Concretely Efficient Input Transformation Based Zero-Knowledge Argument System for Arbitrary Circuits

Frank Y.C. Lu

YinYao Inc.

Abstract. We introduce a new efficient, transparent, interactive zero-knowledge argument system that is based on the new input transformation concept that we will introduce in this paper. The core of this concept is a mechanism that converts input parameters into a format that can be processed directly by the circuit so that the circuit output can be verified through direct computation of the circuit.

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, making the runtime cost of our protocol very efficient. Our benchmark result shows our approach can significantly improve both prover runtime and verifier runtime performance over state-of-the-art by over one order of magnitude while keeping the communication cost comparable with that of the state-of-the-art.

Our approach also allows our protocol to be memory-efficient without forcing it to require a designated verifier. The theoretical memory cost of our protocol is $O(b)$, where b is a parameter set by the user. Lowering the b value will result in better prover runtime performance at the expense of the higher communication cost. Our benchmark result shows the prover speed of our protocol is at least comparable to that of state-of-the-art VOLE-based protocols, but with much smaller proof size and the significant advantage of being non-interactive at the same time.

1 Introduction

Ever since the discoveries of interactive proofs (IPs) [24] 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. [25], and Maller et al. improved the SRS size from quadratic to linear in Sonic [28]. More recently developed protocols such as PLONK [22], MARLIN [17] 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 [16], 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 [33], Virgo [38]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [23] and Ligero/Ligero++ [1] [9] 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 [8] protocols.

NIZKs such as SpartanNIZK [31] and later Lakonia [32] 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 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.

Another category of protocols emphasizes on memory-efficiency such as garbled circuits [27] [21] [26] and Vector Oblivious Linear Evaluation (VOLE) protocols [12] [30] [14] [13] [37] [34] [7] [36] generally offer better prover performance. However, their verifier runtimes are just as expensive and generally require a designated verifier with very expensive communication cost.

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 over those of the state-of-the-art. Finally, we also want our new system to be memory-efficient without requiring a designated verifier like that of VOLE protocols.

2 Summary of Contributions

Our approach is to design a new class of protocols that allows verifiers to validate circuit outputs by directly examining circuit inputs without going through some intermediate translation phase. In our protocol, circuit inputs in the Pedersen commitment form are converted to linear polynomials in $\mathbb{F}_q$ 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 [18] and more recent designated-verifier (which is a limitation itself) VOLE (LPZK in particular) [20] [36] [19] [35] 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 generally 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 (important), and the verifier computed output is still bound to the challenge x. This is a first and brings us three direct benefits; 1) After "computing" the whole circuit using transformed inputs, the verifier can now validate sub-linear sized proof transcripts in sub-linear runtime. 2) Since the whole circuit is linearly/directly computed by the verifier, we can break a large circuit into several smaller sub-circuits, minimizing the memory footprint to that of a sub-circuit. 3) Because the circuit is linearly "computed" by both the prover and the verifier, it gives the developer the power to significantly reduce the size of the circuit by combining "other" protocols in the middle and bypassing the "inactive" part of the circuit logic.

In our protocol, committed input parameters in $\mathbb{G}$ are transformed to linear polynomials in $\mathbb{Z}_q$. For simple, circuit ${a_1}^d+{a_2}^d+{a_3}^d = r$ takes inputs a_1, a_2, a_3 and outputs r. In our protocol, inputs a_1, a_2, a_3 and output r are committed by the prover using Pedersen commitment. The prover also 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 $\mathbb{F}_q$, the verifier can plug these values directly into a circuit and just "execute" them to get an output o e.g. $a'^d_1 + a'^d_2 + a'^d_3 = o$. The circuit output $o \in \mathbb{Z}_p$ is 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 are 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 input transformations are legit, then we say the proof is valid.

The output polynomial in the example above has degree d because the transformed inputs (linear polynomials) have degree 1. Taking them to their dth power and adding them together will get a polynomial of degree d. So if the circuit is something like ${a_1}^3 + {a_2}^3 + {a_3}^3 + , ..., + {a_t}^3 = o$, the output polynomial has a degree of 3 regardless of what the total number of addition and multiplication operations in that circuit is. Throughout our paper, we use the symbol p_d (short for "polynomial degree") to denote the degree of the final polynomial that leads to the circuit output. p_d is different from "multiplication depth" or the "total number of multiplications in a circuit". For example, for a circuit ${a_1}^3 \cdot {a_2}^5 + {a_3}^6 = r$, $s = 7$ (${a_1}^3 \cdot {a_2}^5$ = a polynomial of degree 8), which is bigger than the multiplication depth (5) but smaller than the total number of multiplications (12).

High performance Unlike other zero-knowledge protocols that depend on polynomial commitment, the result of evaluation point x in our protocol is computed by the verifier (through direct computation of transformed inputs in $\mathbb{Z}_q$) itself, so it has to be accurate. In addition, the constant term coefficients are committed by the verifier. This allows us to bypass the expensive PCS protocols and only evaluate a small polynomial of $O(b)$ degrees, where b is a 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 $b = \sqrt{p_d}$.

Specifically, when processing a high-depth circuit of 2^{20} sequential multiplication gates ($s = n$) with 20 inputs on a single CPU thread, the performance of our protocol is: 0.57 seconds for the prover runtime cost; 12 milliseconds for the verifier runtime cost; and 30 kilobytes for the communication cost. To the best of our knowledge, our protocol offers the best prover/verifier runtime performance in the literature by a large margin.

