V} : \vec{C}, \tag{93}$$
$$\mathcal{V} \rightarrow \mathcal{P} : \quad x \xleftarrow{\$} \mathbb{Z}_p \tag{94}$$
$$\mathcal{P}\ compute : \tag{95}$$
$$b'_i = b_i + x \cdot \beta_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{96}$$
$$r'_i = r_i + x \cdot \epsilon_i \in \mathbb{Z}_q \qquad i = \{\beta, 1, ..., n\} \tag{97}$$
$$y_c = \sum_{j=0}^{b} c_j \cdot x^j \in \mathbb{Z}_q \tag{98}$$
$$\mathcal{P} \rightarrow \mathcal{V} : \vec{b}', \vec{r}', y_c \tag{99}$$
$$\mathcal{V} \rightarrow \mathcal{P} : \quad z \xleftarrow{\$} \mathbb{Z}_q \tag{100}$$
$$\mathcal{P}\ compute : \tag{101}$$
$$y_r = r_\beta + \sum_{i=1}^{s} r_i \cdot z^i \in \mathbb{Z}_q; \qquad y_\epsilon = \epsilon_\beta + \sum_{i=1}^{s} \epsilon_i \cdot z^i \in \mathbb{Z}_q \tag{102}$$
$$\mathcal{P} \rightarrow \mathcal{V} : y_r, y_\epsilon \tag{103}$$
$$\mathcal{V}\ verify\ final\ output : \tag{104}$$
$$\textbf{for} \quad i = 1, ..., n \quad \{ \tag{105}$$
$$o_i = computeSubCircuit_i(\vec{b}') \in \mathbb{Z}_q \tag{106}$$

$$y_i = o_i - r'_i \in \mathbb{Z}_q \quad \} \tag{107}$$

$$\textbf{if } (\text{polyEval}(\vec{R_s}, y_r, z; \vec{r}) \ \wedge \ \text{polyEval}(\vec{E_s}, y_\epsilon, z; \vec{\epsilon}) \tag{108}$$

$$\wedge \ 0 = (y_r + y_\epsilon \cdot x - (r'_\beta + \sum_{i=1}^{s} r'_i)) \bmod q \tag{109}$$

$$\wedge \ \text{polyEval}(\vec{C}, y_c, x; \vec{c}) \tag{110}$$

$$\wedge \ y_c = (\sum_{i=1}^{s} y_i \cdot w^i) \bmod q) \ \textbf{then } \textit{continue} \tag{111}$$

$$\textbf{else } \textit{reject} \tag{112}$$

$$\tag{113}$$

Protocol BooleanityTest

Once we know all linear polynomials map to either 0 or 1, it is trivial to recompose the linear polynomial form of a full integer input a'_i from 32 decomposed bits $b'_{i,j}$ for $j = \{1, ..., 32\}$.

$$a'_i = \sum_{j=1}^{32} b'_{i,j} \cdot 2^j \tag{114}$$

D. The Scalable Linear Polynomial to Commitment Mapping Validation Sub-protocol

In Appendix C we demonstrates how to use PCS scheme to batch validate a list of boolean witnesses committed using one Pedersen commitment. This implies the verification time of boolean witnesses increases sub-linearly in group exponentiation operations as long as long as the PCS scheme has sub-linear verification time, which is usually the case.

However, if the witnesses size is bigger than $|q|$, we need to use multiple input commitments, which requires linear verification time if we use the protocol we defined in Section 5. Fortunately, we can tweak our input-mapping validation algorithm to allow group exponentiation operation achieve sub-linear verification time for witness of any size. We call it quasi-salable because the time complexity for cheap operations in $\mathbb{F}$ is still linear.

