Input Transformation Based Zero-Knowledge Argument System for Arbitrary Circuits with Both High Runtime Efficiency and Memory Efficiency

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 which 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.
Our benchmark result shows our approach can significantly improve verifier runtime performance by more than one order of magnitude over the state of the art while keeping the prover runtime and communication cost competitive with that of the state of the art.
In addition, the direct computation mechanism in our protocol allows the prover to add specifically designed gates to optimize the evaluation process. This is because the circuit is verified by verifiers linearly "computing" the circuit, which also enables us to bypass the "inactive part" of the circuit to further improve its performance.
Last but not least, our protocol is also memory-efficient. The theoretical memory cost of our protocol is just $O(n^{\frac{1}{2}})$, where n stands for the number of gates in a circuit. Unlike memory-efficient voice-based protocols, our protocol offers both high runtime performance and small communication cost and can be easily made non-interactive with the Fiat-Shamir heuristic.

1 Introduction

Ever since the discoveries of interactive proofs (IPs) [28] and probabilistically checkable proofs (PCPs) [5] [4] [3] [2] 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 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 the cost of running the encoder to translate circuit logic to constraint system form.

Due to the expensive computation cost in the setup phase of earlier SNARKs (Succinct Non-Interactive Argument of Knowledge), 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. The first universal SNARK was in Groth et al. [29], and Maller et al. improved the SRS size from quadratic to linear in Sonic [33]. More recently developed protocols such as PLONK [26], MARLIN [21] 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 [20], usually takes over 100 seconds on a single-threaded CPU for a circuit with over 2^{20} constraints.

Protocols belong to the Goldwasser, Kalai, and Rothblum (GKR) class such as Hyrax [38], Virgo [43]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [27] and Ligero/Ligero++

[1] [9] offer efficient prover runtimes 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 industry (e.g., the blockchain community) due to their expensive verifier runtime and high communication cost (hundreds of KBs) compared to fully succinct protocols such as STARK [7], PLONK, MARLIN, and Supersonic [19]. Furthermore, state-of-the-art GKR protocols generally have 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 (e.g., inputs are floating point numbers) are expected on smart contracts.

Memory-efficient privacy-free garbled circuits [31] [25] [30] and Vector Oblivious Linear Evaluation (VOLE) protocols [14] [34] [16] [15] [42] [40] [6] [41] generally offer better prover performance. However, their verifier runtimes are just as expensive as their prover runtime and generally cannot be easily made fully non-interactive, and their communication cost is easily many orders of magnitude more expensive than other approaches.

NIZKs such as SpartanNIZK [35] and later Lakonia [36] seem to offer a much more balanced approach, where they offer efficient prover runtime (6-18 seconds single thread) and competitive communication costs for large circuits (2^{20} constraints) 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.

Our aim is to create a new transparent zero-knowledge protocol that offers great flexibility to optimize and the best overall performance. Specifically, we want to keep the prover runtime cost and communication cost comparable to those of the state-of-the-art and improve the verifier runtime by one order of magnitude over that of the state-of-the-art. Finally, we also want our new system to be memory-efficient, or at least have the option to be memory-efficient.

1.1 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 the integer field so that verifiers can use standard integer operations to compute and verify each circuit output. In addition to performance gains, we believe such an approach offers more flexibility in designing customized sub-circuits/gates and would allow developers to code business rules exactly as they are described in business language.

In the past, Cramer and Damgård [24] first introduced a mechanism where input commitments are directly used in validating each multiplication gate. However, such an approach requires validation to be performed on each multiplication gate and therefore introduces a large communication overhead and requires both provers and verifiers to perform expensive operations in $\mathbb{G}$ for every multiplication gate. Although this approach, combined with some clever design, is adopted in some more recent protocols such as Hyrax [38], we still found that such an approach has some inherent inefficiencies and cannot be used to get the desired result we are seeking.

In our protocol, we start by transforming each committed input parameter into a circuit to some integer value in linear polynomial form, where verifiers can perform arithmetic operations (e.g., addition and multiplication) on them like they do on normal integers. Since field operations are cheap, the verifier can perform this step with very high efficiency.

For a simple circuit ${a_1}^d + {a_2}^d + {a_3}^d = r$ s.t. circuit inputs are a_1, a_2, a_3 and the circuit output is 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 $\mathbb{Z}_p$ s.t. $a'_i = a_i + X\alpha_i \in \mathbb{Z}_p$ (α_i is its blinding key). Since the transformed inputs are in $\mathbb{F}$, the verifier can plug these values directly into the circuit to compute the "temporary" output o e.g. $a'^d_1 + a'^d_2 + a'^d_3 = o$. The "temporary" circuit output $o \in \mathbb{Z}_p$ is the evaluation at point x of the output polynomial s.t.

$f(x) = o$. Since the degree of the output polynomial will increase after it is multiplied with another polynomial, the degree of the output polynomial of the sample circuit is $d + 1$. The constant term of this polynomial is the circuit output r and all other coefficients serve as its blinding keys. If the prover can prove that 1) it knows all coefficients of the output polynomial (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 $d+1$ because the transformed inputs (linear polynomials) are of degree 1. Multiplying them d times will give a polynomial with a degree of $d + 1$. So if the circuit is something like ${a_1}^2 + {a_2}^2 + {a_3}^2 +, ..., + {a_t}^2 = r$, the degree of the output polynomial is $3 = d + 1 \quad (d = 2)$ regardless of the value of t. Throughout our paper, we use the symbol m_p to denote the maximum number of multiplications included in any path that leads to the circuit output, which also defines the degree of the output polynomial (e.g. if the degree of the output polynomial is $d + 1$, then $m_p = d$).

This input transformation approach pioneered by our protocol does not require a"front end" encoder to compile the business logic relation R into some zero-knowledge friendly representation $\hat{R}$. This construct makes our protocol easy to implement and also makes it easier for end developers to design customized sub-circuits/gates (see section 3.3).

For a deep circuit where m_p value is large (unlike GKR based protocols, layers made up of addition gates make negligible performance impact in our protocol), the prover runtime of the base version of our protocol (Protocol 1) is dominated by $O({m_p}^2 + m_p + l)$ field operations and $O(m_p + m_p^{1/2} + l)$ group exponentiations, where m_p stands for the total number of multiplication gates included in the path that contains most multiplications and l stands for the number of inputs to a circuit; the verifier runtime is dominated by $O(n + m_p^{1/2} + l)$ field operations and $O({m_p}^{1/2} + l)$ group exponentiations; and the communication cost is dominated by $O({m_p}^{1/2} + 1)$ group elements and $O({m_p}^{1/2} + l)$ field elements.

On the other hand, if the circuit is shallow (e.g., for a circuit with $n/2$ addition operations and $n/2$ multiplication operations: $r = \sum_{i=1}^{n} a \cdot b$ where r is the circuit output and a, b are circuit inputs and n stands for the total number of gates in a circuit., we have $m_p = 1$), the prover work would be dominated by $O(n/2)$ field addition operations, which is very cheap.

Our protocol is specifically efficient for proving complex application logic where the circuit depth is high and can be greatly simplified using customized gates. Furthermore, our protocol gives the developer the power to significantly reduce the size of the circuit by bypassing the "inactive" part of the circuit logic.

Specifically, when processing an extremely deep circuit of 2^{20} sequential multiplication gates ($m_p = n$) with 960 input bits on a single CPU thread in the non-interactive setting, the prover runtime of our protocol is 5.3 seconds, the verifier runtime is 36 milliseconds, and the communication cost is approximately 76 kilobytes. This result shows a significant improvement in verifier runtime by one order of magnitude over the state of the art while keeping the prover runtime and communication cost competitive with the state of the art. If one interaction is allowed for the same circuit, the prover runtime of our protocol will further improve to 2.7 seconds, the verifier runtime will improve to 15 milliseconds, and the communication cost will improve to 65 kilobytes. To the best of our knowledge, our protocol offers the best prover performance in literature in the non-interactive setting. Alternatively, our protocol allows even faster prover runtime in exchange for bigger proof size.

On the memory side, the theoretical memory cost of our protocol is just $O(2 \cdot n^{\frac{1}{2}})$ to the circuit size n. This makes our protocol extremely attractive because voice-based memory-efficient protocols generally require one round of interaction and are extremely expensive in terms of verifier runtime cost and communication cost.

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.

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.