In the memory-efficient setting, the theoretical memory cost of our protocol is $O(b)$. This makes our protocol extremely attractive because VOLE-based memory-efficient protocols generally require one round of interaction and are extremely expensive in terms of verifier runtime cost and communication cost.

Same Format for Circuit Inputs and Outputs Having both inputs and output(s) in the same format (Pedersen Commitment) allows the output(s) of one circuit to be directly reused as inputs to another circuit. This is a really useful feature in practice when verifying data in a publicly accessible/verifiable data store (not limited to blockchain). e.g., allows many participants to continuously manage/update a datastore as long as they can prove these updates were correctly computed from existing data. While other zero-knowledge protocols may 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.

Since both inputs and outputs are in the same formats: Pedersen commitment in $\mathbb{G}$ (standard format) and linear polynomial in $\mathbb{F}_q$ (transformed format). Since data are perfectly hidden under both formats, results of direct computation are automatically perfectly hidden. Therefore, our protocol is zero knowledge as long as we can ensure transcripts used for transformations between two formats are zero knowledge.

Ease of Use Another less appreciated benefit of the input transformation based approach is that developers can just wire a circuit in a much more straight forward fashion than the expensive arithmetization process used in other protocols.

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.

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:

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

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

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

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

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

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 [29]. 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 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:

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

g, h, u 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 $\mathbb{F}$ 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 $p_d + 1$ 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 sub-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 1 states. Before jumping into details, there are three key concepts that our protocol depends on that's 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 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 $\mathbb{F}$ 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 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 using a technique called "breakers" to bootstrapping the p_d value by breaking a circuit into multiple smaller sub-circuits, a concept we will detail in Section 4.2.

4.1 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.

We make sure the domain of transformed input $\mathbb{Z}_q$ 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.

Setup Phase Before the random challenge x is available, the prover commits to two of the group elements S_i, T_i for each ith Pedersen input.

S_i commits the randomly generated blinding keys $\omega_i \in \mathbb{Z}_p$, which will be used to hide witnesses in $\mathbb{F}$. S_i's blinding key υ_i is also the input commitment $\vec{P}_i$ but on a different generator.

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

Note the exponent of g is $w \cdot q$ instead of w, we will see shortly why this is needed.

F_i commits to the difference between each υ_i and α_i, where α_i is the new blinding key for transformed input in $\mathbb{Z}_q$ s.t. $a'_i = a_i + x \cdot \alpha_i \in \mathbb{Z}_q$. This commitment will be used to validate the mapping between the Pedersen input's blinding key υ_i and that of the transformed input in $\mathbb{Z}_q$.

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

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

$Input : (; \vec{a}, \vec{\alpha}, \vec{\omega} \in \mathbb{Z}_q, \vec{\upsilon} \in \mathbb{Z}_p)$
- $\mathcal{P}'s\ input : (; \vec{a}, \vec{\alpha}, \vec{\omega}, \vec{\upsilon})$;
- $\mathcal{P}\ compute :$
  - $S_i = g^{\omega_i \cdot q} u^{\upsilon_i} \in \mathbb{G}$ $\qquad i = \{1, ..., l\}$
  - $T_i = g^{\upsilon_i - \alpha_i} \in \mathbb{G}$ $\qquad i = \{1, ..., l\}$
- $\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}$

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).

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

The part of e_i on the left of the addition sign $(((x\ \alpha_i \bmod q) - x\ \alpha_i) \cdot x)$ 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}. Note this makes $|e_i| \leq |\omega| + |q| \approx$ 244-bits.

With transcripts S_i, T_i, e_i, the verifier can compute a new commitment that closely matches the structure of P_i.

$$g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i = (g^x)^{a_i + x\upsilon_i} u^{\upsilon_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{8}$$

With $(g^x)^{a_i + x\upsilon_i}$ available, we can compute a"public key" PK_{υ_i} for witness υ_i:

$$(h^x/(g^{x^2}u))^{\upsilon_i} = \frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{9}$$

Where $(h^x/(g^{x^2}u))$ is the generator for witness υ_i. 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:

$$PK_{\upsilon_t} = \prod_{i=1}^{l} \left(\frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{10}$$

The verifier can confirm the correctness of all transformations if the prover can prove the knowledge of $\upsilon_t = \sum_{i=1}^{l} \upsilon_i \cdot k^i$ on generator $h^x/(g^{x^2}u) \in \mathbb{G}$. To ensure soundness, note that only elements carrying input data (P_i and a'_i) are taking to the first power x, and 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.

$$\textbf{if } (e_i \bmod q) \stackrel{?}{=} 0, \textbf{ then } \textit{continue} \tag{11}$$

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 10 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.

$$\textbf{if } a_i + (x\alpha_i \bmod q) \geq q, \textbf{then } e_i = e_i - x \cdot q \qquad i = \{1, ..., l\} \tag{12}$$

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 (183-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:

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

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

$\mathcal{P}$ *compute* :

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

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

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

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

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

$\mathcal{P}$ *compute* :

$$\upsilon_t = \sum_{i=1}^{l} \upsilon_i k^i \ \in \mathbb{Z}_p$$

$tr_{\upsilon_t} = ProveDL((h^x/(g^x u)), \upsilon_t)$

$\mathcal{P} \to \mathcal{V} : tr_{\upsilon_t}$

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

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

else *reject*

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

if $VerifyDL((h^x/(g^{x^2} u)), PK_{\upsilon_t}, tr_{\upsilon_t}),$ **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.

