Input Transformation Based Efficient Zero-Knowledge Argument System for Arbitrary Circuits with Practical Succinctness

Frank Y.C. Lu

YinYao

Abstract. We introduce an efficient transparent interactive zero knowledge argument system with practical succinctness. Our system converts circuit inputs in Pedersen commitment form to linear polynomials so that the verifier can use standard integer operations to compute and verify the circuit output. The verifier runtime of our protocol is linear to the number of multiplication gates in the path that contains the most multiplications in a circuit (we use symbol d_m to denote its value). However, its practical performance still compares favorably against state-of-the-art transparent zero-knowledge protocols with sub-linear verifier work.
The asymptotic cost of our protocol is $O(d_m \log d_m)$ for prover work, $O(d_m)$ for verifier work, and $O({d_m}^{1/2})$ for communication cost, where d_m stands for the total number of multiplication gates in the path that contains the most multiplications in a circuit (e.g. for a circuit with $n = 2^{20}$ sequential multiplications, $d_m = n$). Specifically, when running a circuit with 2^{20} multiplication gates on a single thread CPU, the prover runtime of our protocol is 1.9 seconds, the verifier runtime is 32 ms and the communication cost is 56 kbs.
In this paper, we will first introduce a base version of our protocol in which the prover work is dominated by $O({d_m}^2)$ field operations. Although field operations are significantly faster than group operations, they become increasingly expensive as d_m value gets large. So in the follow up sections, we will introduce a mechanism to apply number theoretic transformation (NTT) to bring down the prover time to $O(d_m \log d_m)$.
Another added benefit of our protocol is that it does not require a front end encoder to translate NP relation R to some zero-knowledge friendly representation $\hat{R}$ (such as R1CS constraint system) before the relation can be converted to a proof system, making our protocol relatively easy to implement and also easier to use compared to constraint system based protocols.

Keywords: zero knowledge · interactive oracle proofs

1 Introduction

Ever since the discoveries of interactive proofs (IPs) [14] and probabilistically checkable proofs (PCPs) [5] [4] [3] [2] in the late last century, there has been tremendous amount of research in the area of proof systems. More recently, the rise Blockchain and Web3 has finally triggered real-world deployments of zero knowledge systems.

Popular zero knowledge systems are often divided into two phases: the first part, a "front-end" encoder converting a specification of an NP-relation R into a "zero-knowledge friendly" representation $\hat{R}$ (e.g. rank-1 constraint system); and then another "back-end" system converting $\hat{R}$ to a zero-knowledge proof system for R. The encoder based two-phased design has accelerated the development of zero knowledge system applications, but it has added cost of running the encoder to translate circuit logic to constraint system form.

Due to expensive computation during setup time of earlier SNARKs, it has become a significant interest to have the structured reference string (SRS) be constructible in a "universal and updatable" fashion, meaning that the same SRS can be used for statements about all circuits of a certain bounded size. Maller et al. constructed for the first time a universal fully succinct SNARK for circuit satisfiability called Sonic [15]. More recently developed protocols such as PLONK [12], MARLIN [11] are universal fully-succinct SNARK with significantly improved prover run time compared to the fully-succinct Sonic. However, there are two draw backs with these SNARKs: first, most of these universal succinct SNARKs systems require trusted setup; second, the prover run-time of these protocols are very expensive, which is prohibitive for many applications.

Protocols belong to the Goldwasser, Kalai, and Rothblum (GKR) class such as Hyrax [18], Virgo [21]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [13] and Ligero/Ligero++ [1] [8]; memory efficient VLOE protocols such as Wolverine [19], Mac'n'Chess [6], Quicksilver [20], offer efficient prover time that are at least one order of magnitude more efficient than pairing based SNARKs, and many of these protocols do not require trusted setups. However, these protocols are largely ignored by the Blockchain community due to expensive verifier time and high communication cost (hundreds of KBs, or MBs for VLOE protocols) than fully succinct PIOP protocols such as STARK [7], PLONK, MARLIN, and Supersonic [10]. Furthermore, state-of-art GKR protocols such as Virgo has additional dependency on circuit depth where protocol complexity increases and performance significantly degrades as the circuit depth gets longer, making them less attractive to the industry where complex business logics are expected on smart contracts.

NIZKs such as SpartanNIZK [16] and later Lakonia [17] seems to offer a much more balanced approach, where it offers efficient prover time (6-18 seconds) and competitive communication cost for large circuits (2^{20} constraints) and 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 thread CPU.

We aim to create a new transparent zero knowledge protocol designed to handle complex circuits and offers prover time and communication cost comparable to that of Spartan and Lakonia, but with verifier time significantly improved over the current state of the art systems.

1.1 Summary of Contributions

We introduce a new efficient, transparent, interactive zero-knowledge argument system that offers great verifier performance despite not being asymptotically succinct. We achieve this by transforming each committed input parameter of a circuit to some obfuscated scalar value in linear polynomial form, where verifiers can perform normal arithmetic operations (e.g., addition and multiplication) on these values as they do on integers. Since field operations are cheap, the verifier time our protocol compare favorably against state-of-the-art transparent zero knowledge protocols.

Another advantage of our design is that our protocol does not require a"front end" encoder to compile business logic relation R into some zero-knowledge friendly representation $\hat{R}$. This construct makes our protocol relatively easy to implement and also makes it easier for end developers to apply zero knowledge design to real world applications.

The base version (Protocol 1) of our protocol offers prover work of $O({d_m}^2)$ field operations and $O(l)\mathbb{G}$ group exponentiations, where d_m stands for the depth of multiplication gates in a circuit and l stands for the number of inputs to a circuit; the verifier work is $O(n)$ field operations and $O(l)\mathbb{G}$ group exponentiations; and the communication cost is $O(l)\mathbb{G}$ group elements.

Prover runing time will get increasingly expensive as d_m gets large, so in the full version of our protocol (Protocol 2), we introduce a mechanism to apply number theoretic transform (NTT) to bring down the prover time to $O(d_m \log d_m)$ field operations and $O(l)\mathbb{G}$ group exponentiations.

Our protocol is specifically efficient for proving applications with complex business logic (e.g., a smart contract validating complex business trade rules) where circuit depth is high and the number of input parameters is much smaller than the circuit size.

We introduce our protocol in the interactive setting where all verifier challenges are random field elements. In practice, we assume that the Fiat-Shamir heuristic would be applied in order to obtain a non-interactive zero-knowledge argument in the random oracle model.

2 Preliminaries

2.1 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.

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

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 $g_0, ..., g_{n-1}$.

2.2 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. When we say knowledge of an argument, we imply that the argument has witness-extended emulation.