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

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

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 $\boldsymbol{Setup}$ 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**

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

- **Computational Soundness**

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

- **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 $(\textbf{Setup}, \mathcal{P}, \mathcal{V})$ 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 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. [12] 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 [39] and Spartan [35]. Similar techniques may be used to improve our implementation in the future to reduce proof size in the expense of longer verifier runtime. 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. $y = f(x)$.

Assume the polynomial commitment scheme we use in this paper is the one defined by Bootle et al. [12]. In section 5, we will introduced a modified version of the polynomial commitment evaluation scheme defined by Bootle et al. that tailors to our need.

2.4 Zero Knowledge Proof of Discrete Logarithm

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

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

In this paper, we assume the underlying implementation of the proof of discrete logarithm protocol is Schnorr's protocol. 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., $C = u_1^{a_1} u_2^{a_2} ... u_n^{a_n} \in \mathbb{G}$ 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. 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 integers a_i for $i = \{1, 2, ..., n\}$.

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

2.6 The Input Transformation Concept

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:

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

The above relation states that each input parameter to a circuit is represented by a commitment P_i in $\mathbb{G}$, which hides input value a_i with blinding key υ_i. r is the output of circuit E_c on inputs $\vec{P}$. The circuit output value is also represented by a commitment $R \in \mathbb{G}$ with blinding key ϵ.

We cannot take Pedersen commitments as inputs to a circuit because we cannot perform multiplications on them. We introduce a new concept called input transformation. The main idea is to transform committed inputs in $\mathbb{G}$ to linear polynomials in $\mathbb{F}$ where both the prover and the verifier can perform addition and multiplication operations. 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 integer value in linear polynomial form $a_i{}' \in \mathbb{Z}_p$:

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

Note that the blinding key of each input is updated to a random α_i s.t. $\alpha_i \neq \upsilon_i$. Likewise, the circuit output commitment $R = g^r h^\epsilon \in \mathbb{G}$ also has a matching linear polynomial.

$$r' = r + X\epsilon \in \mathbb{Z}_p \tag{3}$$

Since inputs represented by linear polynomials are just integer values, verifiers can perform arithmetic operations on them just as they do on integers. The output value of a circuit evaluation is now a polynomial with $m_p + 1$ 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 of the circuit output.

One of the key processes of our protocol is to reduce this output polynomial of $m_p + 1$ degree to a linear polynomial of degree 1 as stated in 3 by subtracting out all other terms of the output polynomial.

We are now ready to expand the definition of relation 1 above with the new blinding keys used to hide inputs and outputs. We say witnesses $r, \epsilon, \vec{\tau}$ are coefficients of the output polynomial after evaluating circuit E_c where $\vec{\tau}$ are coefficients of degree 2 term to degree $m_p + 1$ term. The updated relation is as follows:

$$\begin{aligned}\{(g, h, R \in \mathbb{G}, \vec{u} \in \mathbb{G}^{m_p}, \vec{P} \in \mathbb{G}^l, E_c \in \mathcal{C}_{\mathbb{F}}; \vec{a}, \vec{v} \in \mathbb{Z}_p^l, r, \epsilon \in \mathbb{Z}_p, \vec{\tau} \in \mathbb{Z}_p^{m_p}) : \\ P_i = g^{a_i} h^{\upsilon_i} \ \forall_i \in [1, l] \wedge R = g^r h^\epsilon \wedge E_c(\vec{a}, \vec{v}) = r, \epsilon, \vec{\tau}\}\end{aligned} \tag{4}$$

In our protocol, we can break the evaluation process into two sub-statements: to prove the first sub-statement, the prover proves each input in $\mathbb{G}$ is correctly transformed to $\mathbb{Z}_p$ by proving their mapping is legit; To prove the second sub-statement, the prover proves the circuit output is correctly computed from transformed inputs by showing the constant term of the output polynomial maps to the committed circuit output value.

To prove the first sub-statement we show how to prove each input transformation from P_i to a'_i is legit. Unlike commitments, the linear polynomial $a'_i \in \mathbb{Z}_p$ is 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 $a_i + \delta$). This is ok because the prover must commit to circuit output r' and its blinding key ϵ before the evaluation point X is known.

The prover commits to the differences between blinding key pairs $(\alpha_i - \upsilon_i)$ using blinding key μ. The verifier uses a random challenge k to generate $\vec{k} = k^1, ..., k^l$ so that verifiers can verify all transformed inputs in batch s.t.:

$$\kappa = \sum_{i=1}^{l} (\alpha_i - \upsilon_i) k^i \in \mathbb{Z}_p \tag{5}$$

$$PK_{\kappa\mu} = h^\kappa u^\mu \in \mathbb{G} \tag{6}$$

If the prover can prove the knowledge of κ, μ on generator $h, u \in \mathbb{G}$ using any proof of knowledge (proof of discrete logarithm) protocol, we can confirm that only the sum of products of blinding keys (exponent of h) is being updated except for a negligible probability. To prove and verify the knowledge of two exponents on two generators of a committed value, we extend the functionality of proveDL and verifyDL defined earlier:

- $\boldsymbol{ProveEDL}(h, u, \kappa, \mu) \rightarrow tr_{\kappa\mu}$ generates proof transcript $tr_{\kappa\mu}$, where κ, μ are witnesses.
- $\boldsymbol{VerifyEDL}(h, u, PK_{\kappa\mu}, tr_{\kappa\mu}) \rightarrow b \in \{0, 1\}$ takes a proof transcript $tr_{\kappa\mu}$ and verify a pair of discrete log relation ($PK_{\kappa\mu} = h^\kappa u^\mu$), and outputs $true$ if the knowledge of these relation is verified, $false$ otherwise.

The implementation of the two functions above is trivial and has been used in many existing protocols [13], one such example can be found in Appendix C. The key is to not allow attackers to retrieve h^κ and u^μ from proof transcripts. In protocol 1, the prover creates the proof transcript for the knowledge of κ, μ on generators $h, u \in \mathbb{G}$ as follows:

$$tr_{\kappa\mu} = ProveEDL(h, u, \kappa, \mu) \tag{7}$$

To verify all input mappings in one batch, the verifier adds the commitment to blinding key differences $PK_{\kappa\mu}$ back to the sum of the products of all input commitments:

$$P_t = (\prod_{i=1}^{l} P_i^{k^i}) \cdot PK_{\kappa\mu} \in \mathbb{G} \tag{8}$$

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

$$\alpha_t = \sum_{i=1}^{l} (\alpha_i) k^i \in \mathbb{Z}_p \tag{9}$$

$$tr_{\alpha_t \mu} = ProveEDL((h/g^x), u, \alpha_t, \mu) \tag{10}$$

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

$$t = \sum_{i=1}^{l} a_i' k^i \in \mathbb{Z}_p \tag{11}$$

$$PK_{\alpha_t \mu} = P_t / g^t = (h/g^x)^{\alpha_t} u^{\mu} \tag{12}$$

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.

To prove the second sub-statement To prove the circuit output is correctly computed from transformed inputs $\vec{a}'$, the prover needs to show it knows the coefficients of all terms 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 :

$$o = a_1{}' + a_2{}' = r + X \cdot \epsilon \tag{13}$$

Computing the output polynomial 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:

$$o = a_1 \cdot a_2 = r + X \cdot \epsilon + X^2 \cdot \tau \tag{14}$$

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 maximum number of multiplications included in any path that leads to the circuit output (m_p) increases.

We also need to reduce the circuit output from a degree $m_p + 1$ polynomial to a linear polynomial $r' = r + X\epsilon$ maps to the commitment $R = g^r h^{\epsilon}$. To get the linear polynomial we need from the raw output o, the verifier needs to subtract out all 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 of terms with degrees higher than one y to the verifier and engages with the verifier to prove $f(x) = X^2\tau = y$. With $y = X^2\tau$ validated, the verifier can subtract y from o to get the output in linear polynomial form:

$$r' = o - y \tag{15}$$

We call y the "breaker" of our protocol, because it subtracts all noises (polynomial terms of degree higher than one) from the raw circuit output o. In practice, the prover and the verifier engage in a polynomial commitment protocol to confirm $f(x) = y$.

$$C = \prod_{i=1}^{m_p} u_i^{\tau_i} \in \mathbb{G} \tag{16}$$

In the final version of our protocol, the polynomial commitment evaluation logic is broken apart and integrated with our protocol (protocol 3), so that we no longer need to call a polynomial commitment evaluation function as we do in Protocol 1 and Protocol 2 (see section 5). // // We define two more functions for our protocol. function ***computeKeys*** is used by the prover to compute the keys of a polynomial, and the function ***computeCircuit*** is used by verifiers to compute the value of the result polynomial at an evaluation point X:
1. function ***computeKeys***("input values", "input keys") takes 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 multiplies and adds input polynomials to compute coefficients of o.
2. function ***computeCircuit***("input values in linear polynomial form") trivially computes the result o from the inputs provided as they are integer values.