To prove PHVZK for relation 4, we define a simulator $\mathcal{S}_{input}$. To start, simulator $\mathcal{S}_{input}$ randomly generates $\vec{S}, \vec{T}$ and sends them to the verifier. After receiving challenge x from the verifier, the simulator first generates a set of linear polynomials $\vec{a}\,'$, and then simulates the proof transcripts proving the mapping between committed values $\vec{P}$ and $\vec{a}\,'$ it generated.

The simulator $\mathcal{S}_{input}$ randomly generates and sends $\vec{e}$ according to the protocol specification (e_i is generated by first randomly generating a value $v_i \in \mathbb{Z}_p$ and multiplying it by q s.t. $e_i = v_i \cdot q$) and sends them to the verifier. When challenge k is received, the simulator $\mathcal{S}_{input}$ calls simulator $\mathcal{S}_{dlog}$ to generate transcript tr_{υ_t} and send it to the verifier.

The verifier then follows the protocol to compute PK_{υ_t} using transcripts $\vec{S}, \vec{T}, \vec{a}\,', \vec{e}$ and calls the $VerifyDL$ function to test it against the input transcript tr_{υ_t}, We already know for a fact that there exists a simulator $\mathcal{S}_{dlog}$ that can simulate witnesses for any discrete-log relation, and that simulators $\mathcal{S}_{input}$ and $\mathcal{S}_{dlog}$ 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. 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 protocol InputMapping is PHVZK.

To prove CWEE, we construct an extractor $\mathcal{X}$ that also uses extractor $\mathcal{X}_{dlog}$ to extract witnesses from proof of knowledge transcripts (which we know exist for a fact). 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 $\vec{e}_1, \vec{a}'_1$, the extractor rewinds and repeats this step with another challenge x_2 to retrieve $\vec{e}_2$, $\vec{a}'_2$.

After receiving transcripts $\vec{S}, \vec{T}$ and transformed inputs $\vec{a}\,'$ from the prover, the extractor generates k_1 and then follows the protocol to get $tr_{\upsilon_{t1}}$ (from the prover), $PK_{\upsilon_{t1}}$. The extractor $\mathcal{X}$ then calls the extractor $\mathcal{X}_{dlog}$ to retrieve υ_{t1} from generator $h^x/(g^{x^2}u)$. The extractor then rewinds and repeats this step l times to retrieve $\upsilon_{t2}, ..., \upsilon_{tl+1}$. Through interpolation, the extractor retrieves witnesses υ_i for all i in $\{1, ..., l\}$. Since we know for a fact that e_i cannot alter a_i and committed values $\vec{P}, \vec{S}, \vec{T}$ all applied to different powers of x, a cannot be altered except for a negligible probability or we find a non-trivial relationship between generators g, h, u.

Using any two different challenges x_i, x_{i+1} we mentioned earlier, the extractor gets $\vec{a}'_1$ and $\vec{a}'_2$ from the prover, 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, u, we can re-write the left and right sides of equation 10 to:

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

Take out challenge k_i, for each ith input we have:

$$\frac{h^{x \cdot \upsilon_i}}{g^{x^2 \upsilon_i} u^{\upsilon_i}} = \frac{g^{a_i \cdot x} h^{\upsilon_i \cdot x}}{g^{a'_i \cdot x - e_i + (\upsilon_i - \alpha_i)\cdot x^2 + \omega_i \cdot q} \,.\, u^{\upsilon_i}} \in \mathbb{G}$$

This implies generator g's exponent in the nominator of the right-hand side element must cancel out, which automatically implies the g's exponent in the denominator of the right hand side element must be $a_i \cdot x + x^2 \upsilon_i$ for g's exponent on the left hand side to be $x^2 \upsilon_i$. Since we know e_i cannot alter $a_i \in \mathbb{Z}_q$ because $e_i \bmod q = 0$, we can trivially observe that no other value besides a_i in g's exponent in the denominator $((a'_i) \cdot x - e_i + (\upsilon_i - \alpha_i) \cdot x^2 \; + (\omega_i \cdot q))$ 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 (adversary guessed x correctly), or we find a non-trivial relationship between generators g, h, u, and this satisfies our CWEE definition.

4.2 The Complete Protocol

To prove the circuit output is correctly computed from executing transformed inputs $\vec{a}'$, the prover provides the proof for all coefficients of the output polynomial. 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:

$$o = a_1\,' + a_2\,' = r + x \cdot \epsilon \tag{15}$$

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:

$$o = a_1\,' \cdot a_2\,' = r + x \cdot \epsilon + x^2 \cdot \tau \tag{16}$$

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. The degree of the polynomial (p_d) will increase after each multiplication operation, so the efficiency will also drop as there are more computations involved at higher degrees.

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 commits to τ before the challenge x is known. When the challenge x is available, the prover sends y, the sum of products of coefficients of degree 2 and higher at evaluation point x, to the verifier and proofs to prove $x^2\tau = y$. The verifier can then get the output in the linear polynomial form defined in equation 3:

$$r' = o - y \tag{17}$$

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, the circuit 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 $y_i = \sum_{j=1}^{b} \tau_{i,j} \cdot x^{j+1}$.

$$r_i' = o_i - y_i \qquad \text{for} \qquad i = \{1, ..., s\} \tag{18}$$

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 set breakers 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 each sub-circuit 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; this allows us to build an efficient protocol by 1) using vector commitment to commit the output pair (r_i, ϵ_i) of each sub-circuit/sub-polynomial, 2) evaluating all other coefficients and checking if they can compute to each breaker y_i.