Definition 2. (Interactive Argument) Let's say $(\mathcal{P}, \mathcal{V})$ denotes a pair of PPT interactive algorithms and ***Setup*** denotes a non-interactive setup algorithm that outputs public parameters pp given a security parameter λ that both $\mathcal{P}$ and $\mathcal{V}$ have access to. 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 $\mathcal{A}$, the following properties hold:

- **Perfect Completeness**

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

- **Computational Soundness**

$$Pr\left[\begin{array}{c} \forall w (pp, x, w) \notin \mathcal{R} \wedge \\ \langle \mathcal{A}(pp, x, s), \mathcal{V}(pp, x) \rangle = 1 \end{array} \middle| \begin{array}{c} pp \leftarrow \boldsymbol{Setup}(1^\lambda) \\ (x, s) \leftarrow \mathcal{A}(pp) \end{array} \right] \leq \boldsymbol{negl}(\lambda)$$

- **Public Coin** All messages sent from $\mathcal{V}$ to $\mathcal{P}$ are chosen uniformly at random and independently of $\mathcal{P}$'s messages

Definition 3. (Computational Witness-Extended Emulation) Given a public-coin interactive argument tuple $(\mathbf{Setup}, \mathcal{P}, \mathcal{V})$ and arbitrary prover algorithm $\mathcal{P}^*$, let ***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 ***Setup***. Furthermore, let $\mathcal{E}$ ***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 computational witness-extended emulation if for every deterministic polynomial time $\mathcal{P}$ there exists an expected polynomial time emulator $\mathcal{E}$ such that for all non-uniform polynomial time adversaries $\mathcal{A}$ the following condition holds:

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

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

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 (PSHVZK) argument of knowledge for $\mathcal{R}$ if there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that all pairs of interactive adversaries $\mathcal{A}_1, \mathcal{A}_2$ have the following property for every $(x, w, \sigma) \leftarrow \mathcal{A}_2(pp) \wedge (pp, x, w) \in \mathcal{R}$, where σ stands for verifier's public coin randomness for challenges

$$Pr\left[\mathcal{A}_1(tr)=1 \;\middle|\; \begin{array}{c} pp \leftarrow Setup(1^\lambda), \\ tr \leftarrow \langle \mathcal{P}(pp,x,w), \mathcal{V}(pp,x) \rangle \end{array} \right] =$$

$$Pr\left[\mathcal{A}_1(tr)=1 \;\middle|\; \begin{array}{c} pp \leftarrow Setup(1^\lambda), \\ tr \leftarrow \mathcal{S}(pp,x,\sigma) \end{array} \right]$$

Above property states that 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.

2.3 Polynomial Commitment Function

As in the case of other popular zero knowledge protocols that offer succinct proof size, our protocol uses a polynomial commitment evaluation protocol to construct most of our proof transcript. Our protocol uses a version of the polynomial commitment scheme defined by Bootle. et al. [9] Others have improved the square-root-based polynomial commitments by applying the inner product approach defined by Bunez et. al. and adding support for multilinear polynomials such as Hyrax [18] and Spartan [16]. Similar techniques may be used to improve our implementation in the future to reduce proof size in the expense of longer verifier time. The polynomial commitment function PolyCommitEval is defined as:

- $\boldsymbol{PolyCommitEval}(C, y, x; \vec{\tau}, \phi) \rightarrow boolean$ C is the committed polynomial in $\mathbb{G}$ where $\vec{\tau}$ are its coefficients and ϕ is its blinding key. The function returns a boolean value“åtrue” if the polynomial can be correctly evaluated at point x s.t. $y = f(x)$.

While most polynomial commitment evaluation schemes use constant sized commitment, the commitment of the scheme defined by Bootle. et al. [9] is a set of $d_m^{1/2}$ group elements. In order to not lose generality, we use a single element $C \in \mathbb{G}$ to denote a polynomial commitment in this paper.

2.4 Zero Knowledge Proof of Discrete Logarithm

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

- ***ProveDL***$(g, \phi) \rightarrow tr_\phi$ generates proof transcript tr_ϕ, where ϕ is the witness.
- ***VerifyDL***$(g, s, tr_\phi) \rightarrow b \in \{0, 1\}$ takes a proof transcript tr_ϕ and a pair of group elements with discrete log relation $(g, s \in \mathbb{G} \wedge s = g^\phi)$, 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. We know for a fact that Shnorr's protocol has perfect completeness, special honest verifier zero knowledge, and computational witness-extended emulation.

2.5 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 then generators are denoted by capital letters. e.g., $C = g_1^{a_1} g_2^{a_2} ... g_n^{a_n} h^\phi \in \mathbb{G}$ is a commitment commits 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. We use bold letters to denote generators created during the initialization phase. e.g., generator set $\vec{\boldsymbol{g}}$ is the SRS of our protocol generated during the initialization phase. Greek letters are used to label hidden key values. e.g. ϕ is the blinding key for commitment C on generator $h \in \mathbb{G}$.

We use standard vector notation $\vec{v}$ to denote vectors. i.e. $\vec{a} \in \mathbb{Z}_p^n$ is a list of n integers a_i for $i = \{1, 2, ..., n\}$. $\vec{a} \circ \vec{z} = (a_1 \cdot z_1, ..., a_n \cdot z_n) \in \mathbb{F}^n$ is a Hadamard product of two vectors. $\langle \vec{a} \cdot \vec{z} \rangle = \sum_{i=1}^n a_i \cdot z_i \in \mathbb{Z}_p$ is the inner product of two vectors, and $\vec{a} \cdot z = (a_1 \cdot z, ..., a_n \cdot z) \in \mathbb{Z}_p^n$ is the entry wise multiplication such that every element of the first vector a_i is multiplied by the second integer z.

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

3 Zero Knowledge Argument for Arbitrary Arithmetic Circuit with Practical Succinctness

In this section we will first introduce the base version of our protocol, and then we will present the improved version of the baseline protocol in section 4, which leverages number theoretic transform (NTT) to speed up polynomial multiplication operations.

The prover work of the baseline version of our protocol is $O(l+1)\mathbb{G}$ exponentiations and $O({d_m}^2)\mathbb{F}$ operations, where d_m stands for the depth of multiplication gates, or the path that contains the most multiplication gates in a circuit. Unlike GKR based protocols, layers of additive operations have no impact on the asymptotic performance of our protocol. This also implies that the worst case scenario of prover time of our protocol with n gates is n sequential multiplications. In section 4, we will introduce the full version of our protocol that uses NTT to reduce the asymptotic prover time for field operations to $d_m \log d_m$.

The verifier work is dominated by $O(n)$ field operations and $O({d_m}^{\frac{1}{2}})\mathbb{G}$ group operations for using the univariate polynomial commitment scheme defined by Bootle et al. [9] Although $O(n)\mathbb{F}$ field operations are not technically sub-linear, it is still still fast even for large circuits with 2^{20} multiplication gates in practice, so we therefore claim that the verifier work of our protocol achieves practical succinctness.