For example, if the circuit is to return the product of n inputs, then the computeCircuit function simply performs $o = a_1 \times a_2 \times, ..., \times a_n$. Since $a_1, ..., a_n$ are linear polynomials evaluated at point X, o is the evaluation of the resulting polynomial at point X, and the *computeKeys* function computes all coefficients of the resulting polynomial.

We don't waste space describing them in detail here since they are trivial to implement. With all the information available, we now formally introduce Protocol 1:

$$Input : (\vec{P} \in \mathbb{G}^l, \vec{u} \in \mathbb{G}^{m_p}, g, h \in \mathbb{G}, \vec{a}, \vec{v} \in \mathbb{Z}_p^l) \tag{17}$$

$$\mathcal{P}'s\ input : (\vec{P}, \vec{u}, g, h; \vec{a}, \vec{v}) \tag{18}$$

$$\mathcal{V}'s\ input : (\vec{P}, \vec{u}, g, h) \tag{19}$$

$$\mathcal{P}\ compute : \tag{20}$$

$$\alpha_i \xleftarrow{\$} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{21}$$

$$r, \varepsilon, \vec{\tau} = computeKeys(\vec{a}, \vec{\alpha}) \tag{22}$$

$$R = g^r h^{\varepsilon} \in \mathbb{G} \tag{23}$$

$$C = \prod_{i=1}^{m_p} u_i^{\tau_i} \in \mathbb{G} \tag{24}$$

$$\mathcal{P} \rightarrow \mathcal{V} : C, R \tag{25}$$

$$\mathcal{V}\ compute : \tag{26}$$

$$x \xleftarrow{\$} \mathbb{Z}_p \tag{27}$$

$$\mathcal{V} \rightarrow \mathcal{P} : x \tag{28}$$

$$\mathcal{P}\ compute : \tag{29}$$

$$a'_i = a_i + x\alpha_i \in \mathbb{Z}_p \qquad i = \{1, ..., l\} \tag{30}$$

$$y = \sum_{i}^{m_p} \tau_i \cdot x^{i+1} \in \mathbb{Z}_p \tag{31}$$

$$tr_\epsilon = ProveDL((h/g^x), \epsilon) \tag{32}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \ \vec{a}', y, tr_\epsilon \tag{33}$$

$$\mathcal{V}\ verify\ final\ output\ R : \tag{34}$$

$$o = computeCircuit(\vec{a}') \tag{35}$$

$$r' = o - y \in \mathbb{G} \tag{36}$$

$$PK_\epsilon = R/g^{r'} \in \mathbb{G} \tag{37}$$

$$\textbf{if}\ PolyCommitEval(C, y, x; \vec{\tau}) \tag{38}$$

$$\textbf{then}\ continue \tag{39}$$

else $reject$ (40)

if $VerifyDL((h/g^x), PK_\epsilon, tr_\epsilon)$ (41)

then $continue$ (42)

else $reject$ (43)

$\mathcal{V}$ $compute:$ (44)

$k \xleftarrow{\$} \mathbb{Z}_p$ (45)

$\mathcal{V} \rightarrow \mathcal{P} : k$ (46)

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

$$\alpha_t = \sum_{i=1}^{l} \alpha_i k^i \in \mathbb{Z}_p \quad (48)$$

$$tr_{\alpha_t \mu} = ProveEDL((h/g^x), u, \alpha_t, \mu) \quad (49)$$

$$\kappa = \sum_{i=1}^{l} (\alpha_i - \upsilon_i) k^i \in \mathbb{Z}_p \quad (50)$$

$$PK_{\kappa\mu} = h^\kappa u^\mu \in \mathbb{G} \quad (51)$$

$$tr_{\kappa\mu} = ProveEDL(h, u, \kappa, \mu) \quad (52)$$

$\mathcal{P} \rightarrow \mathcal{V} : PK_{\kappa\mu}, tr_{\kappa\mu}, tr_{\alpha_t\mu}$ (53)

$\mathcal{V}$ $verify\ inputs:$ (54)

$$P_t = (\prod_{i=1}^{l} P_i^{k^i}) \cdot PK_{\kappa\mu} \in \mathbb{G} \quad (55)$$

$$t = \sum_{i=1}^{l} a_i' k^i \in \mathbb{Z}_p \quad (56)$$

$$PK_{\alpha_t\mu} = P_t / g^t \quad (57)$$

if $VerifyEDL(h, u, PK_{\kappa\mu}, tr_{\kappa\mu})$, (58)

and $VerifyEDL((h/g^x), u, PK_{\alpha_t\mu}, tr_{\alpha_t\mu})$ (59)

then $accept$ (60)

else $reject$ (61)

Protocol 1

Theorem 1. *(Input Transformation Based Zero Knowledge Argument for Arbitrary Circuits). 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 their 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 circuit output o is a polynomial with $m_p + 1$ degree. By subtracting out all terms with degrees greater than one as in equation 15 explained, the verifier gets the circuit output in linear polynomial form that maps to the commitment R.

3 Making Input Transformation Based Zero Knowledge Protocol Efficient with NTT

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

3.1 Applying 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 $m_p \log m_p$, a major improvement over ${m_p}^2$ runtime of the trivial approach. However, a major drawback of NTT is that it requires a prime modulo q of the form $q = r \cdot 2^k + 1$ to be the prime order of the group, where k and c are arbitrary constants. Since the prime 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 a group $\mathbb{G}$ 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) the smaller the q value in bits, the lower the communication cost. In our benchmark testing, we set q to a 61-bit prime number for the interactive case and to a 93-bit prime number for the non-interactive case.

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

$$a'_i = a_i + x\alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{62}$$

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

$$S_i = g^{\omega_i \cdot q} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{63}$$

$$T_i = g^{\upsilon_i - \alpha_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{64}$$

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.:

$$e_i = ((x\alpha_i \bmod q) - x\alpha_i) \cdot x + \omega_i \cdot q \in \mathbb{Z}_p \qquad i = \{1, ..., l\} \tag{65}$$

The idea here is that when we subtract e_i from $a_i \cdot x$ in the next step, we can subtract out the blinding modulo q element ($x\ \alpha_i \bmod q$) from a'_i (e.g., $a'_i \cdot x - e_i = (a_i + x\alpha_i) \cdot x - \omega_i q$), assuming there is no overflow. The verifier can replace the $x^2\alpha_i - \omega_i q$ 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.

$$g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i = (g^x)^{a_i + x\upsilon_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{66}$$

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}.

$$PK_{\upsilon_t} = \prod_{i=1}^{l} \left(\frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{67}$$

$PK_{\upsilon_t} = ((h/g)^x)^{\upsilon_t}$. 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 $(h/g)^x \in \mathbb{G}$.

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.

$$\textbf{if } (e_i \bmod q) \stackrel{?}{=} 0, \textbf{ then } \textit{continue} \tag{68}$$

This test also implies the transformation process explained in this section is sound since the soundness of equation 67 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 68 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 66 to be true, the left side of the equality 66 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., $T_i^{x^2} \cdot S_i$), which only happens for a negligible chance of $1/q$.

This transformation process is zero-knowledge because 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, and can be represented as $s \cdot q$ for some s. This implies $e_i = (s + \omega) \cdot q$ for some randomly chosen ω. Since every other transcript (in $\mathbb{G}$) is trivially zero-knowledge, we say this transformation process is zero-knowledge if ω is correctly chosen (e.g., any number between $-s$ to p/q s.t. $e_i < p$).