First, we commit to the outputs of each sub-circuit. Instead of using Pedersen commitments to commit each output as we do with the circuit output R, we use two vector commitments to batch commit the outputs of all sub-circuits. Since we use the ECC group to commit values in $\mathbb{Z}_q$, we define an opening $r_{p_i} = r_i + x \cdot \epsilon_i \in \mathbb{F}$ s.t. r_{p_i} does not overflow p.

Since r_{p_i} is a raw number with approximately $2|q|$ bits, anyone can easily extract witnesses r_i, ϵ_i with x, so we use a blinding key $\omega_i \in \mathbb{Z}_m$ that is 80 bits longer than $|q|$ to hide r_{p_i}.

$$\omega_i \xleftarrow{\$} \mathbb{Z}_m \qquad i = \{1, ..., l\} \tag{19}$$

$$r'_{p_i} = r_i + x \cdot \epsilon_i + \omega_i \cdot q \in \mathbb{Z}_p \qquad i = \{1, ..., s\} \tag{20}$$

$$R_t = \prod_{i=1}^{s} u_i^{r_i + \omega_i \cdot q} \in \mathbb{G}, \qquad E_t = \prod_{i=1}^{s} u_i^{\epsilon_i} \in \mathbb{G} \tag{21}$$

Once mod q is applied to r'_{p_i}, the blinding term $\mu_i \cdot q$ will disappear.

$$r'_i = r_{pi} \bmod q \in \mathbb{Z}_q \qquad i = \{1, ..., s\} \tag{22}$$

After challenge x is known, the prover sends $\vec{r_p}'$ to the verifier. The verifier uses the following equality to check if the set $\vec{r_p}'$ matches the committed values.

$$\prod_{i=1}^{s} g_i^{r_{pi}} = \prod_{i=1}^{s} R_i \cdot E_t^x \tag{23}$$

The output o_i of each polynomial evaluation at point x is computed by the verifier directly using inputs and circuit logic. Since r'_i is a committed value, this implies each breaker y'_i is also a committed value because it can be directly computed from o_i and the committed output r'_i.

$$y'_i = o_i - r'_i \qquad \text{for} \qquad i = \{1, ..., s\} \tag{24}$$

We can observe that the verifier can validate $y'_i = \tau_{i,1}x^2 +, ..., +\tau_{i,b}x^{b+1}$ itself if coefficients are passed to them. To make it safe to do so, we use a challenge w that's made available after the prover commits to R_t, E_t but before the challenge x is available to allow the prover to hide and pass all coefficients for each degree term in one single element z_j.

$$z_j = \sum_{i=0}^{s} \tau_{i,j} \cdot w^i \in \mathbb{Z}_q \qquad j = \{1, ..., b\} \tag{25}$$

z_0 is the new blinding key added to each ϵ_i to protect the special cases when $\epsilon_i = 0$ s.t.:

$$\epsilon_i = \epsilon_i - \tau_{i,0} \in \mathbb{Z}_q \tag{26}$$

The updated matrix with blinding elements $\tau_{1,0}, ..., \tau_{s,0}$ is presented in Figure 3, note challenge w also applies to $\vec{y}'$.

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

Figure 3

When challenge x is available, the verifier validates $\vec{y}'$ by multiplying each y'_i by w^i and each z_j by x^{j+1}; their difference must be equal to 0.

$$\sum_{j=0}^{b} z_j \cdot x^{j+1} = \sum_{i=0}^{s} y_i \cdot w^i \in \mathbb{Z}_q \tag{27}$$

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 (some of it may be 0), the dishonest prover needs to find a set $\vec{\Delta}$ before challenge x is known s.t.

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

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.

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

To get equation 27 work, the prover needs to send $\vec{z}$ to the verifier. While this works, the more secure and communication-cost-efficient option is to use a polynomial commitment scheme to commit and evaluate $\vec{z}$ at evaluation point x. Since the size of $|\vec{z}| = \sqrt{p_d}$, the prover and verifier runtime cost for running PCS is insignificant so we focus on minimizing the communication cost. The PCS scheme we use in our protocol is based on the one defined by Bootle et al. [10]. We define two functions that our protocol will use to commit and evaluate coefficients $\vec{z}$ at evaluation point x.

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

Let $y_t \in \mathbb{Z}_q$ be the evaluation output at point x for a univariate polynomial with coefficients $\vec{z}$ committed in $\vec{C}$. The equation 27 is now updated to:

$$y_t = (\sum_{i=1}^{s} y_i \cdot w^i) \bmod q \tag{28}$$

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

4.3 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 1, 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:

$$Input : (\vec{P}, R \in \mathbb{G};\ \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_p) \tag{29}$$

$$\mathcal{P}'s\ input : (\vec{P}, R;\ \vec{a}, \vec{v}, r, \phi);\quad \mathcal{V}'s\ input : (\vec{P}, R) \tag{30}$$

$$\mathcal{P}\ compute : \tag{31}$$

$$\tau_{i,0} \xleftarrow{\$} \mathbb{Z}_p, \qquad i = \{1, ..., s\} \tag{32}$$

$$\alpha_i \xleftarrow{\$} \mathbb{Z}_q, \qquad i = \{1, ..., l\} \tag{33}$$

$$\textbf{for}\quad i = 1, ..., s\quad \{ \tag{34}$$

$$r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}', r_i, \epsilon_i, \vec{\tau}_i); \tag{35}$$

