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, our protocol only requires the prover to use vector commitment to commit to the square root of the polynomial degree ($\sqrt{p_d}$) the circuit generates. 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 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) [23] 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. [24], and Maller et al. improved the SRS size from quadratic to linear in Sonic [27]. More recently developed protocols such as PLONK [21], MARLIN [16] 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 [15], 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 [32], Virgo [37]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [22] 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 [30] and later Lakonia [31] 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.

Another category of protocols emphasizes on memory-efficiency such as garbled circuits [26] [20] [25] and Vector Oblivious Linear Evaluation (VOLE) protocols [11] [29] [13] [12] [36] [33] [7] [35] 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 offers great flexibility to optimize and 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 [17] and more recent designated-verifier (which is a limitation itself) VOLE (LPZK in particular) [19] [35] [18] [34] 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, we begin by transforming each committed input parameter in $\mathbb{G}$ into its linear polynomial form 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 then provides the transformed inputs a_1, a_2, a_3 in the 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 the circuit and just "execute" them to get the output o e.g. $a_1'^d + a_2'^d + a_3'^d = o$. 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 d coefficients are its 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) 2) all input transformations are legit, then we say the proof is legit.

The output polynomial in the example above has a degree of d because the transformed inputs (linear polynomials) are of degree 1. Taking to its dth power will give a polynomial with a degree of d. So if the circuit is something like ${a_1}^3 + {a_2}^3 + {a_3}^3 +, ..., + {a_t}^3 = o$, the degree of the output polynomial is 3 regardless of the value of t. 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. Precisely, it is actually one less than the degree of the output polynomial (e.g. if the degree of the output polynomial is 3, then $p_d = 2$). p_d is different from "multiplication depth" or the 'total number of multiplications in a circuit'". For example, for a circuit ${a_1}^3 \cdot {a_2}^5 + {a_3}^6 = r, \; p_d = 7$ (${a_1}^3 \cdot {a_2}^5 =$ 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 depends on polynomial commitment, the result of evaluation point x is computed by the verifier (through direct computation of verified inputs in $\mathbb{Z}_q$) not sent by the prover, so it has to be accurate. In addition, the constant term and degree 1 term coefficients are also committed by the verifier. This allows us to bypass the expensive polynomial commitment evaluation protocols and only commits to $O(b)$ coefficients, 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 our benchmarking, we set $b = \sqrt{p_d}$.

Specifically, when processing a high-depth circuit of 2^{20} sequential multiplication gates ($p_d = n$) with 20 inputs on a single CPU thread, the performance of our protocol is: 0.7 seconds for the prover runtime cost; 8 milliseconds for the verifier runtime cost; and 39 kilobytes for the communication cost. To the best of our knowledge, our protocol offers the best prover/verifier runtime performance in the literature (transparent/non-interactive/high-depth protocols) by a large margin.

On the memory side, 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 be directly reused as inputs to other 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 feature in theory by mapping witnesses to some publicly accessible committed/encrypted data, it does not come naturally and will require additional cost that is not accounted for.

Zero-Knowledgeness Under our protocol, 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.

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.

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

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 $\mathcal{A}$:

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

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

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

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

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

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

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

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

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

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

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

3.3 Zero Knowledge Proof of Discrete Logarithm

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

- ***ProveDL***$(g, \kappa) \rightarrow tr_\kappa$ generates the proof transcript tr_κ, where κ is the witness.
- ***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 [28]. 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 a committed value $R \in \mathbb{G}$ 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 $\mathbb{F}$, where the verifier can perform addition and multiplication operations "as is". For an input commitment P_i s.t. $P = g^{a_i} h^{\upsilon_i} \in \mathbb{G}$ where a_i is the input value and υ_i is its blinding key, we create a corresponding value in the linear polynomial format ${a_i}' \in \mathbb{Z}_q$:

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

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 a_i' ($i = \{1, ..., l\}$) by the verifier. Since inputs represented by linear polynomials 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 prove the circuit output is correctly computed from circuit inputs as relation 1 states.