We define an updated version of the sub-protocol we defined in Section 5 so that the verification cost for group exponentiations scales sub-linearly as the witness(es) size increases. To achieve verification time scalability, we can no longer use Pedersen commitment to commit each input independently. Instead, we use polynomial commitment to batch commit input witnesses. The relation 115 is updated to the following:

$$\begin{aligned} \{(\vec{A}, \vec{V} \in \mathbb{G}, \vec{a}' \in \mathbb{Z}_q \, ; \, \vec{a}, \vec{\alpha} \in \mathbb{Z}_q, \vec{v} \in \mathbb{Z}_p) \\ : \vec{A} = \text{polyCommit}(\vec{a}) \\ \wedge \vec{V} = \text{polyCommit}(\vec{v}) \wedge a'_i = a_i + X\alpha_i\} \end{aligned} \tag{115}$$

$\vec{A}$ is the polynomial commitment of inputs $\vec{a}$ and $\vec{V}$ is the polynomial commitment of inputs' blinding keys $\vec{v}$. However, not using Pedersen commitments to store each witness independently means we can no longer achieve the reusability of circuit inputs and outputs.

Blinding Input

Since elements in $\vec{a}$ are in $\mathbb{Z}_q$, evaluating PCS in $\mathbb{Z}_p$ may leak information about $\vec{a}$. To guard against that, we create a blinding input a'_0 defined as follows:

$$a'_0 = a_0 + x \cdot \alpha_0 \in \mathbb{Z}_p \tag{116}$$

Where as a_0 and α_0 are random values in $\mathbb{Z}_p$, the prover will also create a matching key υ_0 that it will commit with the rest of the blinding keys $\vec{v}$.

Setup Phase

Before the random challenge x is available, the prover make two commitments for each i-th input.

The first polynomial commitment commits randomly generated blinding keys $\omega_i \in \mathbb{Z}_p$ for $i = \{0, ..., l\}$ used in transformation validation.

$$\text{polyCommit}(\vec{\omega}) \rightarrow: \vec{S} \tag{117}$$

The second polynomial commitment commits to the new blinding key $\alpha_i \in \mathbb{Z}_q$ used to hide data in transformed input $a'_i \in \mathbb{Z}_q$.

$$\text{polyCommit}(\vec{v'}) \rightarrow: \vec{T} \tag{118}$$

The setup phase of the protocol is detailed below. This part is called before the random challenge x is generated.

$Input : (; \vec{\omega}, \vec{v} \in \mathbb{Z}_q, \vec{v} \in \mathbb{Z}_p)$
$\mathcal{P}'s\ input : (; \vec{\omega}, \vec{v})$;
$\mathcal{P}\ compute :$
polyCommit$(\vec{\omega}) \rightarrow: \vec{S}$
polyCommit$(\vec{v}) \rightarrow: \vec{T}$
$\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}$

Protocol Scalable InputMapping - Setup

Once the setup phase completes, the prover then sends $\vec{S}, \vec{T}$ to the verifier.

Validation Phase

After the random challenge x is generated, the prover computes $\vec{a}'$ and sends them to the verifier. Next, the protocol checks the mapping between transformed inputs in $\mathbb{Z}_q$ to those in group $\mathbb{G}$.

For each a'_i, the prover provides transcript e_i, which is used to convert the blinding element $x\alpha \in \mathbb{Z}_q$ of each a'_i to its raw form $x\alpha \in \mathbb{F}$ (without mod q).

$$e_0 = \omega_0 \cdot q \in \mathbb{F} \tag{119}$$

$$e_i = ((x\alpha_i \bmod q) - x\alpha_i) \cdot x + \omega_i \cdot q \in \mathbb{F}, \qquad i = \{1, ..., l\} \tag{120}$$

The part of e_i on the left of the addition sign $(((x\ \alpha_i \bmod q) - x\ \alpha_i) \cdot x)$ is around $\approx$ 183-bits, which is small enough s.t. an adversary can use brutal force attack to extract α_i. We can rewrite this left part to $s_i\ \cdot q$ for some $|s_i| \leq$ 122-bits and equation 7 to $e_i = (s_i + \omega_i) \cdot q$. The prover can use a 183-bit random value ω_i (80-bits larger than s_i) to perfectly hide it except for a negligible probability of at most 2^{-80} as we explained in Section 5 already. To make this protocol simpler, we set $\omega_i \in \mathbb{Z}_p$. This makes $|e_i| \leq |\omega| + |q| \approx$ 313-bits.