The communication cost is dominated by $O(l)$ field elements and $O({d_m}^{\frac{1}{2}})\mathbb{G}$ group elements when using the univariate polynomial commitment scheme based on the one developed by Bootle et al.[9]

3.1 Zero Knowledge Argument with Practical Succinctness

We first define the relation $\mathcal{R}_1$ for the base version of our protocol. For l input parameters, let $\mathcal{C}_\mathbb{F}$ represents the set of arbitrary arithmetic circuits in $\mathbb{F}$, there exists a zero knowledge argument for the relation:

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

Each input parameter is a commitment P_i in $\mathbb{G}$ with committed value a_i and blinding key υ_i. The protocol checks whether r, ϵ of R and C are computed from circuit E using input witnesses $\vec{a}, \vec{\upsilon}$.

The main idea is to transform committed inputs in Perdersen commitment form in $\mathbb{G}$ to linear polynomials in $\mathbb{F}$ so that both the prover and the verifier can perform addition and multiplication operations just as they add and multiply polynomials in $\mathbb{F}$. 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 use the same value pair to create its corresponding integer value in linear polynomial form $a_i{}' \in \mathbb{Z}_p$:

$$a_i{}' = a_i + X\upsilon_i \in \mathbb{Z}_p \tag{2}$$

The linear polynomial a_i' is obviously not binding to a_i as we can easily manipulate the value of a_i by altering its blinding key if the value of challenge X is known. e.g. $a_i' = (a_i + \delta) + x(\upsilon_i - \delta/x)$ (the "committed" value a_i is altered to $a_i + \delta$). To combat this, verifiers use the following equation to confirm the mapping between the scalar $a_i{}'$ and P_i for some challenge x:

$$P_i / g^{a_i{}'} = \frac{g^{a_i} h^{\upsilon_i}}{g^{a_i + x\upsilon_i}} = (h/g^x)^{\upsilon_i} \tag{3}$$

If the prover can prove the knowledge of υ_i on generator $(h/g^x) \in \mathbb{G}$ using any proof of knowledge protocol, we have a concrete proof that the witnesses of $a_i{}'$ and P_i must match for some challenge x except for a negligible probability.

Before we move on, there are two issues in real world applications that we have to consider before we construct the base version of our protocol:

First, since each committed P_i may be used multiple times as inputs to different circuits, an attacker can easily deduce the witness pair of P_i from two different challenges x_1, and x_2. (e.g. for $P_i = g^{a_i} h^{\upsilon_i}$, its linear polynomial form value from the first challenge x_1 would be $a_i + x_1\upsilon_i$, and from the second challenge x_2 would be $a_i + x_2\upsilon_i$, subtracting the two will get $\upsilon_i(x_1 - x_2)$ where an attacker can trivially extract the witness pair υ_i and a_i).

Second, for a circuit with l input parameters, it would be pretty inefficient to create l proof of knowledge transcripts for all l inputs, so we need some kind of batching mechanism to verify them in batch.

The prover can get around the first issue by creating new blinding keys α_i for every linear polynomial $a_i{}'$ s.t. $i = \{1, ..., l\}$, and then commit to the differences between each blinding key pair (e.g. $\alpha_i - \upsilon_i$). To batch verify them, the verifier use a random challenge k to generate l challenges $\mathbf{k} = k^1, ..., k^l$ s.t.:

$$\kappa = \sum_{i=1}^{l} (\alpha_i - \upsilon_i) k^i \in \mathbb{Z}_p \tag{4}$$

$$PK_\kappa = h^\kappa \in \mathbb{G} \tag{5}$$

$$tr_\kappa = ProveDL(h, \kappa) \tag{6}$$

If the prover can prove the knowledge of κ on generator $h \in \mathbb{G}$ using any proof of knowledge (or proof of discrete logarithm) protocol, we know that only the sum of products of blinding keys (exponent of h, not the committed values on generator g) are updated after performing equation 7 except for a negligible probability.

$$P_t = \left(\prod_{i=1}^{l} P_i^{k^i}\right) \cdot PK_\kappa \in \mathbb{G} \tag{7}$$

The prover use the new blinding keys $\vec{\alpha}$ to create linear polynomials such that $a_i{}' = a_i + x\alpha_i$ for all i. After challenge k is known, the prover can batch prove the mapping between committed values and their linear polynomial values by providing transcripts for tr_{α_t}.

$$\alpha_t = \sum_{i=1}^{l} (\alpha_i) k^i \in \mathbb{Z}_p \tag{8}$$

$$tr_{\alpha_t} = ProveDL((h/g^x), \alpha_t) \tag{9}$$

The verifier computes the sum of products of $\vec{a}$ and powers of k. With sum of products of both $\vec{P}, \vec{a'}$ (P_t, t) available, the verifier can trivially compute PK_{α_t} s.t.:

$$t = \sum_{i=1}^{l} a_i' k^i \in \mathbb{Z}_p \tag{10}$$

$$PK_{\alpha_t} = P_t / g^t = (h/g^x)^{\alpha_t} \tag{11}$$

If the prover can prove the knowledge of α_t on generator $(h/g^x) \in \mathbb{G}$ using any proof of knowledge protocol, we know that the mapping between $\vec{P}$ and $\vec{a'}$ is correct except for a negligible probability.

The main philosophy of our protocol is that once we have transformed input parameters in linear polynomials, verifiers can just perform arithmetic operations on polynomials, which they can't do on Pedersen commitments. To provide zero knowledge proof for a circuit, the prover needs to to prove it knows all coefficients of the result polynomial after finish running the circuit. For example, adding two input values in $a_i{}'$ is the same as adding two polynomials:

$$o = a_1{}' + a_2{}' = r + X \cdot \epsilon \tag{12}$$

Where $r = (a_1 + a_2)$ and the blinding key is $\epsilon = (\alpha_1 + \alpha_2)$. Multiplying two values in $a_i{}'$ is the same as multiplying two polynomials:

$$o = a_1 \cdot a_2 = r + X \cdot \epsilon + X^2 \cdot \tau \tag{13}$$

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 result polynomial after all circuit operations because the degree of the polynomial will increase after each multiplication operation, and we still want to use the linear polynomial $r' = r + X\epsilon$ to represent the result of the circuit execution that maps to the commitment $R = g^r h^\epsilon$.

To do that, we need to subtract out all terms with degree higher than one. In the simple case above, the verifier needs to eliminate the term with coefficient τ and the prover needs to commit to τ before the challenge x is known, so that verifiers can obtain r' by subtracting $y = X^2\tau$ from o s.t.:

$$r' = o - y \tag{14}$$

If we let d_m to denote the multiplication depth of a circuit, then o will be a polynomial with degree $d_m + 2$. The constant term is the committed value r and the coefficient of degree 1 term is the blinding key ϵ. The prover uses vector commitment with blinding key ϕ to commit to all coefficients with degree term higher than two. In order to not lose generality, we use standard vector commitment notation to denote polynomial commitment. In practice, the polynomial commitment we used is actually a set of $d_m{}^{1/2}$ vector commitments (see Bootle. et al. [9]).

$$C = \prod_{i=1}^{d_m} u_i^{\tau_i} \cdot h^\phi \in \mathbb{G} \tag{15}$$

We now define the function Multiply just to show how coefficients are computed from a circuit multiplication operation, we can trivially observe the logic is the same as that of a polynomial multiplication operation. The output of the function Multiply is an array list $o[]$ of size $d_m + 2$, such that $o[1]$ is the value of r, $o[2]$ is the blinding key ϵ, and $o[3], ..., o[d_m + 2]$ are values of $\vec{\tau}$.