Note that we use two prime group orders p, q in our protocol. This is because all transformed inputs are in field $\mathbb{Z}_q$ and their commitments are in group $\mathbb{G}$ of order p. In the default setting, q is a 61-bit number, which is significantly smaller than 253-bit p (curve 25519). Therefore, we must ensure operations in $\mathbb{G}$ do not make committed values (including blinding keys) overflow p, because that would distort the committed values once we convert them back to linear polynomials in $\mathbb{Z}_q$.

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

A sub-protocol that validates committed inputs in $\mathbb{G}$ is correctly mapped to transformed inputs in $\mathbb{Z}_q$. The relation we try to prove for this sub-protocol is:

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

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. Here we also limit $\vec{a} \in \mathbb{Z}_q$ instead of $\mathbb{Z}_p$ in relation 1. This is justified if q is large enough (e.g. over 32-bits).

An important requirement for input transformation is that we want to transform the hidden value in Pedersen commitment to a prime field that 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 $O(p_d \log p_d)$. NTT requires a prime modulo q s.t. $q = r \cdot 2^k + 1$ to be the prime order of the group, where r and k are arbitrary constants, so we need to pick a prime q that is NTT friendly. Also note that the prime q is expected to be smaller than p because the smaller the q value in bits, the lower the communication cost.

Setup Phase Before the random challenge x is available, the prover commits to two sets of group elements. The first element set are blinding elements used to hide data from other transcripts, and the second element set are commitments to blinding key $\vec{\alpha}$, the new blinding keys of $\vec{a}'$ in $\mathbb{Z}_q$.

On the blinding element, the prover first randomly generates blinding keys $\vec{\omega} \in \mathbb{Z}_p^l$ and commits them using generator u and the blinding key of input commitments $\vec{P}$.

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

Instead of committing the new blinding keys $\vec{\alpha}$, the prover commits to the difference between each υ_i and α_i. This will prevent the dishonest prover from adjusting the value of $\vec{\alpha}$ once x is known, and will also be used to link the blinding keys of each a'_i and P_i.

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

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

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

Protocol InputMapping - Setup

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

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

The prover and then computes e_i for each input a_i s.t.:

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

We label $e_i \in \mathbb{Z}_{pq}$ because the upper bound of e_i is $p \cdot q$. This is because the size of the first part of e_i $(((x\ \alpha_i \bmod q) - x\ \alpha_i) \cdot x)$ is very small (around $3|q|$ or $3 * 61 = 183$ bits) compared to the blinding element wq ($|p| + |q| > 313$ bits), so the chance of $e_i > pq$ is negligible, and the prover can just re-generate another ω_i if needed.

e_i doesn't break the zero-knowledge requirement since it does not leak any information to the verifier. This is because the first part of e_i: $((x\ \alpha_i \bmod q) - x\ \alpha_i) \cdot x$ is a multiple of q, and is equivalent to $(s \cdot q) \cdot x$ for some $s \in \mathbb{Z}_q$. This implies $e_i = (s \cdot x + \omega) \cdot q$ for some randomly chosen ω. Since $\omega \in \mathbb{Z}_p$ is significantly larger than $s \cdot x$ ($|s \cdot x| = 2|q|$), it can perfectly hide $s \cdot x$. In our benchmarking, we use curve25519 in our implementation, where p is a 2^{253} bit number. For $|q| = 2^{61}$ and $|s \cdot x| = 2^{122}$, there is only a negligible chance ($\frac{q^2}{p} \approx 2^{-131}$) that $\omega \in \mathbb{Z}_p$ does not perfectly hide s (even if such case happens, the prover can just randomly choose another ω).

The reason for creating $e_i \in \mathbb{Z}_{pq}$ is that we want the verifier to subtract out the blinding element $(x\ \alpha_i \bmod q)$ from a'_i (e.g., $a'_i \cdot x - e_i = (a_i + x\alpha_i) \cdot x - \omega_i q = x(a_i + x\alpha_i) - \omega_i q)$, assuming there is no overflow. The verifier can take out the remaining blinding element $\omega_i q$ and replace $x\alpha_i$ with Pedersen inputs' blinding key $x\upsilon_i$ using the previously committed values S_i, T_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, the verifier can trivially divide each P_i and take their sum with powers of k to get PK_{υ_t}.