With transcripts e_i, we can compute to the following equality with witnesses $a_i, \alpha_i, \upsilon_i, \omega_i$ for $i = \{0, ..., l\}$:

$$(a_i + x \cdot \upsilon_i)x = a'_i \cdot x - e_i + (\upsilon_i - \alpha)x^2 + \omega_i \cdot q \in \mathbb{Z}_p \tag{121}$$

Witnesses are committed using PCS scheme. With evaluation point k, the prover provides the verifiable evaluations of committed polynomials $y_a, y_\alpha, y_\upsilon, y_\omega$, using which we can update the equality 121 above to cover all input witnesses:

$$(y_a + x \cdot y_\upsilon)x = (\sum_{i=0}^{l} a'_i \cdot x - e_i) - (y_\upsilon - y_\alpha)x^2 - y_\omega \in \mathbb{Z}_p \tag{122}$$

To ensure soundness, note that only elements carrying input data (y_a and a'_i) are taking to the first power x, and all other committed values are either taking to the second power x^2 or not at all.

Finally, the verifier validates that e_i doesn't alter the value of a_i. This can be done by taking the modulus q of e_i which must return 0. This is trivial to understand since a'_i is in $\mathbb{Z}_q$ so e_i must be a multiple of q.

$$\textbf{if } (e_i \bmod q) \stackrel{?}{=} 0, \textbf{ then } \textit{continue} \tag{123}$$

We have so far skipped the overflow problem. If $a_i + (x\ \alpha_i \bmod q)\ > q$, then we will have an overflow problem in equation **??** 8 when computing $a'_i \cdot x - e_i$. To get around this, the prover simply needs to check if $a_i + (x\ \alpha_i)$ mod q overflows q and subtract $q \cdot x$ from e_i if that's the case.

$$\textbf{if } a_i + (x\alpha_i \bmod q) \geq q, \textbf{then } e_i = e_i - x \cdot q \qquad i = \{1, ..., l\} \tag{124}$$

This adjustment does not break the zero-knowledgeness of e_i for the same reason stated for the protocol in Section 5. The validation part of the input-mapping sub-protocol is defined as follows:

$Input : (\vec{A}, \vec{V}, \vec{S}, \vec{T} \in \mathbb{G},\ \vec{a}' \in \mathbb{Z}_q; \vec{a}, \vec{\alpha}, \vec{\omega} \in \mathbb{Z}_q, a_0, \alpha_0,\ \vec{v} \in \mathbb{Z}_p)$

$\mathcal{P}'s\ input : (\vec{A}, \vec{V},\ \vec{S}, \vec{T}; \vec{a}, \vec{\alpha}, \vec{\omega}, \vec{v});\ \mathcal{V}'s\ input : (\vec{A}, \vec{V}, \vec{S}, \vec{T})$

$\mathcal{P}$ *compute* :

$e_0 = \omega_0 \cdot q \in \mathbb{F}$

$e_i = ((x\ \alpha_i \bmod q) - x\ \alpha_i)x\ + \omega_i\ q; \qquad i = \{1, ..., l\}$

if $a_i + (x\ \alpha_i \bmod q)\ > q,$

then $e_i = e_i - q \cdot x \qquad i = \{0, ..., l\}$

$\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}'$

$\mathcal{V} \rightarrow \mathcal{P} : k \xleftarrow{\$} \mathbb{Z}_p$

$\mathcal{P}$ *compute* :

$$y_a = \sum_{i=0}^{l} a_i \cdot k^i \in \mathbb{Z}_p; \qquad y_\alpha = \sum_{i=0}^{l} \alpha_i \cdot k^i \in \mathbb{Z}_p$$

$$y_\omega = \sum_{i=0}^{l} \omega_i \cdot k^i \in \mathbb{Z}_p; \qquad y_\upsilon = \sum_{i=0}^{l} \upsilon_i k^i \in \mathbb{Z}_p$$

$\mathcal{P} \rightarrow \mathcal{V} : y_a, y_\alpha, y_\omega, y_\upsilon$

$\mathcal{V}$ *verify inputs* :

if $(e_i\ \bmod q) \stackrel{?}{=} 0,$ **then** *continue*; $\qquad i = \{0, ..., l\}$

else *reject*

if $(y_a + x \cdot y_\upsilon)x = (\sum_{i=0}^{l} a'_i \cdot x - e_i) - (y_\upsilon - y_\alpha)x^2 - y_\omega \in \mathbb{Z}_p$