The returned list $o[]$ may be used again as input to another multiplication or addition function.

```
$Input : (lists\ o_l[], o_r[])$                                          (16)
    $Define\ a\ list\ o[]\ of\ size\ n = |o_r| + |o_l - 1|$              (17)
    $Define\ a\ matrix\ o_t[][]\ of\ size\ |o_r| \times |o_l|$           (18)
    for $j = 1, ...,$ size $o_r$                                         (19)
        for $i = 1, ..., |o_l|$                                          (20)
            $o_t[j][i] = o_l[j] \cdot o_r[j]$                            (21)
    for $j, ..., |o_r|$                                                  (22)
        for $i = 1, ..., |o_l|$                                          (23)
            $o[i + j - 1] = o[i + j - 1] + o_t[j][i]$                    (24)
    return $o[]$                                                         (25)
```

Function Multiply

Like that of function Multiply, function Add is polynomial addition operation and outputs the coefficients of result polynomial in array list $o[]$.

```
$Input : (lists\ o_l[], o_r[])$                                          (26)
    $Define\ a\ list\ o[]\ of\ size\ n =$ max length of $o_r, o_l$       (27)
    for $i = 1, ..., n$                                                  (28)
        $o[i] = o_l[i] + o_r[i]$                                         (29)
    return $o[]$                                                         (30)
```

Function Add

We define two more functions for our protocol. function ***computeKeys*** is used by the prover to compute keys of a polynomial, and function ***computeEquation*** is used by verifiers to compute the value of the result polynomial at evaluation point X:
1. function ***computeKeys***(circuit, "input values", "input keys") take input values $\vec{a}$ and keys $\vec{v}$ to compute $r, \epsilon, \vec{\tau}$ (coefficients of o) using the circuit provided to the protocol. function computeKeys uses function Multiply and function Add defined above to compute coefficients of o.
2. function ***computeEquation***(circuit, "input values in linear polynomial form") trivially compute the result o from the inputs provided as they are integer values.

Since the logic of these functions are trivial, we don't waste space describing them in detail here. With all the information available, we now formally introduce Protocol 1:

$$Input : (\vec{P} \in \mathbb{G}^n, g, h, u \in \mathbb{G}, \vec{a} \in \mathbb{Z}_p^l, \upsilon \in \mathbb{Z}_p) \tag{31}$$

$$\mathcal{P}'s\ input : (\vec{P}, g, h, u; \vec{a}, \upsilon) \tag{32}$$

$$\mathcal{V}'s\ input : (\vec{P}, g, h, u) \tag{33}$$

$$\alpha_i \xleftarrow{\$} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{34}$$

$$r, \varepsilon, \vec{\tau} = computeKeys(equation, \vec{a}, \vec{\alpha}) \in \mathbb{Z}_p^{n+2} \tag{35}$$

$$R = g^r h^\varepsilon \in \mathbb{G} \tag{36}$$

$$tr_\epsilon = ((h/g^x), \epsilon) \tag{37}$$

$$\phi \xleftarrow{\$} \mathbb{Z}_p \tag{38}$$

$$\begin{aligned}
&\quad C = \prod_{i=1}^{n} u_i^{\tau_i} \cdot h^{\phi} \in \mathbb{G} && (39)\\
&\mathcal{P} \to \mathcal{V} : C, R, tr_{\epsilon} && (40)\\
&\mathcal{V}\ compute : && (41)\\
&\quad x \overset{\$}{\leftarrow} \mathbb{Z}_p && (42)\\
&\mathcal{V} \to \mathcal{P} : x && (43)\\
&\mathcal{P}\ compute : && (44)\\
&\quad a'_i = a_i + x\alpha_i \in \mathbb{Z}_p \qquad i = \{1, ..., l\} && (45)\\
&\quad y = \sum_{i}^{n} \tau_i \cdot x^{i+1} \in \mathbb{Z}_p && (46)\\
&\mathcal{P} \to \mathcal{V} :\ \vec{a}\,', y && (47)\\
&\mathcal{V}\ verify\ final\ output\ R : && (48)\\
&\quad o = computeEquation(equation, \vec{a}\,') \in \mathbb{Z}_p && (49)\\
&\quad r' = o - y \in \mathbb{G} \quad //\ r' = r + x \cdot \epsilon && (50)\\
&\quad PK_{\epsilon} = R/g^{r'} \in \mathbb{G} \quad //equal\ to\ (h/g^x)^{\epsilon} && (51)\\
&\quad \textbf{if}\ PolyCommitEval(C, y, x; \vec{\tau}, \phi),\ \textbf{then}\ continue && (52)\\
&\quad \textbf{else}\ reject && (53)\\
&\quad \textbf{if}\ VerifyDL((h/g^x), PK_{\epsilon}, tr_{\epsilon}),\ \textbf{then}\ continue && (54)\\
&\quad \textbf{else}\ reject && (55)\\
&\mathcal{V}\ compute : && (56)\\
&\quad k \overset{\$}{\leftarrow} \mathbb{Z}_p && (57)\\
&\mathcal{V} \to \mathcal{P} : k && (58)\\
&\mathcal{P}\ compute : && (59)\\
&\quad \alpha_t = \sum_{i=1}^{l} \alpha_i k^i \in \mathbb{Z}_p && (60)\\
&\quad tr_{\alpha_t} = ProveDL((h/g^x), \alpha_t) && (61)\\
&\quad \kappa = \sum_{i=1}^{l} (\alpha_i - \upsilon_i) k^i \in \mathbb{Z}_p && (62)\\
&\quad PK_{\kappa} = h^{\kappa} \in \mathbb{G} && (63)\\
&\quad tr_{\kappa} = ProveDL(h, \kappa) && (64)\\
&\mathcal{P} \to \mathcal{V} : PK_{\kappa}, tr_{\kappa}, tr_{\alpha_t} && (65)\\
&\mathcal{V}\ verify\ inputs : && (66)\\
&\quad P_t = (\prod_{i=1}^{l} P_i^{k^i}) \cdot PK_{\kappa} \in \mathbb{G} && (67)\\
&\quad t = \sum_{i=1}^{l} a'_i k^i \in \mathbb{Z}_p && (68)\\
&\quad PK_{\alpha_t} = P_t/g^t && (69)\\
&\quad \textbf{if}\ VerifyDL(h, PK_{\kappa}, tr_{\kappa}), && (70)
\end{aligned}$$

$$\textbf{and } VerifyDL((h/g^x), PK_{\alpha_t}, tr_{\alpha_t}), \textbf{ then } accept \tag{71}$$
$$\textbf{else } reject \tag{72}$$

Protocol 1

Theorem 1. *(Zero Knowledge Argument for Arbitrary Circuits with Practical Succinctness). The proof system presented in this section has perfect completeness, perfect special honest verifier zero-knowledge, and computational witness extended emulation.*

The proof for Theorem 1 is presented in Appendix A.