$$\epsilon_i = (\epsilon_i - \tau_{i,0}) \bmod p\quad \}\quad // \text{ note } r_s = r \tag{36}$$

$$R_t = \prod_{i=1}^{s} g_i^{r_i + \mu_i \cdot q} \in \mathbb{G}, \qquad E_t = \prod_{i=1}^{s} g_i^{\epsilon_i} \in \mathbb{G} \tag{37}$$

$$\text{InputMapping-Setup}(; \vec{a}||r_s, \vec{\alpha}||\epsilon, \vec{v}||\phi) \rightarrow : \vec{S}, \vec{T} \tag{38}$$

$$\mathcal{P} \rightarrow \mathcal{V} : R_t, E_t, \vec{S}, \vec{T} \tag{39}$$

$$\mathcal{V} \rightarrow \mathcal{P} :\quad w \xleftarrow{\$} \mathbb{Z}_p \tag{40}$$

$$\mathcal{P}\ compute : \tag{41}$$

$$z_j = \sum_{i=1}^{s} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{0, ..., b\} \tag{42}$$

$$\text{polyCommit}(\vec{z}) \rightarrow : \vec{C} \tag{43}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{C} \tag{44}$$

$$\mathcal{V} \rightarrow \mathcal{P} :\quad x \xleftarrow{\$} \mathbb{Z}_p \tag{45}$$

$$\mathcal{P}\ compute : \tag{46}$$

$$a'_i = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{47}$$

$$r'_{p_i} = r_i + \mu_i \cdot q + x \cdot \epsilon_i \in \mathbb{Z}_p \qquad i = \{1, ..., l\} \tag{48}$$

$$y_t = \sum_{j=0}^{s} z_j \cdot x^{j+1} \in \mathbb{Z}_q \tag{49}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{a}', \vec{r}'_p, y_t \tag{50}$$

$$\mathcal{V}\ verify\ final\ output: \tag{51}$$

$$r'_i = r'_{p_i} \bmod q \in \mathbb{Z}_q \qquad i = \{1, ..., s\} \tag{52}$$

$$\textbf{for}\quad i = 1, ..., s\quad \{ \tag{53}$$

$$o_i = computeSubCircuit_i(\vec{a}'||\vec{r}') \in \mathbb{Z}_p \tag{54}$$

$$y'_i = o_i - r'_i \in \mathbb{Z}_q\quad \} \tag{55}$$

$$\textbf{if}\ (\prod_{i=1}^{s} g_i^{r_i} = \prod_{i=1}^{s} R_i \cdot E_t^x \tag{56}$$

$$\wedge\ \text{polyEval}(\vec{C}, y_t; \vec{z})\ \wedge\ y_t = (\sum_{i=1}^{s} y_i \cdot w^i) \bmod q)\ \textbf{then}\ continue \tag{57}$$

$$\textbf{else}\ reject \tag{58}$$

$$\textbf{if}\ \text{InputMapping-Verify}(\vec{P}||R, \vec{S}, \vec{T}, \vec{a}'||r'_s; \vec{a}||r, \vec{\alpha}||\epsilon, \vec{v}||\phi) \tag{59}$$

$$\textbf{then}\ continue \tag{60}$$

$$\textbf{else}\ reject \tag{61}$$

Protocol AriCircuit

Note that a b degree polynomial can have at most b roots, and therefore there is a negligible probability of b/p that an attacker can play with coefficients to pass the validation test of an altered output.

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 see.

To prove PHVZK for relation 1, we define a simulator $\mathcal{S}$. Simulator $\mathcal{S}$ calls on simulators $\mathcal{S}_{input}$ defined earlier to generate transcripts and simulate interactions in the InputMapping sub-protocol used in our protocol.

We have already shown $\mathcal{S}_{input}$ can simulate all interactions needed in sub-protocol InputMapping. We now show how $\mathcal{S}$ generates the rest of the transcripts according to the randomness supplied by the adversary from their respective domains or computes them directly as described in the protocol.

Simulator $\mathcal{S}$ randomly generates random input witnesses $\vec{a}^*, \vec{\alpha}^*$ and computes circuit output witnesses $\vec{r}^*, \vec{\epsilon}^*, \vec{\tau}^*$ and creates commitments $R_t,\ E_t$ accordingly to the protocol specification and sends them to the verifier. The simulator then follows the protocol to create $\vec{z}$ with challenge w and sends them to the verifier. After receiving challenge x from the verifier, the simulator computes $\vec{a}'^*, \vec{r_s}'^*$ according to the protocol specification. Note that after randomly generating input witnesses $\vec{a}^*, \vec{\alpha}^*$, the simulator $\mathcal{S}$ just followed the protocol specification to create all other transcripts needed. The rewinding only takes place in the simulator $\mathcal{S}_{input}$. This implies that if the input-mapping process is PHVZK, then the whole 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 extractor $\mathcal{X}$ that calls on extractors $\mathcal{X}_{input}$ defined earlier to extract witnesses for the two sub-protocols used in the protocol AriCircuit.

We already know $\mathcal{X}$ can extract $\vec{a}, \vec{\alpha}, \vec{v}$ and $\vec{r}, \vec{\epsilon}$ using extractor $\mathcal{X}_{input}$ from committed transcripts $\vec{S}, \vec{T}$. Using input witnesses, we can use the function computeSubCircuitKeys to compute circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$. Next, we show how these witnesses must match ones extracted from circuit transcripts $\vec{r_p}', \vec{z}, R_t, E_t$.