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

$PK_{\upsilon_t} = (h^x / (g^{x^2} u))^{\upsilon_t}$. The verifier can confirm the correctness of the transformation if the prover can prove the knowledge of υ_t on generator $h^x / (g^{x^2} u) \in \mathbb{G}$.

Finally, the verifier needs to make sure e_i doesn't alter the value of a_i. This can be done by taking the modulus q of e_i and checking if it returns 0. This is trivial to understand since a'_i is in $\mathbb{Z}_q$. If e_i is a multiple of q then it is obvious that it cannot alter the value of a_i.

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

This test also implies the transformation process explained in this section is sound since the soundness of equation 9 is trivial to prove except for a negligible probability.

For example, if $a'^*_i = a^*_i + x\alpha_i = a_i + \delta + x\alpha_i$. Knowing that e_i must be a multiple of q for equation 10 to be true, we have $a'^*_i \cdot x - e_i = (a_i + x\alpha_i) \cdot x - \omega_i q =$

$x(a_i + x\alpha_i) + x\delta$. In order for equality 8 to be true, the left side of the equality 8 must offset $x\delta$ using committed values T_i and S_i, s.t. $x\delta$ will be removed after applying challenge x to these elements (i.e., $T_i^{x^2} \cdot S_i$, note exponents are different powers of x), which only happens for a negligible chance of $1/q$ when the dishonest prover successfully guessed x correctly.

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 9 when computing $a'_i \cdot x - e_i$. To get around this, the prover simply needs to check if $a_i + (x\ \alpha_i) \bmod q$ overflows q, and subtract $q \cdot x$ from e_i if that's the case.

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

This adjustment does not break the zero-knowledgeness of e_i since $x \in \mathbb{Z}_q$ is significantly smaller than e_i's blinding key $\omega_i \in \mathbb{Z}_p$ ($\frac{q}{p} = 2^{-192}$), 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^{-192}.