The main idea of Protocol 1 is to convert commitments P_i to its linear polynomial form a'_i so that the verifier can just take linear polynomials as input values to the circuit and use standard integer operations to compute the circuit. The computation result o is a polynomial with d_m+1 degree, where d_m is the number of multiplications in the path that has the maximum number of multiplications in a circuit. By subtracting out all term with degree greater than 2 as in equation 14 explained, the verifier will get the scalar value in linear polynomial form that maps to commitment R.

4 Efficient Zero Knowledge Argument for Arbitrary Circuits with Practical Succinctness

Protocol 1's prover isn't efficient because $O({d_m}^2)$ field operations in prover work can become expensive as d_m gets big. In this section, we introduce a mechanism that allows us to use the number theoretic transform (NTT) to cut prover's field operation work to d_m log d_m.

4.1 Using Number Theoretic Transform to Improve Prover Performance in Field Operations

The objective of NTT is to multiply two polynomials such that the coefficients of the resultant polynomials are calculated under a particular modulo in d_m log d_m, a major improvement over ${d_m}^2$ runtime in protocol 1. However, a major drawback of NTT is that it requires a prime modulo q of the form $q = c \cdot 2^k + 1$ to be the order of the group, where k and c are arbitrary constants. Since the order of widely used $\mathbb{G}$ in cryptography is usually not a prime with the aforementioned form, we need a mechanism to map linear polynomials with a prime modulo q that satisfies the aforementioned form to any group with prime order p. q is expected to be smaller than p because: 1) computation in p (e.g. polynomial commitment evaluation) won't overflow 2) lower communication cost. In our benchmark testing, we set q to a 62-bit number.

We redefine equation 2 s.t. a'_i and its blinding key α_i are now in field q instead of the larger field p.

$$a'_i = a_i + x\alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{73}$$

We define a new blinding key $\omega_i \in \mathbb{Z}_p$ and mix that with blinding key υ_i in P_i and its matching α_i in a' to create S_i, T_i.

$$S_i = g^{\omega_i \cdot q} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{74}$$
$$T_i = g^{\upsilon_i - \alpha_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{75}$$

The prover then sends S_i, T_i for $i = \{1, .., l\}$ to the verifier. When the challenge $x \in \mathbb{Z}_q$ is available, the prover sends e_i s.t.:

$$e_i = ((x\alpha_i \bmod q) - x\alpha_i) \cdot x + \omega_i \cdot q \qquad i = \{1, ..., l\} \tag{76}$$

e_i is not in $\mathbb{Z}_p$, but it is a good idea to keep e_i smaller than p to keep the communication cost low. The idea here is that when we subtract e_i from a_i, we can subtract out the blinding modulo q element ($x\alpha_i \bmod q$) from a'_i (e.g. $a'_i \cdot x - e_i = (a_i + x\alpha_i) \cdot x - \omega_i q$). The verifier can replace $x^2\alpha_i - \omega_i q$ part with the new blinding element $x^2 \upsilon_i$ as the exponent of generator g by adding the previously committed values S_i, T_i.

$$g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i = (g^x)^{a_i + x\upsilon_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{77}$$

With $(g^x)^{a_i + x\upsilon_i}$ available, the verifier can trivially divide each P_i and taking their sum with powers of k to get PK_{υ_t}.

$$PK_{\upsilon_t} = \prod_{i=1}^{l} \left(\frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{78}$$

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

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.

$$\textbf{if } (e_i \bmod q) == 0, \textbf{ then } \textit{continue} \tag{79}$$

e_i does not leak any information to the verifier either. This is because the first part of e_i: $((x\alpha_i \bmod q) - x\alpha_i) \cdot x$ is a multiple of q, which is also equivalent to $s \cdot q$ for some s. s value is perfectly hiding with the blinding term $w \cdot q$ in e_i.

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 77 78 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 subtracts $q \cdot x$ from e_i if that's the case.

$$\textbf{if } a_i + (x\alpha_i \bmod q) > q, \textbf{ then } e_i = e_i - q \cdot x \qquad i = \{1, ..., l\} \tag{80}$$

We now merge the NTT conversion code introduced in this section and formally define the efficient version of our protocol in Protocol 2.

$$Input : (\vec{P} \in \mathbb{G}^n, g, h, u \in \mathbb{G}, \vec{a} \in \mathbb{Z}_p^l, \upsilon \in \mathbb{Z}_{p-q}) \tag{81}$$

$$\mathcal{P}'s\,input : (\vec{P}, g, h, u; \vec{a}, \upsilon) \tag{82}$$

$$\mathcal{V}'s\,input : (\vec{P}, g, h, u) \tag{83}$$

$$\alpha_i \xleftarrow{\$} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{84}$$

$$\omega_i \xleftarrow{\$} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{85}$$

$$r, \varepsilon, \vec{\tau} = computeKeys(equation, \vec{a}, \vec{\alpha}) \in \mathbb{Z}_p^{n+2} \tag{86}$$

$$R = g^r h^\varepsilon \in \mathbb{G} \tag{87}$$

$$tr_\epsilon = ((h/g^x), \epsilon) \tag{88}$$

$$\phi \xleftarrow{\$} \mathbb{Z}_p \tag{89}$$

$$C = \prod_{i=1}^{n} g_i^{\tau_i} \cdot h^\phi \in \mathbb{G} \tag{90}$$

$$S_i = g^{\omega_i \cdot q} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{91}$$

$$T_i = g^{\upsilon_i - \alpha_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{92}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}, C, R, tr_\epsilon \tag{93}$$

$$\mathcal{V}\ compute: \tag{94}$$

$$x \xleftarrow{\$} \mathbb{Z}_p \tag{95}$$

$$\mathcal{V} \rightarrow \mathcal{P} : x \tag{96}$$

$$\mathcal{P}\ compute: \tag{97}$$

$$a'_i = a_i + x\alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{98}$$

$$e_i = ((x\alpha_i \bmod q) - x\alpha_i)x + \omega_i q \in \mathbb{Z}_p \qquad i = \{1, ..., l\} \tag{99}$$

$$\textbf{if } a_i + (x\alpha_i \bmod q) > q, \textbf{ then } e_i = e_i - q \cdot x \qquad i = \{1, ..., l\} \tag{100}$$

$$y = \sum_i^n \tau_i \cdot x^{i+1} \in \mathbb{Z}_p \tag{101}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}', y \tag{102}$$

$$\mathcal{V}\ verify\ final\ output: \tag{103}$$

$$o = computeEquation(equation, \vec{a}') \in \mathbb{Z}_p \tag{104}$$

$$r' = o - y \in \mathbb{Z}_p \quad // \ r + x \cdot \epsilon \tag{105}$$

$$PK_\epsilon = R/g^{r'} \in \mathbb{G} \quad //equal\ to\ (h/g^x)^\epsilon \tag{106}$$