First, the extractor $\mathcal{X}$ rewinds b times to extract $b+1$ polynomial outputs $\vec{y_t}$ from evaluation points $x_1, ..., x_{b+1}$, and extracts coefficients $\vec{z}$ using interpolation. With $\vec{z}$, the extractor $\mathcal{X}$ rewinds one more time to line 39 and then rewinds another s times generates $s+1$ challenges $w_1, w_2, ..., w_{s+1}$, using which the extractor can extract witnesses $\vec{\tau}_0, ..., \vec{\tau}_s$ using interpolation. Rearrange equality 27 and substitute $\vec{z}$ for $\vec{\tau}$ as specified in 25 we get the following equality:

$$\sum_{j=1}^{b}(\sum_{i=1}^{s}\tau_{i,j}w^i)\cdot x^{j+1} = \sum_{i}^{s} y'_i \cdot w^i \in \mathbb{Z}_q \tag{62}$$

Next, the extractor $\mathcal{X}$ extracts $\vec{r}, \vec{\epsilon}$ from transcripts $\vec{r_p}'$ that satisfy the condition held in the committed values in R_t, E_t. The extractor $\mathcal{X}$ can use a pair of challenges x_1, x_2 to trivially extract $\vec{r}, \vec{\epsilon}$ from $\vec{r_p}'$, and they trivially satisfy the condition held by committed values in R_t, E_t or else the equality test 23 will not stand. With $\vec{r}, \vec{\epsilon}$ extracted, the equality 62 is updated to following:

$$\sum_{j=1}^{b}(\sum_{i=1}^{s}\tau_{i,j}w^i)\cdot x^{j+1} = \sum_{i}^{s}(o_i - (r_i + x\cdot\epsilon_i))\cdot w^i \in \mathbb{Z}_q \tag{63}$$

Knowing that $\vec{o}$ must be correct since it was directly computed by the verifier and that $\vec{r}, \vec{\epsilon}$ match the committed values, the coefficients $\vec{\tau}$ extracted must satisfy equation 25 for equality above to be true for any pairs of challenges w, x except for a negligible probability.

Finally, we check if the extracted circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$ extracted from circuit transcripts match those computed from input witnesses $\vec{a}, \vec{\alpha}$ using computeSubCircuitKeys functions. Since the computed from computeSubCircuitKeys functions also need to satisfy equality 63 for the same evaluation result $\vec{o}$ and match the commitments for $\vec{r}, \vec{\epsilon}$ given any pairs of challenge w, x. The witnesses computed from $\vec{a}, \vec{\alpha}$ must match these extracted from circuit transcripts except for a negligible probability, or else we find a non-trivial discrete log relationship between generators g, h (for input witnesses).

4.4 Customized Sub-Circuit and The Use of 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 (ePrint)). 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.

It is also not necessary that all breakers $y_1, ..., y_m$ have the same b value. For example, if some value a_i is taken to its 100th power $(a_i'^{100}$, degree term $s = 100)$ and will be used as inputs in multiple places of a circuit, then it would be wise to use a breaker to cut its degree to 1 (in linear polynomial form $a_i^{100} + X\alpha_i$) before being used as inputs in other places of a circuit.

4.5 Memory Efficiency

The memory consumption cost of our protocol is $O(s)$. However, our design approach allows us to improve the memory consumption cost of our protocol to $O(b + 2s)$. We only need to do two thing to achieve a memory cost of $O(b + 2s)$: First, delete $\vec{\tau}$ from memory in line 34; Second, recompute $\vec{\tau}_i$ after challenge x is available so that $\vec{y}'$ can be computed (in line 41).

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 $b = p = \sqrt{s}$ in the default setting, the memory cost is approximately $O(3\sqrt{s})$.

4.6 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.

5 Performance Comparison

We compare the performance of our protocol to some of the most popular transparent zero-knowledge protocols for which open source codes are available. Our test runs are performed on an Intel(R) Core(TM) i7-9750H CPU @ 2.60 Ghz. 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 Hyrax, Ligero, Aurora, and Spartan-NIZK. These protocols were chosen because they are the most representative of popular zero-knowledge protocols and can be verified with open source code. In particular, Aurora outperforms STARK in all key parameters (prover/verifier runtime, proof size), and the NIZK version of Spartan offers the most balanced performance across all performance parameters. We do not consider SNARKs because they are hardly efficient after switching to transparent mode.

We didn't consider protocols that depend on circuit depth, such as GKR-based protocols, because they cannot handle 2^{20} sequential multiplications.

Spartan++ and Lakonia are two more recent developments that we didn't include in our benchmark testing but are worth mentioning. The improvement of Spartan++ over SpartanNIZK is marginal, and the performance of Lakonia is largely comparable to that of SpartanNIZK. (The prover performance of SpartanNIZK is approximately 3X more efficient, and the verifier performance is 1.5X more efficient than that of Lakonia, while Lakonia is 4X more efficient than SpartanNIZK in proof size).

We set the number of inputs to our protocol to 20 integers. The circuit we use performs n sequential multiplications on l inputs. This is because we want to demonstrate that our protocol can handle

high-depth circuits (e.g., $p_d = n$) . If we run a shallow circuit where polynomial degree is small, the benchmark result will likely be significantly better. For example, if we have a circuit that computes the sum-of-products of inputs where $n = 2^{20}$ (e.g., $\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 .... + a_n b_n$), the prover will complete proving such a shallow circuit (one million multiplications and one million sums) in under 50 milliseconds for such a shallow circuit with $p_d = 1$ (group exponentiation cost is approaching zero and NTT is not even necessary for field operations).