$\wedge$ polyEval$(\vec{A}, y_a, k; \vec{a})\ \wedge$ polyEval$(\vec{V}, y_v, k; \vec{\alpha})$

$\wedge$ polyEval$(\vec{S}, y_s, k; \vec{\omega})\ \wedge$ polyEval$(\vec{T}, y_t, k; \vec{v})$

else *reject*

Protocol for Scalable InputMapping - Verify

Theorem 3. *(The Scalable Input-Mapping Protocol). The proof system presented in this section has perfect completeness, PHVZK, and CWEE.*

Proof. The perfect completeness of protocol InputMapping Validation is trivial to observe.

During the setup phase, $\vec{S}, \vec{T}$ are already set, so simulation starts after the setup phase. After receiving challenge x and matching linear polynomials $\vec{a}'$ from the verifier, it simulates all proof transcripts needed proving the mapping between committed inputs $\vec{A}, \vec{V}$, and whatever $\vec{a}'$ it received.

To begin, the simulator $\mathcal{S}_{input}$ randomly generates and sends $\vec{e}^*$ (e_i^* is generated by first randomly generating a value $v_i \in \mathbb{Z}_p$ and multiplying it by q s.t. $e_i^* = v_i \cdot q$ s.t. equality 119 will pass) to the verifier. The simulator $\mathcal{S}_{input}$ then randomly generates and computes $y_{a_1}^*, y_{\alpha_1}^*, y_{\omega_1}^*, y_{v_1}^*$ s.t. the equality 122 is true, and calls simulator $\mathcal{S}_P$ to simulate transcripts for the PCS evaluation.

We already know for a fact that there exists a simulator $\mathcal{S}_P$ that can rewind and simulate transcripts for the PCS scheme we are using, so $y_{a_1}^*, y_{\alpha_1}^*, y_{\omega_1}^*, y_{v_1}^*$ will pass the validation test to prove that they are the correct evaluation outputs for commitments $\vec{A}, \vec{V}, \vec{S}, \vec{T}$.

Given that simulators $\mathcal{S}_{input}$ and $\mathcal{S}_P$ choose all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or compute them directly as described in the protocol, we can conclude that protocol InputMapping is PHVZK.

To prove CWEE, we construct an extractor $\mathcal{X}$ that also uses extractor $\mathcal{X}_P$ to extract witnesses from the PCS scheme we are using (which we know exist for a fact). To start, the extractor $\mathcal{X}$ interacts with the prover and receives polynomial commitments $\vec{A}$, $\vec{V}, \vec{S}$, $\vec{T}$ from the prover.

The extractor generates k_1 and then follows the protocol to get polynomial evaluations $y_{a_1}, y_{\alpha_1}, y_{\omega_1}, y_{\upsilon_1}$ from the prover. The extractor then rewinds and repeats this step l times. Through interpolation, the extractor retrieves witnesses $a_i, \alpha_i, \omega_i, \upsilon_i$ for all i in $\{1, ..., l\}$. Since we know for a fact that e_i cannot alter a_i and evaluations $y_\alpha, y_\upsilon, y_\omega$ all applied to different powers of x, y_a cannot be altered except for a negligible probability.

Plugging witnesses $\vec{a}, \vec{\alpha}, \vec{\upsilon}, \vec{\omega}$ along with the evaluation point k into equation 122 we get:

$$\sum_{i=0}^{l}(a_i + x \cdot \upsilon_i)x \cdot k^i = \sum_{i=0}^{l}(a_i' \cdot x - e_i + (\upsilon_i - \alpha)x^2 + \omega_i \cdot q) \cdot k^i \in \mathbb{Z}_p \tag{125}$$

Since we know e_i cannot alter a_i because $e_i \bmod q = 0$, we can trivially observe that no other value besides a_i on both sides of the equation is multiplied by the single power of x. This implies the equality above must be true for a computationally bounded prover except for a negligible probability (e.g. adversary guessed x correctly), or the soundness PCS scheme is broken, and this satisfies our CWEE definition.

Disclaimer

Special thanks to Danli Xie and Jam Jia fro helping me out on the implementation problems.