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 66 67 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.

$$\textbf{if } a_i + (x\alpha_i \bmod q) \geq q, \textbf{ then } e_i = e_i - q \cdot x \qquad i = \{1, ..., l\} \tag{69}$$

As a consequence of this transformation to a smaller group q, the security level of commitment R is in question if $q << p$ and r, ϵ need to be protected. So we need another generator u and blinding key ϕ to protect r, ϵ in $\mathbb{Z}_q$ s.t.

$$R = g^r h^\varepsilon u^\phi \in \mathbb{G} \tag{70}$$

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

$$Input : (\vec{P} \in \mathbb{G}^l, \vec{u} \in \mathbb{G}^{m_p}, g, h \in \mathbb{G}, \vec{a}, \vec{v} \in \mathbb{Z}_p^l) \tag{71}$$

$$\mathcal{P}'s\ input : (\vec{P}, \vec{u}, g, h; \vec{a}, \vec{v}) \tag{72}$$

$$\mathcal{V}'s\ input : (\vec{P}, \vec{u}, g, h) \tag{73}$$

$$\mathcal{P}\ compute : \tag{74}$$

$$\alpha_i \stackrel{\$}{\leftarrow} \mathbb{Z}_q, \qquad i = \{1, ..., l\} \tag{75}$$

$$\omega_i \stackrel{\$}{\leftarrow} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{76}$$

$$\phi \stackrel{\$}{\leftarrow} \mathbb{Z}_p \tag{77}$$

$$r, \varepsilon, \vec{\tau} = computeKeys(\vec{a}, \vec{\alpha}) \in \mathbb{Z}_q^{m_p+2} \tag{78}$$

$$R = g^r h^\varepsilon u^\phi \in \mathbb{G} \tag{79}$$

$$C = \prod_{i=1}^{m_p} u_i^{\tau_i} \in \mathbb{G} \tag{80}$$

$$S_i = g^{\omega_i \cdot q} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{81}$$

$$T_i = g^{v_i - \alpha_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{82}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}, C, R \tag{83}$$

$\mathcal{V}$ *compute* : (84)

$x \xleftarrow{\$} \mathbb{Z}_q$ (85)

$\mathcal{V} \rightarrow \mathcal{P} : x$ (86)

$\mathcal{P}$ *compute* : (87)

$a'_i = a_i + x\alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$ (88)

$e_i = ((x\alpha_i \bmod q) - x\alpha_i)x + \omega_i q \in \mathbb{Z}_p \qquad i = \{1, ..., l\}$ (89)

if $a_i + (x\alpha_i \bmod q) \geq q$, **then** $e_i = e_i - q \cdot x \qquad i = \{1, ..., l\}$ (90)

$$y = \sum_{i}^{m_p} \tau_i \cdot x^{i+1} \in \mathbb{Z}_q \tag{91}$$

$tr_{\epsilon,\phi} = ProveEDL((h/g^x), u, \epsilon, \phi)$ (92)

$\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}', y, tr_{\epsilon,\phi}$ (93)

$\mathcal{V}$ *verify final output* : (94)

$o = computeCircuit(equation, \vec{a}') \in \mathbb{Z}_q$ (95)

$r' = o - y \in \mathbb{Z}_q \quad // \; r + x \cdot \epsilon$ (96)

$PK_{\epsilon,\phi} = R/g^{r'} \in \mathbb{G} \quad // \; (h/g^x)^\epsilon u^\phi$ (97)

if $PolyCommitEval(C, y, x; \vec{\tau})$, **then** *continue* (98)

else *reject* (99)

if $VerifyEDL((h/g^x), u, PK_{\epsilon,\phi}, tr_{\epsilon,\phi})$, **then** *continue* (100)

else *reject* (101)

$\mathcal{V}$ *compute* : (102)

$k \xleftarrow{\$} \mathbb{Z}_p$ (103)

$\mathcal{V} \rightarrow \mathcal{P} : k$ (104)

$\mathcal{P}$ *compute* : (105)

$$\upsilon_t = \sum_{i=1}^{l} \upsilon_i k^i \in \mathbb{Z}_p \tag{106}$$

$tr_{\upsilon_t} = ProveDL((h/g)^x, \upsilon_t)$ (107)

$\mathcal{P} \rightarrow \mathcal{V} : tr_{\upsilon_t}$ (108)

$\mathcal{V}$ *verify inputs* : (109)

if $(e_i \bmod q) \stackrel{?}{=} 0$, **then** *continue* $\qquad i = \{1, ..., l\}$ (110)

else *reject* (111)

$$PK_{\upsilon_t} = \prod_{i=1}^{l} \left(\frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{112}$$

if $VerifyDL((h/g)^x, PK_{\upsilon_t}, tr_{\upsilon_t})$, **then** *accept* (113)

else *reject* (114)

Protocol 2

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

Protocol 2 is a simplified version of Protocol 3. The proof for Theorem 2 is presented in Appendix B, which is also used to prove Theorem 3.

Since we are now evaluating the output polynomial at a smaller field q, the soundness error is increased due to the existence of polynomial roots. We use two NTT friendly primes in our implementation for performance test. One is a 61-bit prime $q = 1945555039024054273$, where $r = 27, k = 56, g = 5$ s.t. $q = r * 2^k + 1$ for the non-interactive case, and the second prime is a 93-bit number $q = 5241902353849032101525979137$, where $r = 271, k = 84, g = 3$ for the non-interactive case. The reason being that the smaller 61-prime number cannot provide soundness error at minimum 2^{-80} as required in most non-interactive use cases. See section 5 for a more detailed explanation.

3.2 The Asymptotic Cost of Protocol 2

The prover runtime of Protocol 2 is dominated by $O(m_p \log m_p + m_p + l)$ field operations and $O(m_p + {m_p}^{1/2} + l)$ group exponentiations; the verifier runtime is dominated by $O(n + {m_p}^{1/2} + l)$ field operations and $O({m_p}^{1/2} + l)$ group exponentiations; and the communication cost is dominated by $O({m_p}^{1/2} + l)$ group elements and $O({m_p}^{1/2} + l)$ field elements.

The worst possible scenario for our protocol is when we have a sequence of multiplications where both multiplier and multiplicand are the product of the previous multiplication operation (e.g. $((a^2)^2)^2$ is technically 3 multiplications, but it will result in $m_p = 2^3$), in such case m_p will grow exponentially. One way to tackle such a problem is to break the circuit into segments of smaller sub-circuits so that the m_p value will be refreshed whenever it grows oversize, similar to the idea of bootstrapping in fully homomorphic encryption. We will explore this option in the next section.

4 Enhancing Efficiency for Circuits with High Depth

The efficiency of protocol 2 will degrade as the total number of multiplications in the path computing the output (m_p value) grows, a not-so-uncommon scenario for circuits with high depth. One way to get around this problem is to have multiple breakers so that the number of polynomial terms will never exceed b s.t. $b = \frac{m_p}{m_b}$ (symbol m_b stands for the number of breakers).

4.1 Batch Verification With Multiple Breakers

Each breaker y_i is an evaluation at point x for terms $x^2, ..., x^{b+1}$ of the polynomial accumulated thus far in the computation. Obviously, it is not efficient for us to commit and evaluate m number of polynomials, where m stands for the number of breakers $(y_1, ..., y_m)$ we use to evaluate a circuit.

Fortunately, we just need to make a simple modification to the polynomial commitment evaluation protocol defined by Bootle et al. to enable verifiers to evaluate m_b breakers all at once. We start by aligning each breaker y_i to the coefficients of the ith generator of vector commitments (columns of the $m_b \times b$ matrix).

$$\begin{pmatrix} u_1^{y'_1} \\ u_2^{y'_2} \\ u_3^{y'_3} \\ . \\ . \\ u_{m_b}^{y'_{m_b}} \end{pmatrix} = \begin{pmatrix} u_1^{\tau_{1,1}} & u_1^{\tau_{1,2}} & . & . & u_1^{\tau_{1,b}} \\ u_2^{\tau_{2,1}} & u_2^{\tau_{2,2}} & . & . & u_2^{\tau_{2,b}} \\ u_3^{\tau_{3,1}} & u_3^{\tau_{3,2}} & . & . & u_3^{\tau_{3,b}} \\ . & . & . & . & . \\ . & . & . & . & . \\ u_{m_b}^{\tau_{m_b,1}} & u_{m_b}^{\tau_{m_b,2}} & . & . & u_{m_b}^{\tau_{m_b,b}} \end{pmatrix} \begin{pmatrix} x^2 \\ x^3 \\ x^4 \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

Figure 1

Let $y_i = (y'_i) \bmod q$, we can observe from figure 1 that each y'_i can be computed from the sum of products of exponents of u_i (e.g. $y'_i = \tau_{i,1}x^2 + \tau_{i,2}x^3 + ... + \tau_{i,b}x^{b+1}$).