The validation part of the input-mapping sub-protocol is defined as follows:

```
$Input : (\vec{P}, \vec{S}, \vec{T} \in \mathbb{G},\ \vec{a}' \in \mathbb{Z}_q; \vec{a}, \vec{\alpha}, \vec{\omega} \in \mathbb{Z}_q,\ \vec{\upsilon} \in \mathbb{Z}_p)$
    $\mathcal{P}'s\ input : (\vec{P},\ \vec{S}, \vec{T}; \vec{a}, \vec{\alpha}, \vec{\omega}, \vec{\upsilon});\ \mathcal{V}'s\ input : (\vec{P},\ \vec{S}, \vec{T})$
    $\mathcal{P}\ \ compute :$
        $e_i = ((x\ \alpha_i \bmod q) - x\ \alpha_i)x + \omega_i\ q; \qquad i = \{1, ..., l\}$
        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 $(e_i \bmod q) \overset{?}{=} 0$, 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.

$$a'_{1_i} - a'_{2_i} = \alpha_i(x_1 - x_2) \tag{12}$$

With $\vec{a}, \vec{\alpha}$ extracted, we can also extract ω from equation 7. Plugging witnesses $\vec{a}, \vec{\alpha}, \vec{v}, \vec{\omega}$ into generators g, h, u, we can re-write the left and right sides of equation 9 to:

$$(h^x/(g^{x^2}u))^{\sum_i^l v_i \cdot k_i} = \prod_{i=1}^{l} \left(\frac{g^{a_i \cdot x} h^{v_i \cdot x}}{g^{a_i' \cdot x - e_i + (v_i - \alpha_i) \cdot x^2 + \omega_i \cdot q} \cdot u^{v_i}} \right)^{k^i} \in \mathbb{G} \tag{13}$$

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

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

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 $a_i \cdot x + x^2 v_i$. Since we know e_i cannot alter $a_i \in \mathbb{Z}_q$ because $e_i \bmod q = 0$ and $a_i' \cdot x = a_i \cdot x + \alpha \cdot x^2$, we can trivially observe that no other value besides a_i in g's exponent in the denominator $((a_i') \cdot x - e_i + (v_i - \alpha_i) \cdot x^2 + (\omega_i \cdot q))$ 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 of $\frac{1}{q}$ (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 transformed inputs $\vec{a}'$, the prover needs to show it knows all coefficients of the output polynomial. For example, for a simple circuit that just outputs the sum of two inputs, the prover needs to show it knows the constant term r and the coefficient of the degree 1 term ϵ of the output polynomial :

$$o = {a_1}' + {a_2}' = r + x \cdot \epsilon \tag{14}$$

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

$$o = {a_1}' \cdot {a_2}' = r + x \cdot \epsilon + x^2 \cdot \tau \tag{15}$$

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 a point X. The degree of the polynomial will increase after each multiplication operation, so the efficiency will drop as the circuit polynomial degree (p_d) increases.

To get the linear polynomial we need 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 term of degree 2 to get the linear polynomial. To do so, the prover commits to τ before the challenge x is known. When the challenge x is available, the prover sends the evaluation value y to the verifier and proofs to prove $f(x) = x^2\tau = y$. The verifier can subtract y from o to get the output in linear polynomial form defined in equation 3:

$$r' = o - y \tag{16}$$

This is a kind of 'bootstrapping'. We call value y a "breaker". Breaker(s) subtracts all "noises" (polynomial terms of degree higher than one) from the raw output o.

In most arithmetic circuits with both addition/subtraction and multiplication/division operations, the polynomial degree is likely smaller than the number of multiplications. This is because every time we add two output wires, their polynomials get merged (e.g. $a_1^7 + a_2 \cdot a_3^6 + a_4^7$ only outputs a polynomial of degree 8, but 21 multiplications are performed). However, there are cases where the p_d value of a circuit exceeds the total number of multiplications of the circuit and therefore significantly degrade the performance of the protocol.

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, 2) a mechanism to efficiently commit to p_d coefficients.