$$\textbf{if } PolyCommitEval(C, y, x; \vec{\tau}, \phi), \textbf{ then } continue \tag{107}$$

$$\textbf{else } reject \tag{108}$$

$$\textbf{if } VerifyDL((h/g^x), PK_\epsilon, tr_\epsilon), \textbf{ then } continue \tag{109}$$

$$\textbf{else } reject \tag{110}$$

$$\mathcal{V}\ compute: \tag{111}$$

$$k \xleftarrow{\$} \mathbb{Z}_p \tag{112}$$

$$\mathcal{V} \rightarrow \mathcal{P} : k \tag{113}$$

$$\mathcal{P}\ compute: \tag{114}$$

$$\upsilon_t = \sum_{i=1}^l \upsilon_i k^i \in \mathbb{Z}_p \tag{115}$$

$$tr_{\upsilon_t} = ProveDL(h/g^x, \upsilon_t) \tag{116}$$

$$\mathcal{P} \rightarrow \mathcal{V} : tr_{\upsilon_t} \tag{117}$$

$$\mathcal{V}\ verify\ inputs: \tag{118}$$

$$\textbf{if } (e_i \bmod q) == 0, \textbf{ then } continue \tag{119}$$

$$\textbf{else } reject \tag{120}$$

$$PK_{\upsilon_t} = \prod_{i=1}^l \left(\frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{121}$$

$$\textbf{if } VerifyDL((h^x/g^x), PK_{\upsilon_t}, tr_{\upsilon_t}), \textbf{ then } accept \tag{122}$$

$$\textbf{else } reject \tag{123}$$

Protocol 2

Theorem 2. *(Efficient Zero Knowledge Protocol for Arbitrary Circuit with Practical Succinctness). The proof system presented in this section has perfect completeness, perfect special honest verifier zero-knowledge, and computational witness extended emulation.*

The proof for Theorem 2 is presented in Appendix B.

Since we are now evaluating the output polynomial at a smaller field q, the soundness error is increased due to existence of polynomial roots in the smaller field. The NTT acceptable prime we use in our test case in $q = 1945555039024054273$, where $r = 27, k = 56, root = 5$ s.t. $q = r * 2^k + 1$. For a circuit with $2^2 0$ sequential multiplications ($d_m = n$), the soundness error is at most 2^{-41}, acceptable in most real-life use cases.

4.2 Booleanity Check and Bit Decomposition/Reposition

A common occurrence in proof systems is the need to enforce input data $a_i \in \{0,1\}$ 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, like the int type in Java) in order to perform comparison operations on input data. If a committed value a_i is in $[0,1]$, then its linear polynomial form a_i must have the following property:

$$(a'_i \cdot a'_i - a'_i) = \beta_1 x + \beta_2 x^2 \tag{124}$$

Where $\beta_2 = \alpha^2$, and $\beta_1 = \alpha$ when a_j is 1 and $\beta_1 = -\alpha$ when a_j is 0. To prove the correctness for all $a_i \in \{0,1\}$, the prover commits to two polynomials K_1, K_2 s.t.

$$K_1 = g_1^{\beta_{1_1}} g_2^{\beta_{1_2}} ... g_l^{\beta_{1_l}} h^{\rho_1} \qquad and \qquad K_2 = g_1^{\beta_{2_1}} g_2^{\beta_{2_2}} ... g_l^{\beta_{2_l}} h^{\rho_2} \tag{125}$$

Where K_1 commits to coefficients on x term for $i \in \{1, ..., l \cdot 32\}$ and K_2 commits to coefficients on x^2 term for $i \in \{1, ..., l \cdot 32\}$. The prover sends K_1, K_2 to the verifier. When the challenge k is received, the prover sends the evaluation results y_1, y_2 to the verifier, and the verifier uses the polynomial commitment protocol to verify the correctness of y_1, y_2 at point k, and checks if the equality below is true:

$$y_1 \cdot x + y_2 \cdot x^2 = \sum_{j=1}^{l \cdot 32} (a'_i \cdot a'_i - a'_i) \cdot k^i \tag{126}$$

Once we know all linear polynomials maps 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 $a'_{i,j}$ for $j = \{1, ..., 32\}$.

$$a'_i = \sum_{j=1}^{32} a'_{i,j} \cdot 2^j \tag{127}$$

In practice, we will conduct booleanity test on all $l \cdot 32$ bit values at once and then use equation 127 to convert them to l full integer values so that we can perform the "linear polynomial to Pedersen commitment" mapping test explained in the last two sections.

5 Performance Comparison

We compare the performance of our protocol to some of the most popular transparent Zero Knowledge Protocols that open source codes are available. Our test runs are performed on Intel(R) Core(TM) i7-9750H CPU @ 2.60 Ghz. Only one core is being utilized, and all tests are run on a single CPU thread.

The baseline protocols we picked are Hyrax, Ligero, Aurora, and Spartan-NIZK. These protocols are 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 time, verifier time, proof size), and Spartan offers the most balanced performance across all performance parameters.

We didn't consider transparent protocols that highly depend on circuit depth such as GKR based protocols simply because they can't handle 2^{20} sequential multiplications. We also don't consider VOLE based protocols as they are only optimized for prover work. Other popular transparent schemes such as Bulletproofs are also not being considered because they have linear verifier time and therefore are not succinct.

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 inputs to 20 integers, and each input is represented by 32 bits so that there are a total of $20 \cdot 32 = 640$ input bits to the circuit. The performance of our protocol correlates with the depth of multiplication gates, so we set the multiplication depth of our protocol to $d_m = \frac{n}{32}$ (when $n = 2^{20}$, $d_m = 2^{16}$) and $d_m = n$. When we use 32 bits to represent each input to a circuit, its multiplication depth is at most $\frac{n}{32}$ for sequential multiplications, so the $d_m = n$ case is almost impossible to exist in a reasonably constructed circuit, we only show that for bench marking purpose.

The prime number q we picked for our benchmark testing is 1945555039024054273, a 61-bit number that implies the soundness error will be at most 2^{-41} for a circuit with 2^{20} multiplications and $d_m = n$, and at most 2^{-46} for a similar sized circuit and that $d_m = \frac{n}{32}$), either way more than enough in most real-life applications.

To maximize the advantage of the NTT algorithm in computing sequential multiplications, we arrange our circuit in binary tree format. Such tuning may not be required in real-world applications since large circuits should have multiplication gates somewhat balanced out across layers.