There are special cases where in some sub-circuits we have p_d value grow faster than the number of multiplications performed (e.g., o = $(((a'_1)^2)^2)^2$). In such cases, we may want to place breakers more aggressively to better handle the situation. This can be done by either setting fixed $b < \sqrt{p_d}$ or placing more breakers inside these uncommon sub-circuits. In a real-world situation where there are both multiplication and addition operations, it is unlikely that p_d is bigger than n.

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. Such tuning will likely not be required in real-world applications since large circuits are usually layered and multiplication gates should be somewhat balanced out across layers already.

5.1 Benchmark at different at balanced b value

We set $b = s = \sqrt{s}$ to get a more balanced result. Alternatively, one can set b smaller to get better prover runtime performance in exchange for more expensive verifier and communication costs (section 5.2). This is because doing so will 1) may significantly cut down the polynomial degree p_d value of the circuit. 2) evade expensive NTT computations at high degrees and better leverage Pippenger acceleration in computing R_t, E_t, which will continuously improve group exponentiation operations before peaking out at around $m = 2^{14}$.

Table 1. Prover performance comparison (seconds)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Hyrax	1	2.8	9	36	117	486
Ligero	0.1	0.4	1.6	4	17	69
Aurora	0.5	1.6	6.5	27	116	485
SpartanNIZK	0.02	0.05	0.16	0.6	1.7	6.2
This work	0.004	0.005	0.01	0.03	0.13	0.57

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 shows that the communication cost of our protocol dominates that of Ligero and Aurora, while being largely comparable to SpartanNIZK and Hyrax. The fixed cost of one additional input is 112 bytes, you can add or subtract as many inputs as needed to get the communication cost that fits your scenario rather than take the default $l = 20$. Please note that other protocols also incur comparable costs when they map more witnesses to some pre-committed/encrypted value in the public data store.

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

Table 2. Proof size comparison (kilobytes)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Hyrax	14	17	21	28	38	58
Ligero	546	1,076	2,100	5,788	10,527	19,828
Aurora	477	610	810	1,069	1,315	1,603
SpartanNIZK	9	12	15	21	30	48
This work	3	4	7	11	19	36

Table 3. Verifier performance comparison (milliseconds)

Circuit size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Hyrax	206	253	331	594	1.6s	8.1s
Ligero	50	179	700	2s	7.5s	33s
Aurora	192	590	2s	7.2s	29.8s	118s
SpartanNIZK	7	11	17	36	103	387
This work	1.4	1.4	1.5	1.8	3	12.5

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 ($b = 2^{10}$ when $s = 2^{20}$ and q is a 61-bit prime), 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 dumb and 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 only needs to compute vector commitment R_t once), 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. e.g., at 128-bit, which will require optimization at the assembly level for which no open source code is available at the moment, unlike that for 64-bit (come with the CPU) and 256-bit (optimized over the years because of ECC).

It is worth noticing that input transformation costs can be shared across multiple circuits if the inputs are reused as inputs to other circuits, which may lead to further reductions in communication costs in the real world.

5.2 Benchmark at different b value in memory efficient setting

One of the biggest advantages of our protocol is that it provides a blueprint for developing a fast, memory-efficient, non-interactive zero-knowledge protocol. The theoretical memory cost is $O(b + s)$. Using the memory-efficient implementation mentioned in Section 4.5, we get the results listed in Table 4.

The prover performance shown in Table 5 shows our protocol peaks at around 1.43 million multiplications per second when $b = 2^5$, then it starts to increase again afterwards. This is because the cost of computing the vector commitments R_t, E_t gets increasingly expensive as the number of sub-circuits (m) increases. Therefore, the cost of performing group exponentiation operations committing $\vec{r}, \vec{\epsilon}$ is getting more expensive relative to the shrinking cost of field operations computing $\vec{r}, \vec{\epsilon}, \vec{\tau}$ as b decreases.

Table 4. Performance comparison in memory efficient setting

b	Prover	Verifier	Communication Cost
2^{10}	1.14 s	11 ms	55 KB
2^{9}	1.03 s	11 ms	55 KB
2^{8}	0.93 s	18 ms	106 KB
2^{7}	0.83 s	27 ms	208 KB
2^{6}	0.74 s	50 ms	414 KB
2^{5}	0.70 s	100 ms	827 KB
2^{4}	0.72 s	201 ms	1.6 MB
2^{3}	0.91 s	397 ms	3.3 MB

However, table 4 is a little bit deceiving because the smaller the b may also lead to a smaller polynomial degree of a circuit (e.g., when some output of a sub-circuit with a high degree is used multiple times to multiply other values of a circuit.), which in turn leads to faster runtime performance.

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 their paper [36]), given that our protocol is non-interactive and offers a significantly smaller proof size without requiring pre-setup like that of Ant-Man.

Table 5. 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	$\frac{1}{b}$	$\geq$ 1.43 M	Yes

Note that there are other techniques for improving prover memory: Commit-and-prove to glue sub-circuits together (Lunar/Eclipse [15] [2]), streaming SNARKs (Gemini [11]). However, the reported constructions of these protocols require trusted setup (non-transparent), and the prover runtime cost of these protocols is magnitudes more expensive.