Breaking a large circuit into p_d smaller circuits We can break the circuit we are evaluating into p_d sub-circuits and then batch evaluate p_d circuits at once. So when the polynomial degree of a sub-circuit i reaches degree $b+1$, we can reset it to a linear polynomial of degree 1 by evaluating the sub-circuit and then use its output as an input to the next sub-circuit (Figure 1).

$$\begin{matrix} o_1 \\ o_2 \\ . \\ . \\ o_{p_d} \end{matrix} = \begin{pmatrix} r_1 & \epsilon_1 & \tau_{1,1} & \tau_{1,2} & . & . & \tau_{1,b} \\ r_2 & \epsilon_2 & \tau_{2,1} & \tau_{2,2} & . & . & \tau_{2,b} \\ . & . & . & . & & . & \\ . & . & . & & . & . & \\ r_{p_d} & \epsilon_{p_d} x & \tau_{p_d,1} & \tau_{p_d,2} & . & . & \tau_{p_d,b} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

Figure 1

We evaluate each sub-circuit $i = \{1, ..., p_d\}$ in the same way we evaluate the full circuit. The output of a sub-circuit is also in the linear polynomial form $r'_i = r_i + x \cdot \epsilon_i$ like that of the full circuit 3. Since the output of each sub-circuit is in the same linear polynomial format as inputs to the circuit, they can be reused as inputs to subsequent sub-circuits.

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. Under this setup, the output of each sub-circuit r'_i for $i = \{1, ..., p_d\}$ is computed by subtracting the breaker y_i of the sub-circuit from its raw output o_i.

$$r'_i = o_i - y_i \qquad \text{for} \qquad i = \{1, ..., p_d\} \tag{17}$$

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.

$$\begin{array}{c} o_1 \\ o_2 \\ . \\ . \\ o_{p_d} \end{array} = \underbrace{\begin{pmatrix} r_1 & \epsilon_1 \\ r_2 & \epsilon_2 \\ . & . \\ . & . \\ r_{p_d} & \epsilon_{p_d} \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix}}_{outputs} + \underbrace{\begin{pmatrix} \tau_{1,1} & \tau_{1,2} & . & . & \tau_{1,b} \\ \tau_{2,1} & \tau_{2,2} & . & . & \tau_{2,b} \\ . & . & . & & . \\ . & . & & . & . \\ \tau_{p_d,1} & \tau_{p_d,2} & . & . & \tau_{p_d,b} \end{pmatrix} \begin{pmatrix} x^2 \\ x^3 \\ . \\ . \\ x^{b+1} \end{pmatrix}}_{breakers}$$

Figure 2

By proving the knowledge of all coefficients of each sub-circuit (e.g., the prover and the verifier engage to evaluate the sub-polynomial defined in each row of Figure 2), we know each sub-circuit output r'_i is correct.

It is worth noticing that the prover can set breakers anywhere on the circuit at any time depending on the application. For simplicity, we assume all breaker are set at $b + 1$ degrees, where $b = p_d/p_d$.

Make group exponentiation operations sub-linear to the p_d value Using polynomial evaluation protocol to evaluate each sub-circuit would be expensive. In our case, the verifiers already have the evaluation result o_i of each polynomial (sub-circuit) available using direct computation of inputs, and the coefficients for both constant and degree 1 terms are committed values (output of the sub-circuit). We only need to make sure the prover cannot commit to some altered circuit output (e.g. $r^*_i = r_i - \delta_i$ for some arbitrary δ_i) while the altered breaker ($y'^*_i = y'_i + \delta_i$ s.t. $r^*_i = o_i - y'^*$) can still be computed from "other" coefficients (degree 2 to degree b coefficients) of the sub-circuit.

We use two challenges x, w to allow the prover to "commit" to "other" coefficients of $1, ..., p_d$ polynomials in batches without using expensive ECC operations.

First, we commit to the outputs of sub-circuits. Instead of using Pedersen commitments to commit on each output as we do with the circuit output R, we use two vector commitments to batch commit the outputs of sub-circuits. Since the ECC group we use has an order p, we need to define $r_{p_i} = r_i + x \cdot \epsilon_i \in \mathbb{Z}_p$ s.t. $r_{p_i} \bmod q = r'_i$. The prover will send this value to the verifier.