In our protocol, the prover commits to the columns of the matrix in figure 1 in the same way as that in Bootle et al.'s polynomial commitment evaluation scheme.

$$C_j = \prod_{i=1}^{m_b} u_i^{\tau_{i,j}} \qquad \text{for} \qquad j = \{1, ..., b\} \tag{115}$$

When the evaluation point x is known, the verifier computes the exponent of each C_j. If the equality below is true, all breakers are verified.

$$\prod_{i=1}^{m_b} u_i^{y'_i} \stackrel{?}{=} \prod_{j=1}^{b} C_j^{x^{j+1}} \tag{116}$$

Note that if each breaker y'_i is equal to the sum of products of b terms and each term is a product of $x^{j+1} \in \mathbb{Z}_q$ and $\tau_{i,j} \in \mathbb{Z}_q$, then the bit length of $|y'_i|$ is approximately $|y'_i| \leq 2 \cdot |q| + |b|$. The value of y'_i can also be expressed as $y'_i = y_i + z \cdot q$ for some z and $|z| \leq |q| + |b|$. However, passing raw y'_i values to the verifier may leak some information about the coefficients.

To cope with that, we make the prover commit to a blinding vector $\vec{\beta} \in \mathbb{Z}_m^{m_b}$ and m value is unknown to the verifier [1]. Each y'_i is now computed as:

$$y'_i = \sum_{j=1}^{b} \tau_{i,j} x^j + \beta_i \qquad \text{for} \qquad i = \{1, ..., m_b\} \tag{117}$$

Note that the power (exponent) of x in each term in the equation above is one degree lower than it needs to be, so to get y_i from y'_i the verifier needs to multiply x one more time:

$$y_i = (y'_i \cdot x) \bmod q \in \mathbb{Z}_q \qquad \text{for} \qquad i = \{1, ..., m_b\} \tag{118}$$

This implies the value of the circuit output r' is now updated to m_b number of $r'_i = r_i + x(\epsilon_i - \beta_i) \in \mathbb{Z}_q$, so we need to adjust the blinding key of r' by adding β_i to it in the *computeKeys* function. The updated equality graph is shown in figure 2 below.

$$\begin{matrix} u_1^{y'_1} \\ u_2^{y'_2} \\ u_3^{y'_3} \\ . \\ . \\ \mathrm{u}_{m_b}^{y'_{m_b}} \end{matrix} = \begin{pmatrix} u_1^{\tau_{1,1}} & u_1^{\tau_{1,2}} & . & . & u_1^{\tau_{1,b}} \\ u_2^{\tau_{2,1}} & u_2^{\tau_{2,2}} & . & . & u_2^{\tau_{2,b}} \\ u_3^{\tau_{3,1}} & u_3^{\tau_{3,2}} & . & . & u_3^{\tau_{3,b}} \\ . & . & . & . & . \\ . & . & . & . & . \\ u_{m_b}^{\tau_{m_b,1}} & u_{m_b}^{\tau_{m_b,2}} & . & . & u_{m_b}^{\tau_{m_b,b}} \end{pmatrix} \begin{pmatrix} x \\ x^2 \\ x^3 \\ . \\ . \\ x^b \end{pmatrix} \cdot \begin{pmatrix} u_1^{\beta_1} \\ u_2^{\beta_2} \\ u_3^{\beta_3} \\ . \\ . \\ u_{m_b}^{\beta_{m_b}} \end{pmatrix}$$

Figure 2

[1] Alternatively, if m is a public information for whatever reason, one can set the upper bound of $\vec{\beta} \in \mathbb{Z}_{m'_j}^{m_b}$ to m'_j s.t. $m'_j = m - \sum_{i=1}^{m_b} u_i^{\tau_{i,j}} \cdot q$

Let $B = \prod_{i=1}^{m_b} u_i^{\beta_i}$, the equality in figure 2 can also be expressed using the equality check below:

$$\prod_{i=1}^{m_b} u_i^{y_i'} \stackrel{?}{=} \prod_{j=1}^{b} C_j^{x^j} \cdot B \tag{119}$$

If the commitments $\vec{C}, B$ and vector $\vec{y}$ satisfy the equation above, then we know breakers $\vec{y}$ are valid. The verifier applies equation 118 to each y_i' to get the actual breakers y_i used in computing o.

Since we are now evaluating "multiple commitments" by applying m_b breakers, we break the function $computeCircuit$ we defined earlier to m_b number of $computeSubCircuit_i$ functions.

function $\boldsymbol{computeSubCircuit}_i$ ("inputs in linear polynomial form", "output from the previous computeSubCircuit function"): for $i = \{1, .., m_b\}$, it trivially computes the result o_i from the inputs to the whole circuit and output from the previous computeSubCircuit function.

We are now ready to introduce Protocol 3, which replaces the generic polynomial commitment evaluation in Protocol 2 with the multi-breaker mechanism we introduced in this section.

$$Input : (\vec{P} \in \mathbb{G}^l, \vec{u} \in \mathbb{G}^{m_b}, g, h \in \mathbb{G}, \vec{a}, \vec{v} \in \mathbb{Z}_p^l) \tag{120}$$

$$\mathcal{P}'s\ input : (\vec{P}, \vec{u}, g, h; \vec{a}, \vec{v}) \tag{121}$$

$$\mathcal{V}'s\ input : (\vec{P}, \vec{u}, g, h) \tag{122}$$

$$\mathcal{P}\ compute : \tag{123}$$

$$\alpha_i \stackrel{\$}{\leftarrow} \mathbb{Z}_q, \qquad i = \{1, ..., l\} \tag{124}$$

$$\omega_i \stackrel{\$}{\leftarrow} \mathbb{Z}_p, \qquad i = \{1, ..., l\} \tag{125}$$

$$\beta_i \stackrel{\$}{\leftarrow} \mathbb{Z}_m, \qquad i = \{1, ..., m_b\} \tag{126}$$

$$\phi \stackrel{\$}{\leftarrow} \mathbb{Z}_p \tag{127}$$

$$r, \varepsilon, \vec{\tau} = computeKeys(\vec{a}, \vec{\alpha}) \in \mathbb{Z}_q^{m_p+2} \tag{128}$$

$$R = g^r h^{\varepsilon} u^{\phi} \in \mathbb{G} \tag{129}$$

$$C_j = \prod_{i=1}^{m_b} u_i^{\tau_{i,j}} \in \mathbb{G} \qquad i = \{1, ..., b\} \tag{130}$$

$$S_i = g^{\omega_i \cdot q} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{131}$$

$$T_i = g^{v_i - \alpha_i} \in \mathbb{G} \qquad i = \{1, ..., l\} \tag{132}$$

$$B = \prod_{i=1}^{m_b} u_i^{\beta_i} \in \mathbb{G} \tag{133}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{S}, \vec{T}, \vec{C}, B, R \tag{134}$$

$$\mathcal{V}\ compute : \tag{135}$$

$$x \stackrel{\$}{\leftarrow} \mathbb{Z}_p \tag{136}$$

$$\mathcal{V} \rightarrow \mathcal{P} : x \tag{137}$$

$$\mathcal{P}\ compute : \tag{138}$$

$$a_i' = a_i + x\alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\} \tag{139}$$

$$e_i = ((x\alpha_i \bmod q) - x\alpha_i)x + \omega_i q \in \mathbb{Z}_p \qquad i = \{1, ..., l\} \tag{140}$$

$$\textbf{if } a_i + (x\alpha_i \bmod q) > q, \textbf{ then } e_i = e_i - q \cdot x \qquad i = \{1, ..., l\} \tag{141}$$

$$y_i' = \sum_{j}^{b} \tau_{i,j} \cdot x^j + \beta_i \in \mathbb{Z}_p \qquad i = \{1, ..., m_b\} \tag{142}$$

$$tr_{\epsilon,\phi} = ProveEDL((h/g^x), u, \epsilon, \phi) \tag{143}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}\,', \vec{y'}, tr_{\epsilon,\phi} \tag{144}$$

$$\mathcal{V}\ verify\ final\ output: \tag{145}$$

$$\textbf{if } (\prod_{i=1}^{m_b} u_i^{y_i'} \stackrel{?}{=} \prod_{j=1}^{b} C_j^{x^j} \cdot B) \textbf{ then } continue \tag{146}$$

$$\textbf{else } reject \tag{147}$$

$$\textbf{for } \quad i = 1, ..., m_b \quad \{ \tag{148}$$

$$o_i = computeSubCircuit_i(\vec{a}\,', r') \in \mathbb{Z}_p \tag{149}$$

$$y_i = (y_i' \cdot x) \bmod q \in \mathbb{Z}_q \tag{150}$$

$${r_i}' = o_i - y_i \in \mathbb{Z}_p \tag{151}$$

$$r' = {r_i}' \in \mathbb{Z}_q \quad \} \tag{152}$$

$$PK_{\epsilon,\phi} = R/g^{r'} \in \mathbb{G} \quad // \ (h/g^x)^{\epsilon} u^{\phi} \tag{153}$$