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
This Work ($d_m = \frac{n}{32}$)	0.6	0.7	1	1.1	1.4	1.8
This Work ($d_m = n$)	0.6	0.8	1	1.5	2.7	6

Table 1: Prover time comparison (KBs)

Table 1 shows that as the circuit size gets bigger, the prover performance of our protocol is becoming increasingly more efficient than all of our baseline protocols. SpartanNIZK seems to match that of our for circuits with 2^{20} constraints (which will also diminish as the circuit grows bigger) when compared to the worst case scenario version of our protocol ($d_m = n$) . However, this is not a fair comparison in our favor since we're comparing 2^{20} constraints in SpartanNIZK with the unlikely scenario of 2^{20} sequential multiplications in that of our protocol. Furthermore, our protocol doesn't use a constraint system; savings from eliminating copy constraints and encoder for a constraint system will further boost the actual performance of our protocol

Table 2 shows that the communication cost of our protocol is more efficient than all baseline protocols as we approach $n = 2^{20}$ when $d_m = \frac{n}{32}$. Like that of prover performance, SpartanNIZK holds a small advantage in the unlikely case where $d_m = n$. For higher input counts, see Table 4 for more detail.

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($d_m = \frac{n}{32}$)	11	12	13	15	18	24
This Work($d_m = n$)	13	14	17	24	36	62

Table 2: Proof size comparison (KBs)

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($d_m = \frac{n}{32}$)	7	8	8	9	11	16
This Work($d_m = n$)	8	9	10	13	17	25

Table 3: Verifier time comparison (ms)

We can observe from table 1-2 that our protocol is at least as competitive as the current state of art in prover runtime and verifier runtime. Table 3 demonstrates that our protocol achieves significant improvement by at least one order of magnitude in verifier runtime over all baseline protocols we are comparing against. Like that of communication cost, the verifier time of our protocol will grow when the number of inputs to the protocol grows.

One may consider 20 integer inputs and 640 input bits to a circuit too small, so in table 4 we list performance benchmarks for different number of inputs (l) to a circuit with 2^{20} multiplications. To better mock real-life scenarios, we decompose each committed input value to 32 bits, so $d_m = \frac{n}{32}$.

Input bits ($l \cdot 32$)	Input Integers (l)	Prover time(s)	Verifier time(ms)	Proof size(kb)
960	30	1.8	15	28
1,280	40	1.8	17	32
1,600	50	1.8	20	36
1,920	60	1.9	22	40
2,240	70	1.9	23	44
2,560	80	1.9	25	48
2,880	90	1.9	29	52
3,200	100	1.9	32	56

Table 4: Performance comparison for different input numbers on circuits with 2^{20} multiplications and $d_m = \frac{n}{32}$

In table 4 we can observe that increases in prover time and verifier time are hardly noticeable as the input bits count approaching $3,200$. This is because the total input number is still small compared to the size of the circuit with 2^{20} sequential multiplications. Communication cost gets impacted the most as the input count gets higher. This is because the prover have to send $l \cdot 32$ linear polynomials to the verifier. Technically speaking, more inputs usually implies lower circuit depth and less complex business logic.

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. 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
3. 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
4. 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
5. 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
6. 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
7. 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
8. 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
9. 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, Heidelberg, Germany, Vienna, Austria (May 8–12, 2016). https://doi.org/10.1007/978-3-662-49896-$5_1$2
10. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 677–706. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-$1_2$4
11. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., 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
12. 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`
13. 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)
14. 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
15. 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
16. 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_2$5
17. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), `https://eprint.iacr.org/2020/1275`

18. 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`
19. 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
20. 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
21. 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

Appendiex

A. Proof for Theorem One

Proof. Perfect completeness follows from the fact that Protocol 1 is trivially complete. To prove perfect honest-verifier zero-knowledge, we define a simulator $\mathcal{S}$ to show that protocol 1 has perfect special honest verifier zero-knowledge for relation 1. $\mathcal{S}$ uses simulator $\mathcal{S}_S$ to simulate proof transcripts for proof of knowledge (or proof of discrete logarithm) protocols, and simulator $\mathcal{S}_p$ to simulate proof transcripts for polynomial commitment evaluation function PolyCommitEval.

Simulator $\mathcal{S}$ generates random group elements for C, R, proof of knowledge transcript tr_ϵ. After receiving challenge x from the verifier, the simulator generates l random integers to represent linear polynomials $\vec{a}'$ and one random integer to represent y and sends them to the verifier.

The verifier follows the protocol to compute PK_ϵ, then simulator $\mathcal{S}$ calls simulator $\mathcal{S}_S$ to interact with the verifier and generate all necessary transcripts to prove it knows the value of ϵ. This makes sense since we already know for a fact that schnorr and many other proof of knowledge protocols have perfect special honest verifier zero-knowledge. Similarly, the simulator $\mathcal{S}$ calls simulator $\mathcal{S}_P$ to simulate the transcripts for proving y is the evaluated value at point x for polynomial commitment C.

The simulator then simulates the transcripts to prove it knows α_t and κ. The simulator simply sends randomly generated PK_κ and random transcripts for tr_κ and tr_{α_t}, and calls simulator $\mathcal{S}_S$ to simulate transcripts needed to prove the knowledge of κ and α_t.

Simulator $\mathcal{S}$ chooses all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or computes 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 1 has perfect special honest verifier zero-knowledge.

To prove computational witness extended emulation, we construct an extractor $\mathcal{X}$, which uses extractor $\mathcal{X}$ to extract witnesses from proof of knowledge transcripts and extractor $\mathcal{X}$ to extract witnesses from polynomial commitments.

First we show how to construct an extractor $\mathcal{X}$ for Protocol 1 s.t. on input $\vec{P} \in \mathbb{G}^l, R \in \mathbb{G}$, it either extracts witnesses $r, \vec{a}, \epsilon$ for relation 1 , or discovers a non-trivial discrete logarithm relation between $g, h \in \mathbb{G}$.

To begin, extractor $\mathcal{X}$ interacts with the prover in the same way as any verifier would and receives $C,\ R \in \mathbb{G}, tr_\epsilon$ from the prover.

Extractor $\mathcal{X}$ then generates a challenge x_1 and forwards it to the prover. After receiving $\vec{a}'_1, y_1$, the extractor rewinds the prover and sends another challenge x_2 to retrieve $\vec{a}'_2, y_2$.

The extractor then follows the protocol and computes o and PK_ϵ, then calls extractor $\mathcal{X}_S$ to extract ϵ from tr_ϵ and PK_ϵ. With either x_1 or x_2, we can trivially retrieve r from r' since $r' = r + x \cdot \epsilon$, and validate if $R = g^r h^\epsilon$.

To validate if r, ϵ is correctly computed from the circuit, extractor $\mathcal{X}$ calls extractor $\mathcal{X}_P$ to retrieve set $\vec{\tau}$ from polynomial commitment C. With $\vec{\tau}$, we can trivially validate the correctness r, ϵ from o at any evaluation point x since:

$$o = r + \epsilon \cdot x + \sum_{i=1}^{n} \tau_i \cdot x^{i+1}$$

We have now retrieved witnesses $r, \epsilon, \vec{\tau}$ using the prover committed values C, R, tr_ϵ. If the prover is honest, $r, \epsilon, \vec{\tau}$ must be computed by the prover from witnesses $\vec{a}, \vec{\alpha}$. We now go back to check if that's the case here. With $\vec{a}'_1$ and $\vec{a}'_2$ extractor $\mathcal{X}$ retrieved earlier, we can trivially retrieve $\vec{a}, \vec{\alpha}$ since for all $i = \{1, ..., l\}$ we have:

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

With $\vec{a}, \vec{\alpha}$ retrieved, we can directly compute $r, \epsilon, \vec{\tau}$ and validate if they match the same variable set extracted from the prover transcripts $C,\ R, tr_\epsilon$.