Since r_{p_i} is a raw number with approximately $2|q|$ bits, anyone can easily extract witnesses r_i, ϵ_i with x. This is why we need a blinding key μ_i that multiplies the order of the smaller group q to create a blinding value. In our benchmarking test, we set μ_i 60-bits longer than q, so it will perfectly hide r_{p_i} except for a negligible chance of 2^{-60} (even in such a case, only insignificant information may be leaked, in which case the prover can randomly select another μ_i).

$$\mu_i \xleftarrow{\$} \mathbb{Z}_m \qquad\qquad i = \{1, ..., l\} \qquad (18)$$

$$r'_{p_i} = r_i + x \cdot \epsilon_i + \mu_i \cdot q \in \mathbb{Z}_p \qquad i = \{1, ..., p_d\} \tag{19}$$

$$R_t = \prod_{i=1}^{p_d} u_i^{r_i + \mu_i \cdot q} \in \mathbb{G}, \qquad E_t = \prod_{i=1}^{p_d} u_i^{\epsilon_i} \in \mathbb{G} \tag{20}$$

Once mod q is applied to r'_{p_i}, this blinding value will disappear in r'_i.

$$r'_i = r_{pi} \bmod q \in \mathbb{Z}_q \qquad i = \{1, ..., p_d\} \tag{21}$$

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 value and then converts them to $\vec{r}\,'$.

$$\prod_{i=1}^{p_d} g_i^{r_i} = \prod_{i=1}^{p_d} R_i \cdot E_t^x \tag{22}$$

The result 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 is computed directly from the evaluation result o_i and the committed output r'_i.

$$y'_i = o_i - r'_i \qquad \text{for} \qquad i = \{1, ..., p_d\} \tag{23}$$

We can observe that if the prover makes the coefficients $\tau_{i,1}, ..., \tau_{i,b}$ "committed" before challenge x is known, then we know $y'_i = \tau_{i,1}x^2 +, ..., +\tau_{i,b}x^{b+1}$ except with negligible probability at most $\frac{q}{b}$ (because of possible existence of polynomial roots). If the breaker is correctly computed, the output r'_i must also be correctly computed.

We use a challenge w that's made available after the prover commits to R_t, E_t and before the challenge x is known to allow the prover to hide and pass all coefficients for each degree in one single element z_j.

$$z_j = \sum_{i=1}^{p_d} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{1, ..., b\} \tag{24}$$

The verifier also verifies $\vec{y}\,'$ as a single element. The update matrix that the verifier now needs to verify is depicted in Figure 3.

$$\begin{matrix} y'_1 w \\ y'_2 w^2 \\ . \\ . \\ y'_{p_d} w^{p_d} \end{matrix} = \begin{pmatrix} \tau_{1,1}w & \tau_{1,2}w & . & . & \tau_{1,b}w \\ \tau_{2,1}w^2 & \tau_{2,2}w^2 & . & . & \tau_{2,b}w^2 \\ . & . & . & & . \\ . & . & & . & . \\ \tau_{p_d,1}w^{p_d} & \tau_{p_d,2}w^{p_d} & . & . & \tau_{p_d,b}w^{p_d} \end{pmatrix} \begin{pmatrix} x^2 \\ x^3 \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

Figure 3

Challenge x is only made available after $\vec{z}$ are known. The verifier validates $\vec{y}\,'$

by multiplying each y'_i by w^i and each z_j by x^{j+2}, their difference must be equal to $0 \in \mathbb{Z}_q$.

$$\sum_{j=1}^{b} z_j \cdot x^{j+1} - \sum_{i}^{p_d} y_i \cdot w^i \bmod q = 0 \tag{25}$$

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.

$$\sum_{j=1}^{b} (z_j + \Delta_j) \cdot x^{j+1} = \sum_{i=1}^{p_d} (y_i + \delta_i) \cdot w^i \in \mathbb{Z}_q \tag{26}$$

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.

$$\sum_{j=1}^{b} \Delta_j \cdot x^{j+1} = \sum_{i=1}^{p_d} \delta_i \cdot w^i \in \mathbb{Z}_q \tag{27}$$

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 $i = \{1, .., p_d\}$, 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 $i = \{1, .., p_d\}$, 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:

$$Input : (\vec{P}, R \in \mathbb{G};\ \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_p) \tag{28}$$

$$\mathcal{P}'s\, input : (\vec{P}, R;\ \vec{a}, \vec{v}, r, \phi);\ \ \mathcal{V}'s\, input : (\vec{P}, R) \tag{29}$$

$$\mathcal{P}\ compute : \tag{30}$$