$$\textbf{if } VerifyEDL((h/g^x), u, PK_{\epsilon,\phi}, tr_{\epsilon,\phi}), \textbf{ then } continue \tag{154}$$

$$\textbf{else } reject \tag{155}$$

$$\mathcal{V}\ compute: \tag{156}$$

$$k \stackrel{\$}{\leftarrow} \mathbb{Z}_p \tag{157}$$

$$\mathcal{V} \rightarrow \mathcal{P} : k \tag{158}$$

$$\mathcal{P}\ compute: \tag{159}$$

$$v_t = \sum_{i=1}^{l} v_i k^i \in \mathbb{Z}_p \tag{160}$$

$$tr_{v_t} = ProveDL((h/g)^x, v_t) \tag{161}$$

$$\mathcal{P} \rightarrow \mathcal{V} : tr_{v_t} \tag{162}$$

$$\mathcal{V}\ verify\ inputs: \tag{163}$$

$$\textbf{if } (e_i \bmod q) \stackrel{?}{=} 0, \textbf{ then } continue \qquad i = \{1, ..., l\} \tag{164}$$

$$\textbf{else } reject \tag{165}$$

$$PK_{v_t} = \prod_{i=1}^{l} \left(\frac{P_i^x}{g^{a_i' \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G} \tag{166}$$

$$\textbf{if } VerifyDL((h/g)^x, PK_{v_t}, tr_{v_t}), \textbf{ then } accept \tag{167}$$

$$\textbf{else } reject \tag{168}$$

Protocol 3

Theorem 3. *(Input Transformation Based Efficient Zero Knowledge Argument for Arbitrary Circuits with High Efficiency). 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 3 is presented in Appendix C.

From line 149 to 153 in protocol 3 we assumed our circuit is a linear circuit where there is only one straight path to the output because it is easy to explain. In practice, binary circuits generally have multiple "bit" paths that executes in parallel.

4.2 Booleanity Check and Bit Decomposition/Reposition

A common requirement in proving binary circuits is the need to enforce input data $b_i \in \{0, 1\}$ for some $i \in \{1, ..., l\}$. In practice, it is useful to decompose l full integer inputs into $l \cdot 32$ bits (assuming we use 32 bits to represent a full integer, like the int type in Java) in order to perform comparison operations on input data. If a committed value b_i is in $[0, 1]$, then its linear polynomial form b_i must have the following property:

$$(b'_i \cdot b'_i - b'_i) = \delta_{1,i}x + \delta_{2,i}x^2 \tag{169}$$

Where $\delta_{2,i} = \beta_i^2$, and $\delta_{1,i} = \beta_i$ when b_i is 1 and $\delta_{1,i} = -\beta$ when b_i is 0. To prove the correctness of all $b_i \in \{0, 1\}$, the prover commits to two polynomials K_1, K_2 s.t.

$$D = u_1^{\delta_1} u_2^{\delta_2} ... u_l^{\delta_l} h^{\rho_1} \qquad and \qquad E = u_1^{\epsilon_1} u_2^{\epsilon_2} ... u_l^{\epsilon_l} h^{\rho_2} \tag{170}$$

Where D commits to coefficients on x term for $i \in \{1, ..., l \cdot 32\}$ and E commits to coefficients on x^2 term for $i \in \{1, ..., l \cdot 32\}$. The prover sends D, E to the verifier. After 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 if the equality below is true:

$$y_1 \cdot x + y_2 \cdot x^2 = \sum_{j=1}^{l \cdot 32} (b'_i \cdot b'_i - b'_i) \cdot k^i \tag{171}$$

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

$$a'_i = \sum_{j=1}^{32} b'_{i,j} \cdot 2^j \tag{172}$$

We define the protocol ValidateBooleanity below with the equations defined above

$Input : (\vec{b}' \in \mathbb{Z}_q : \vec{b}, \vec{\beta}' \in \mathbb{Z}_q)$
$\mathcal{P}'s\ input : (\vec{b}' : \vec{b}, \vec{\beta}')$
$\mathcal{V}'s\ input : (\vec{b}')$
$\mathcal{P}\ compute :$
$l = |\vec{b}'| \in \mathbb{Z}_q$
$\rho_1, \rho_2 \xleftarrow{\$} \mathbb{Z}_q$
$\delta_i = b_i\beta_i + b_i\beta_i - \beta_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$
$\epsilon_i = \beta_i^2 \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$
$D = \prod_{i=1}^{l} u_i^{\delta_i} \cdot h^{\rho_1} \in \mathbb{G}$
$E = \prod_{i=1}^{l} u_i^{\epsilon_i} \cdot h^{\rho_2} \in \mathbb{G}$

$\mathcal{P} \rightarrow \mathcal{V} : D, E$

$\mathcal{V}$ *compute* :

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

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

$\mathcal{P}$ *compute* :

$$y_1 = \sum_{i=1}^{l} \delta_i k^i \in \mathbb{Z}_q$$

$$y_2 = \sum_{i=1}^{l} \epsilon_i k^i \in \mathbb{Z}_q$$

$\mathcal{P} \rightarrow \mathcal{V} : y_1, y_2$

$\mathcal{P}, \mathcal{V}$ engate to evaluate :

if $PolyCommitEval(D, y_1, k; \vec{\delta}, \rho_1)$

$\wedge$ $PolyCommitEval(E, y_2, k; \vec{\epsilon}, \rho_2)$

return true

else

return false

Protocol ValidateBooleanity

In practice, we will conduct booleanity test on all $l \cdot 32$ bit values at once and then use equation 172 to convert them into l full integer values so that we can perform the "linear polynomial to Pedersen commitment" mapping test explained in the last two sections. In some special cases, one may also call Protocol ValidateBooleanity in customized subcircuits as we will show in the next subsection.

4.3 Optimization Techniques and Custom Subcircuit

One way to optimize the prover performance of a circuit is by changing the strategy of setting the breakers $(y_1, ..., y_{m_b})$.

Although we set $m_b = b$ in our earlier example, this is not necessarily the most efficient way of setting breakers. For example, setting b smaller than m_b could offer better prover runtime performance in exchange for more expensive communication costs.

It is also not necessary that all breakers $y_1, ..., y_{m_b}$ have the same b value. For example, If some value a_i is taken to its 100th power $(a_i'^{100}$, degree term $m_p = 100)$ 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.

Another way to optimize performance is through the use of specially designed subcircuits. The idea is similar to "custom gates" found in SNARKs protocols in principle, but our protocol likely offers more freedom in designing and using these "custom gates". This is because both the prover and the verifier evaluate the circuit in a linear fashion in our protocol, so we can insert any type of specially designed gate at any point of the circuit.

For example, suppose there is a conditional statement in a circuit to test if some number $b' = b + X\beta$ is less than 2^{20}, we can design a subcircuit (custom gate) just for this logic because it would be quite inefficient to construct a binary circuit for it. So instead the prover can decompose the number b to its bit representations $(b'_i = b_i + X\beta_i$ s.t. $b = \sum_{i=1}^{32} b_i \cdot 2^{i-1} \wedge b_i \in \{0, 1\}$ for $i = \{1, ..., 32\}$) as shown in the previous sub-section and prove that the sum of all bits greater than 20 is equal to zero.

The prover commits to $\beta_t = \sum_{i=21}^{32} \beta_i$ (sum of blinding keys of bits higher than 21) with another blinding key ρ (this is because β_t is in $\mathbb{Z}_q$, which could be a much smaller field than $\mathbb{Z}_p$)

$$C = g^{\beta_t} h^{\rho} \in \mathbb{G}$$

If $b_i = 0$ for $i = \{21, ..., 32\}$, the verifier can compute $PK_\rho = h^\rho$ from bits $\vec{b}'$ s.t..

$$c = \sum_{i=21}^{32} b'_i \in \mathbb{Z}_q$$
$$PK_\rho = C / g^{\frac{c}{x}} \in \mathbb{G}$$

If the prover can prove the knowledge of ρ, then the statement of " $b < 2^{20}$ " is validated. The "computeSubCircuit" function can move on to the next stage of the circuit computation.

We can also use a custom subcircuit to bypass the "inactive" part of a circuit (similar to that of suBlonk (ePrint). For example, If a conditional statement is in the form of **if** $x < 20$ **then** "do something" **else** "do something else", then both the prover and the verifier can bypass the "else" part of the circuit if the condition returns true.