5.3 Other Thoughts

Very recently, major advances in multivariate polynomial commitment schemes has been made. One intriguing example is Binius, which is promising especially for proving binary field operations. However, it is not clear how much efficiency gain it can offer for integer arithmetics over u16/u32/u64 given the added complexity it may require to support it. Even if we count the cost of PCS schemes to zero for protocols leverages multivariate PCS schemes in exchange for more expensive communication cost, there are still 25% - 50% of prover cost it can't reduce and therefore cannot get even close to the benchmark performance our protocol offers.

6 Acknowledgments

Special thanks to Danli Xie and Jam Jia for helping out on the coding and making the code more efficient.

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., 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
8. 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
9. 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
10. 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
11. 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
12. 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
13. 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
14. 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
15. 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

16. 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
17. 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
18. 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
19. 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
20. 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`
21. 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
22. 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`
23. 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)
24. 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
25. 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
26. 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
27. 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
28. 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
29. 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
30. 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
31. 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
32. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), `https://eprint.iacr.org/2020/1275`
33. 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`

34. 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
35. 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
36. 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
37. 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
38. 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:

$$f(X) = (1\ X^n ...\ X^{(m-1)n}) \begin{pmatrix} z_{0,0} & z_{0,1} & . & . & z_{0,n-1} \\ z_{2,0} & z_{2,1} & . & . & z_{2,n-1} \\ . & . & & & . \\ . & . & & & . \\ . & . & & & . \\ z_{m-1,0} & z_{n-1,1} & . & . & z_{m-1,n-1} \end{pmatrix} \begin{pmatrix} 1 \\ X \\ . \\ . \\ . \\ X^{n-1} \end{pmatrix}$$

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:

$$\vec{c} = (1\ x^n ...\ x^{(m-1)n}) \begin{pmatrix} z_{0,0} & z_{0,1} & . & . & z_{0,n-1} \\ z_{2,0} & z_{2,1} & . & . & z_{1,n-1} \\ . & . & & & . \\ . & . & & & . \\ . & . & & & . \\ z_{m-1,0} & z_{n-1,1} & . & . & z_{m-1,n-1} \end{pmatrix}$$

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.

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

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 $\mathbb{Z}_p$, but is limited to a $|\sum_{i=0}^{m-1} z_{i,j} \cdot x^{j \cdot n}| + 80$-bits number, which is approximately ≈ 207-bits in our case $n = m = 2^5$, enough to hide coefficients of T_i. For 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 will get cancelled by $u_{qj} = u_j \bmod q$ once the domain of the final evaluation is reduced to $\mathbb{Z}_q$.

B. The sub-protocol for boolean circuit validation

For boolean circuits, we need to enforce input data $b_i \in \{0, 1\}$ for some $i \in \{1, ..., l\}$. 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) in order to perform comparison operations on input data. If a committed value b_i is in $[0, 1]$, then its linear polynomial form b_i must have the following property:

$$(b'_i \cdot b'_i - b'_i) = \delta_{1,i} x + \delta_{2,i} x^2 \tag{64}$$

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. To prove the correctness for all $b_i \in \{0, 1\}$, the prover commits to polynomials D_1, D_2:

$$D_1 = u_1^{\delta_{1,1}} u_2^{\delta_{1,2}} ... u_l^{\delta_{1,l}} h^{\rho_1}, \qquad D_2 = u_1^{\delta_{2,1}} u_2^{\delta_{2,2}} ... u_l^{\delta_{2,l}} h^{\rho_2}$$

Where D_1 commits to coefficients on x term for $i \in \{1, ..., l \cdot 32\}$ and D_2 commits to coefficients on x^2 term for $i \in \{1, ..., l \cdot 32\}$. The prover can easily join two polynomial commitments into one and sends only one element D to the verifier.

$$D = \prod_{i=1}^{l} u_i^{\delta_{1i}} \cdot \prod_{i=1}^{l} u_{i+l}^{\delta_{2i}} \cdot h^{\rho} \in \mathbb{G} \tag{65}$$

When the challenge k is received, the prover sends the evaluation results y_1, y_2 to the verifier and engages with the verifier to verify the correctness of y_1, y_2 at point k, and checks if the equality below is true:

$$y_1 \cdot x + y_2 \cdot x^2 = \sum_{j=1}^{l \cdot 32} (b'_i \cdot b'_i - b'_i) \cdot k^i \tag{66}$$

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 $j = \{1, ..., 32\}$.

$$a'_i = \sum_{j=1}^{32} b'_{i,j} \cdot 2^j \tag{67}$$

We define the protocol ValidateBooleanity using two sub-protocols:

$Input : (\vec{b}' \in \mathbb{Z}_q : \vec{b}, \vec{\beta}' \in \mathbb{Z}_q)$

$\mathcal{P}$ *compute* :

$l = |\vec{b}'| \in \mathbb{Z}_q$

$\rho \xleftarrow{\$} \mathbb{Z}_q$

$\delta_{1i} = b_i\beta_i + b_i\beta_i - \beta_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$

$\delta_{2i} = \beta_i^2 \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$

$$D = \prod_{i=1}^{l} u_i^{\delta_{1,i}} \cdot \prod_{i=1}^{l} u_{i+l}^{\delta_{2,i}} \cdot h^{\rho} \in \mathbb{G}$$

Protocol Booleanity-Setup

While our protocol can be tailored to support boolean circuits, our protocol is optimized for arithmetic circuits that can handle programming logics directly. We believe the most intuitive way to use zero knowledge proof for business users and developers is to avoid using boolean circuit altogether, unless it is for special use cases like proof knowledge of a hash's pre-image.