```
    $\mu_i \xleftarrow{\$} \mathbb{Z}_m, \quad i = \{1, ..., p_d\}$  (31)
    $\alpha_i \xleftarrow{\$} \mathbb{Z}_q, \quad i = \{1, ..., l\}$  (32)
    **for** $i = 1, ..., p_d$ {  (33)
        $r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}', r_i, \epsilon_i, \vec{\tau}_i);$  (34)
        $\epsilon_i = (\epsilon_i - \mu_i) \bmod p$ }  // note $r_{p_d} = r$  (35)
    $R_t = \prod_{i=1}^{p_d} u_i^{r_i + \mu_i \cdot q} \in \mathbb{G}, \qquad E_t = \prod_{i=1}^{p_d} u_i^{\epsilon_i} \in \mathbb{G}$  (36)
    InputMapping-Setup$(; \vec{a} || r_{p_d}, \vec{\alpha} || \epsilon, \vec{v} || \phi) \rightarrow: \vec{S}, \vec{T}$  (37)
$\mathcal{P} \rightarrow \mathcal{V} : R_t, E_t, \vec{S}, \vec{T}$  (38)
$\mathcal{V} \rightarrow \mathcal{P} : \quad w \xleftarrow{\$} \mathbb{Z}_p$  (39)
$\mathcal{P}$ *compute* :  (40)
    $z_j = \sum_{i=1}^{p_d} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \quad j = \{1, ..., b\}$  (41)
$\mathcal{P} \rightarrow \mathcal{V} : \vec{z}$  (42)
$\mathcal{V} \rightarrow \mathcal{P} : \quad x \xleftarrow{\$} \mathbb{Z}_p$  (43)
$\mathcal{P}$ *compute* :  (44)
    $a'_i = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \quad i = \{1, ..., l\}$  (45)
    $r'_{p_i} = r_i + \mu_i \cdot q + x \cdot \epsilon_i \in \mathbb{Z}_p \quad i = \{1, ..., l\}$  (46)
$\mathcal{P} \rightarrow \mathcal{V} : \vec{a}', \vec{r}'_p$  (47)
$\mathcal{V}$ *verify final output* :  (48)
    $r'_i = r'_{p_i} \bmod q \in \mathbb{Z}_q \quad i = \{1, ..., p_d\}$  (49)
    **for** $i = 1, ..., p_d$ {  (50)
        $o_i = computeSubCircuit_i(\vec{a}' || \vec{r}') \in \mathbb{Z}_p$  (51)
        $y'_i = o_i - r'_i \in \mathbb{Z}_q$ }  (52)
    **if** $(\prod_{i=1}^{p_d} g_i^{r_i} \stackrel{?}{=} \prod_{i=1}^{p_d} R_i \cdot E_t^x)$  (53)
     $\wedge \ ((\sum_{j=1}^{b} z_j \cdot x^{j+1} - \sum_{i}^{p_d} y_i \cdot w^i) \bmod q = 0)$ **then** *continue*  (54)
    **else** *reject*  (55)
    **if** InputMapping-Verify$(\vec{P} || R, \vec{S}, \vec{T}, \vec{a}' || r'_{p_d}; \vec{a} || r, \vec{\alpha} || \epsilon, \vec{v} || \phi)$  (56)
        **then** *continue*  (57)
    **else** *reject*  (58)
```

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 $\vec{a}'^*, \vec{r_{p_d}}'^*$ 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 $\mathcal{X}$ extracts the same circuit witnesses from transcripts $\vec{z}$ that still satisfy the relation 25 with transcript $\vec{y}'$. The extractor $\mathcal{X}$ generates $p_d + 1$ challenges $\vec{w}$ and forwards them to the prover. After receiving circuit evaluation transcripts $\vec{\tau}_1$ computed from the first challenge w_1, the extractor rewinds and repeats this step p_d times to generate challenges $w_2, ..., w_{p_d+1}$ and retrieve witnesses $\vec{\tau}_2, ..., \vec{\tau}_{p_d+1}$ from interpolation. Rearrange equality 25 and substitute $\vec{z}$ for $\vec{\tau}$ as specified in 24 we get the following equality:

$$\sum_{j=1}^{b}(\sum_{i=1}^{p_d} \tau_{i,j} w^i) \cdot x^{j+1} = \sum_{i}^{p_d} y'_i \cdot w^i \in \mathbb{Z}_q \qquad (59)$$

Second, 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 TBD will not stand. With $\vec{r}, \vec{\epsilon}$ extracted, the equality 59 is updated to following:

$$\sum_{j=1}^{b}(\sum_{i=1}^{p_d} \tau_{i,j} w^i) \cdot x^{j+1} = \sum_{i}^{p_d}(o_i - (r_i + x \cdot \epsilon_i)) \cdot w^i \in \mathbb{Z}_q \tag{60}$$

Knowing that $\vec{o}$ must be true 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 24 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 60 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 $C = g^c h^v$ 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 $p_d = 100$) 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 $O(p_d)$. However, our design approach allows us to improve the memory consumption cost of our protocol to $O(b+2p_d)$. We only need to do two thing to achieve a memory cost of $O(b+2p_d)$: 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  $i = 1, ..., m$  {
    $r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}');$
    $\epsilon_i = (\epsilon_i - \mu_i) \bmod q \in \mathbb{Z}_q;$
    $\vec{a}' = \vec{a} \,||\, r_i, \quad \vec{\alpha}' = \vec{\alpha} \,||\, \epsilon_i;$
    $z_j = z_j + \sum_{i=1}^{p_d} (\tau_{i,j} \cdot w^i) \in \mathbb{Z}_q \qquad j = \{1, ..., b\};$
}
```