For example, we have a circuit that evaluates whether the average of all inputs is less than 2^{20}. if true, it will perform some logic, otherwise, it will do something else (e.g. loop through all inputs and take out the highest values until the condition is met). The ComputeSubCircuit function for this circuit would look like:

computeSubCircuit : $(\vec{a}', \vec{b}' \in \mathbb{Z}^q, tr_\rho)$

$b = (\sum_{i=1}^{l} a_i)/l \in \mathbb{Z}_q$

if $ValidateBooleanity(\vec{b}')$

$\wedge\, b = \sum_{i=1}^{32} b'_i \in \mathbb{Z}_q$ **then**

$c = \sum_{i=21}^{32} b'_i \in \mathbb{Z}_q$

$PK_{\beta_t} = C / g^{\frac{c}{x}} \in \mathbb{G}$

if $VerifyDL(h, PK_\rho, tr_\rho)$ **then**

$o =$ do something

else

$o =$ do something else

else

$o =$ do something

return o

Function computeSubCircuit (Customized)

If the conditions in both "if" conditions are both true, then logics in both "else" statements are considered inactive and can be omitted altogether from the circuit.

However, it is worth noticing that using a customized subcircuit bypassing parts of the circuit may leak information about data to attackers, so one must use such a strategy with extreme caution.

4.4 The Asymptotic Cost of Protocol 3

For a circuit with l input parameters s.t. each input parameter is composed of 32 bits, the prover runtime of the final version (Protocol 3) of our protocol is dominated by $O(m_p \log {m_p}^{1/2} + m_p + l + 32 \cdot l^{1/2})$ field operations (assuming additive operations in $\mathbb{F}$ are practically free) and $O(m_p + {m_p}^{1/2} + l + 32 \cdot l^{1/2})$ group exponentiations; the verifier runtime is dominated by $O(n + {m_p}^{1/2} + l + 32 \cdot l^{1/2})$ field operations and $O({m_p}^{1/2} + l + 32 \cdot l^{1/2})$ group exponentiations; and the communication cost is dominated by $O({m_p}^{1/2} + l + 32 \cdot l^{1/2})$ group elements and $O({m_p}^{1/2} + l + 32 \cdot l^{1/2})$ field elements.

4.5 Memory Efficiency of Protocol 3

The memory consumption cost of protocol 3 is $O(n)$. However, with a simple modification to protocol 3, we can improve the memory consumption cost of our protocol to $m_b + b$ or $O(2 \cdot n^{\frac{1}{2}})$.

Instead of computing $\vec{C}$ after $\vec{\tau}$ is computed s.t. $|\vec{\tau}| = n$. We can break the "computeKeys" function into m_b "computeSubCircuitKeys" functions s.t. we only compute the coefficients for the ith sub-circuit in each round and commit/update $\vec{C}$ after each round for m_b rounds.

For example, let's say we have C_j for $j = 1, ..., b$ after round i, the protocol computes coefficients $\tau_{i',1}, ..., \tau_{i',b}$ for subCircuit i' where $i' = i + 1$. The protocol then updates each element of $\vec{C}$ to $C'_j = C_j \cdot g^{\tau_{i,j}}$ for $j = 1, ..., b$ and then discards the coefficients computed for subCircuit i'. This will give us a memory consumption cost of $O(2 \cdot n^{\frac{1}{2}})$. This can be achieved by replacing lines 128 to 130 with the following steps:

$$r, \epsilon, \vec{\tau}_1 = computeSubCircuit_1(\vec{a}, \vec{\alpha}); \tag{173}$$

$$C_j = u_i^{\tau_{1,j}} \in \mathbb{G} \qquad i = \{1, ..., b\} \tag{174}$$

$$\textbf{for} \quad i = 2, ..., m_b \quad \{ \tag{175}$$

$$\vec{a}' = \vec{a} \,||\, r, \qquad \vec{\alpha}' = \vec{\alpha} \,||\, \epsilon \tag{176}$$

$$r, \epsilon, \vec{\tau}_i = computeSubCircuit_i(\vec{a}', \vec{\alpha}'); \tag{177}$$

$$C_j = C_j \cdot u_i^{\tau_{i,j}} \in \mathbb{G} \qquad i = \{1, ..., b\} \quad \} \tag{178}$$

$$R = g^r h^\epsilon u^\phi \in \mathbb{G} \tag{179}$$

In this example, outputs r, ϵ from the $computeSubCircuitKeys_i$ function are appended to the circuit inputs $\vec{a}, \vec{\alpha}$ and then used as inputs for the $computeSubCircuit_{i+1}$ function, which will output a new pair of r, ϵ. Since coefficients $\vec{\tau}_i$ will get discarded from the memory after each iteration, we have to recompute them after challenge x is available. Replace line 142 with the following lines:

$$\textbf{for} \quad i = 1, ..., m_b \quad \{ \tag{180}$$

$$\vec{a}' = \vec{a} \,||\, r, \qquad \vec{\alpha}' = \vec{\alpha} \,||\, \epsilon \tag{181}$$

$$r, \epsilon, \vec{\tau}_i = computeSubCircuit_i(\vec{a}', \vec{\alpha}'); \tag{182}$$

$$y'_i = \sum_j^b \tau_{i,j} \cdot x^j + \beta_i \in \mathbb{Z}_p \quad \} \tag{183}$$

The obvious caveat is that we can no longer use Pippenger acceleration to compute each C_j as we did in protocol 3, and we also have to recompute coefficients for each sub-circuit to get $\vec{y}$. Although the cost of recomputing coefficients is relatively cheap, the cost of leaving out Pippenger acceleration is expensive.

A meet-in-the-middle approach is to break the computation of $\vec{C}$ into t segments, and in each segment we compute $\frac{n}{t}$ coefficients and update $\vec{C}$ after each segment. By doing so, we can achieve a consumption cost of $O(\frac{n}{t})$. This strategy really helps when the circuit is big (e.g., when the circuit size is greater than 2^{20}, arguably the only time memory cost would even matter).

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. Only one core is being utilized, and all tests are run on a single CPU thread. Our test code is a non-interactive implementation (using Fiat-Shamir heuristic) of Protocol 3 (not the memory-efficient option mentioned in section 4.5 because we want to leverage the Pippenger acceleration to get the most optimal runtime result).

The baseline protocols we picked are 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 runtime, verifier runtime, proof size), and the NIZK version of Spartan offers the most balanced performance across all performance parameters. We also do not consider SNARKs even though most of them can be made transparent by switching to a transparent polynomial commitment scheme, as they are hardly efficient after the switch.

We didn't consider transparent protocols that depends on circuit depth such as GKR-based protocols simply because they can't handle 2^{20} sequential multiplications. We also don't consider voice-based protocols, as they are only optimized for prover work and generally require one round of interaction. Other popular transparent schemes such as Bulletproofs are also not being considered because they have a linear verifier runtime 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 the number of inputs to our protocol to 30 integers, and each input is represented by 32 bits so that there are a total of $30 \cdot 32 = 960$ input bits to the circuit. The circuit we use performs n sequential multiplications on l inputs, so we have $m_p = n$, likely much closer to the worst-case scenario of our protocol than the test cases of other protocols that we are comparing against. If we run a shallow circuit where m_p number is small, the benchmark result will likely be significantly better. For example, if we have a circuit where $m_p = 1$, then its prover runtime performance will be comparable to the verifier runtime of our protocol.

We picked two NTT prime numbers for our benchmark testing, one for the interactive case and another for the non-interactive case.

The NTT prime number we picked for the non-interactive case is $q = 1945555039024054273$, a 61-bit number that implies the soundness error will be at most 2^{-51} for a circuit with 2^{20} sequential multiplications where $m_p = n$, which is more than enough in real-life applications where one interaction is allowed.

For the non-interactive case (through using the Fiat-Shamir heuristic), the prime number for our integer field is 52419023538490321015259791 37, a 93-bit prime that implies the soundness error will be at most 2^{-83} for a deep circuit with 2^{20} sequential multiplications where $m_p = n$.

To maximize the advantage of the NTT algorithm in computing sequential multiplications, we process each segment $(1, ..., m_b)$ 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.

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. One challenge we had was building an efficient 128-bit multiplication function. Unlike the 64-bit (provided by the CPU) and 256-bit (heavily optimized with assembly code for various crypto computations such as ECC) multiplications, we couldn't find any efficient multiplication function for 128-bit multiplication so we had to do it ourselves. We

did the best we could to optimize our 128-bit multiplication without using assembly code, so there are still rooms for improvement if assembly code is used.