Finally, we need to check if linear polynomials $\vec{a}'$ match commitments $\vec{P}$ by testing if we can extract witnesses $\vec{a}, \vec{v}$ using $\vec{a}'$ and $\vec{P}$. The extractor first generates k_1 and then follows the protocol to get $PK_{\kappa_1}, tr_{\kappa_1}, PK_{\alpha_{t1}}, tr_{\alpha_{t1}}$ from the prover. The extractor then calls extractor $\mathcal{X}_S$ to retrieve κ_1 and α_{t1}. Rewind and repeat this procedure for another l times to retrieve $\kappa_2, ..., \kappa_{l+1}$ and $\alpha_{t2}, ..., \alpha_{tl+1}$ using evaluation points $k_2, ..., k_{l+1}$.

Through interpolation technique the extractor retrieves $(\alpha_i - v_i)$ and α_i for i in $\{1, ..., l\}$. With these information, we can now trivially compute $\vec{v}$ and verify if they can be mapped $\vec{P}$ s.t. $P_i = g^{a_i} h^{v_i}$ unless we found a non-trivial relationship among generators g, h.

B. Proof for Theorem Two

Proof. Perfect completeness follows from the fact that Protocol 2 is trivially complete. To prove perfect honest-verifier zero-knowledge, we define a simulator $\mathcal{S}$ to show that protocol 2 has perfect special honest verifier zero-knowledge for relation 1. $\mathcal{S}$ uses simulator $\mathcal{S}_S$ to simulate proof transcripts for proof of knowledge (discrete logarithm) protocols, and simulator $\mathcal{S}_p$ to simulate proof transcripts for polynomial commitment evaluation function PolyCommitEval.

The simulator $\mathcal{S}$ generates random group elements for $\vec{S}, \vec{T}, C, R$, proof of knowledge transcript tr_ϵ. After receiving challenge x from the verifier, the simulator generates l random integers to represent $\vec{e}$, l random integers to represent $\vec{a}'$, and one random integer to represent y and sends them to the verifier.

The simulator follows the protocol to compute o and PK_ϵ, then the simulator $\mathcal{S}$ calls simulator $\mathcal{S}_S$ to interact with the verifier and randomly generate all necessary transcripts to prove it knows the value of ϵ. This makes sense since we already know for a fact that schnorr and many other proof of knowledge protocols have perfect special honest verifier zero-knowledge. Similarly, the simulator $\mathcal{S}$ also calls simulator $\mathcal{S}_P$ to simulate transcripts for proving y is the evaluated value at point x for polynomial commitment C.

Next, simulator $\mathcal{S}$ simulates transcripts for proving the mapping from $\vec{a}'$ to $\vec{P}$. After challenge k is received from the prover, the simulator follows the protocol to compute PK_{v_t}, then calls simulator $\mathcal{S}_S$ to simulate transcripts needed to prove knowledge of v_t.

The simulator chooses all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or computes 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 2 is perfect special honest verifier zero-knowledge.

To prove computational witness extended emulation, we construct an extractor $\mathcal{X}$, which uses extractor $\mathcal{X}$ to extract witnesses from proof of knowledge transcripts and extractor $\mathcal{X}$ to extract witnesses from polynomial commitment C.

First, we show how to construct an extractor $\mathcal{X}$ for Protocol 2 s.t. on input $\vec{P} \in \mathbb{G}^l, R \in \mathbb{G}$, it either extracts witnesses $r, \epsilon, \vec{\tau}$, or discovers a non-trivial discrete logarithm relation between $g, h \in \mathbb{G}$. After this is done, we show the witnesses used to compute $r, \epsilon, \vec{\tau}$ match the witnesses of input commitments $\vec{P}$ s.t. relation 1 is satisfied.

The extractor $\mathcal{X}$ interacts with the prover in Protocol 2 and receives $\vec{S}$, $\vec{T}$, C, R, tr_ϵ from the prover. The extractor $\mathcal{X}$ then generates a challenge x_1 and forward it to the prover. After receiving $\vec{e}_1, \vec{a}'_1, y_1$, the extractor rewinds the prover and sends another challenge x_2 to receive $\vec{e}_2, \vec{a}'_2, y_2$.

The extractor then follows the protocol and calls extractor $\mathcal{X}_S$ to extract ϵ from tr_ϵ and PK_ϵ. With either x_1 or x_2, we can trivially retrieve r from r' since $r' = r + x \cdot \epsilon$ and validate $R = g^r h^\epsilon$.

Likewise, the extractor $\mathcal{X}$ calls extractor $\mathcal{X}_P$ using either x_1, y_1 or x_2, y_2 pair to retrieve coefficient set $\vec{\tau}$ from polynomial commitment C. Like that of Protocol 1, we can trivially validate the correctness r, ϵ from o at any evaluation point x since:

$$o = r + \epsilon \cdot x + \sum_{i=1}^{n} \tau_i \cdot x^{i+1}$$

We have now retrieved witnesses $r, \epsilon, \vec{\tau}$ using transcripts C, R, tr_ϵ. If the prover is honest, r, ϵ, τ are coefficients computed by the prover from $\vec{a}, \vec{\alpha}$, where as $\vec{\alpha}$ maps to blinding keys $\vec{\upsilon}$. We now go back to validate if $\vec{a}, \vec{\alpha}$ from $\vec{a}'$ maps to $\vec{a}, \vec{\upsilon}$ in $\vec{P}$ correctly by checking if we can extract these witnesses.

The extractor first generates k_1 and then follows the protocol to get $PK_{\upsilon_{t1}}, tr_{\upsilon_{t1}}$, then calls extractor $\mathcal{X}_S$ to retrieve υ_{t1} . The extractor then rewinds and repeats the above step l times to retrieve $\upsilon_{t2}, ..., \upsilon_{tl+1}$. Through interpolation the extractor retrieves witnesses υ_i for all i in $\{1, ..., l\}$. Dividing dividing P_i by h^{υ_i} we will get:

$$P_i / h^{\upsilon_i} = g^{a_i} \tag{128}$$

Using the two different challenges x_1, x_2 we mentioned earlier, the extractor gets $\vec{a}'_1$ and $\vec{a}'_2$ from the prover, which we can trivially retrieve $\vec{a}, \vec{\alpha}$ for all $i = \{1, ..., l\}$ since:

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

a_i must be the exponent of g in equality 128 or we found a non-trivial relationship among generators g, h.