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

4.6 Cost Analysis

The prover runtime of our protocol is dominated by $O(\iota\, p_d \log p_d + l)$ field operations and $O(b + l)$ group exponentiations. We set $b = \sqrt{p_d}$ 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 even smaller than the number of multiplications. This is because every time we perform the "addition" operation on two output wires, two polynomials get merged (e.g. circuit $a_1^8 + a_2^3 \cdot a_3^5 + a_4^7 + a_5^3 = r$ only outputs a polynomial of degree 8, but 27 multiplications are performed). On the other hand, if p_d can be bigger than the total number of gates, 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 p_d number 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).

As we mentioned earlier, every time an "addition or subtraction" operation is performed, the p_d value of two input wires merges into one. e.g., 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 $p_d = 7$. In other words, the more addition/subtraction operations evenly distributed in a circuit, the smaller the p_d value compared to n and gets closer to the value of the multiplication depth of the circuit. The sum-of-products circuit explained earlier has a multiplication depth of 1.

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 $b = p_d = \sqrt{p_d}$ 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. 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 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. 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

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

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 $p_d = 2^{20}$ 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 $O(b + 2p_d)$. Using the memory-efficient implementation mentioned in Section 4.5, we get the results listed in Table 4.

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

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.

However, table 4 is a little bit deceiving because the smaller the b may also lead to a smaller polynomial degree p_d value 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.).

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

Note that there are other techniques for improving prover memory: Commit-and-prove to glue sub-circuits together (Lunar/Eclipse [14] [2]), streaming SNARKs (Gemini [10]). However, the reported constructions of these protocols require

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	$\frac{1}{b}$	≥ 1.43 M	Yes

trusted setup (non-transparent), and the prover runtime cost of these protocols is magnitudes more expensive.

5.3 Boolean Circuit

Our protocol is optimized for arithmetic circuits, and we believe the best way to use our protocol in the real world is to run an arithmetic circuit with embedded range-proof to perform comparison and floating point operations as explained in Section 4.3. We believe this joined approach can eliminate the need to use boolean circuits in most use cases (except for things like proof of hash, which we doubt has any real-world application).

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, Heidelberg, Germany, 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, Heidelberg, Germany, 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, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_2$3
9. Bhadauria, R., Fang, Z., Hazay, C., Venkitasubramaniam, M., Xie, T., Zhang, Y.: Ligero++: A new optimized sublinear IOP. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2025–2038. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3417893
10. Bootle, J., Chiesa, A., Hu, Y., Orrù, M.: Gemini: Elastic SNARKs for diverse environments. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 427–457. Springer, Heidelberg, Germany, Trondheim, Norway (May 30 – Jun 3, 2022). https://doi.org/10.1007/978-3-031-07085-3$_1$5
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243868
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354255
13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_1$6
14. Campanelli, M., Faonio, A., Fiore, D., Querol, A., Rodríguez, H.: Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part III. LNCS, vol. 13092, pp. 3–33. Springer, Heidelberg, Germany, Singapore (Dec 6–10, 2021). https://doi.org/10.1007/978-3-030-92078-4$_1$
15. 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, Heidelberg, Germany, Lyon, France (Apr 23–27, 2023). https://doi.org/10.1007/978-3-031-30617-4$_1$7
16. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.P.: Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 738–768. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-1$_2$6

17. 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, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 23–27, 1998). https://doi.org/10.1007/BFb0055745
18. 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
19. 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`
20. 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, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46803-6_7
21. 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`
22. 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)
23. 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
24. 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, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2018). https://doi.org/10.1007/978-3-319-96878-$0_2$4
25. 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, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45727-$3_1$9
26. 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
27. 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
28. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 239–252. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 1990). https://doi.org/10.1007/0-387-34805-$0_2$2
29. 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

30. 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, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2020). https://doi.org/10.1007/978-3-030-56877-1_25
31. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), `https://eprint.iacr.org/2020/1275`
32. 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`
33. 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
34. 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
35. 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
36. 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
37. 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