We set $m_b = b$ to get a more balanced result. Alternatively, one can set $m_b > b$ to get better prover runtime performance in exchange for more expensive communication costs. This is because doing so will 1) evade expensive NTT computations at high degrees and 2) better leverage Pippenger acceleration in computing $\vec{C}$, which will continuously improve group exponentiation operations before peaking out at around $m_b = 2^14$.

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(61 bit)	0.04	0.06	0.14	0.33	0.94	2.7
This Work(93 bit)	0.08	0.15	0.3	0.8	2	5.3

Table 1. Prover performance comparison (seconds)

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. This is because the cost associated with the number of inputs to the circuit is fixed (960 bits), 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. If that's not enough, we can always set $m_b > b$ to get an even better prover runtime benchmark.

From the chart, we can observe that only SpartanNIZK offers comparable (only slightly worse) prover runtime performance in the non-interactive case to that of our protocol, but it is worth noting that this is not a fair comparison in our favor since we're comparing the evaluation of 2^{20} constraints in SpartanNIZK (provided by its test code) with the unlikely scenario of 2^{20} sequential multiplications in that of our protocol.

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(61*bit*)	16	17	21	27	39	65
This Work(93*bit*)	20	22	26	33	48	77

Table 2. Proof size comparison (kilobytes)

Table 2 shows that the communication cost of our protocol dominates that of Ligero and Aurora, while largely comparable to SpartanNIZK and Hyrax. For higher input number counts, see Table 4 for more detail.

Table 1 and 2 show that our protocol is largely comparable to the current state of the art in terms of prover runtime and communication cost. Table 3 demonstrates that our protocol achieves a

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(61*bit*)	5	5	6	7	9	15
This Work(93*bit*)	5	5	6	8	15	37

Table 3. Verifier performance comparison (milliseconds)

significant improvement of one order of magnitude in verifier runtime over all the baseline protocols we are comparing against. Like that of communication cost, the verifier runtime of our protocol will grow when the number of inputs to the protocol grows.

Some may consider 30 integer inputs and 960 input bits to a circuit too small, so in table 4 we list performance benchmarks for different numbers of inputs (l) to a circuit with 2^{20} sequential multiplications.

Input bits ($l \cdot 32$)	Input Integers (l)	Prover time(s)	Verifier time(ms)	Proof size(kb)
base	base	2.7	11	52
320	10	2.7	13	58
640	20	2.7	14	62
960	30	2.7	15	65
1,280	40	2.7	15	69
1,600	50	2.8	17	73
1,920	60	2.8	19	77
2,240	70	2.8	21	81
2,560	80	2.8	22	85
2,880	90	2.8	23	88
3,200	100	2.8	24	93

Table 4. Performance comparison for different input numbers of a circuit with 2^{20} sequential multiplications s.t. $m_p = n$ using 61-bit prime

In table 4 and table 5 we can observe that increases in prover runtime and verifier runtime are small as the input bit count approaches $3,200$. This is because the total input number is still small compared to the size of the circuit (2^{20} sequential multiplications).

Communication costs get impacted the most as we add more inputs to the circuit. Generally speaking, more inputs usually imply shallower circuit depth (e.g. $m_p << n$) and less complex business logic; therefore, a larger input number would generally result in better prover and verifier runtime performance at the expense of a more costly proof size (communication cost).

It is worth noticing that input transformation costs can be shared across multiple circuits if the inputs are reused in other circuit verifications. Only the "base" case is the pure circuit cost that cannot be shared between circuits, this may lead to further reductions in communication costs in the real world.

6 Related Work

Recursive SNARK is a hot area of research of late, they are especially useful for Blockchain use cases where the proof of the earlier block can be used as an input to the circuit of proving the later block.

Input bits ($l \cdot 32$)	Input Integers (l)	Prover time(s)	Verifier time(ms)	Proof size(kb)
base	base	5.3	33	59
320	10	5.3	35	68
640	20	5.3	36	72
960	30	5.3	37	77
1,280	40	5.3	39	82
1,600	50	5.4	40	88
1,920	60	5.4	41	94
2,240	70	5.4	42	98
2,560	80	5.4	44	104
2,880	90	5.4	45	109
3,200	100	5.4	46	114

Table 5. Performance comparison for different input numbers of a circuit with 2^{20} sequential multiplications s.t. $m_p = n$ using 93-bit prime

Early recursive SNARKs [37] [23] [10] [8] [22] built a prover for the whole SNARK circuit and then reuse this prover repeatedly. More recent recursive SNARKs are built on accumulation schemes [13] [18] [11] [17] [32] that are more scalable. One requirement for recursive SNARKs is that the protocol needs to have succinct proof size and verification time. Although our protocol performs well in practice, it is not fully succinct as the verifier runs a linear number of field operations (in $computeCircuit$ function). This linear operation part constitutes approximately 25% of the total verification cost when running a circuit with 2^{20} sequential multiplications.

Appendix

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 4. $\mathcal{S}$ uses simulator $\mathcal{S}_S$ to simulate proof transcripts for proof of knowledge (or proof of discrete logarithm, which we know for a fact that exists) protocols, and simulator $\mathcal{S}_p$ to simulate proof transcripts for polynomial commitment evaluation function PolyCommitEval (also known to exist).

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 $PK_{\kappa\mu}$ and random transcripts for $tr_{\kappa\mu}$ and $tr_{\alpha_t\mu}$, 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 $\mathcal{X}_s$ to extract witnesses from proof of knowledge transcripts and extractor $\mathcal{X}_p$ to extract witnesses from polynomial commitments (we know for a fact that there are various types of extractors $\mathcal{X}_s$ and $\mathcal{X}_p$ exist).

We validate the soundness of Protocol 1 in three steps. 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 $r, \epsilon, \vec{\tau}$ for relation 4, or discovers a non-trivial discrete logarithm relation among $g, h, \vec{u} \in \mathbb{G}$. Next, we show that the extractor $\mathcal{X}$ either extracts witnesses $\vec{a}, \vec{v}$ s.t. $\vec{v}$ maps to $\vec{\alpha}$ or discovers a non-trivial discrete logarithm relation among $g, h, u \in \mathbb{G}$. Finally, we validate the proof by checking if $r, \epsilon, \vec{\tau}$ can be computed from witnesses $\vec{a}, \vec{\alpha}$.

In step one, extractor $\mathcal{X}$ interacts with the prover in the same way as any verifier would and receives $C,\ R$ from the prover. The extractor $\mathcal{X}$ then generates a challenge x_1 and forwards it to the prover. After receiving $\vec{a}'_1, y_1, tr_\epsilon$, the extractor rewinds the prover and sends another challenge x_2 to retrieve $\vec{a}'_2, y_2, tr_\epsilon$. 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, ϵ 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. We have now retrieved witnesses $r, \epsilon, \vec{\tau}$ using the prover committed values C, R, tr_ϵ. We also know o is computed from $\vec{a}, \vec{\alpha}$ and evaluation point x since:

$$o = r + \epsilon \cdot x + \sum_{i=1}^{n} \tau_i \cdot x^{i+1} \tag{184}$$

Which is true except for an acceptable probability (soundness error).

In the second step, we validate if witnesses of $\vec{a}'$ $(\vec{a}, \vec{\alpha})$ used in computing o maps to $\vec{a}, \vec{v}$ in $\vec{P}$ by checking if we can extract these witnesses. With $\vec{a}'_1$ and $\vec{a}'_2$, we can trivially retrieve $\vec{a}, \vec{\alpha}$ since for all $i = \{1, ..., l\}$ we have:

$$a'_{1_i} - a'_{2_i} = \alpha_i (x_1 - x_2) \tag{185}$$

We then extracts witnesses $\vec{a}, \vec{v}$ using $\vec{a}'$ and input commitments $\vec{P}$. The extractor first generates k_1 and then follows the protocol to get $PK_{\kappa\mu_1}, tr_{\kappa\mu_1}, tr_{\alpha_t\mu_1}$ 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 $k_2, ..., k_{l+1}$. (The extractor also retrieves blinding keys $\vec{\mu}$ in the process, which is also validated with extractor $\mathcal{X}_s$). Through interpolation technique the extractor retrieves $(\alpha_i - \upsilon_i)$ and α_i for i in $\{1, ..., l\}$. With this information, we can now trivially compute $\vec{v}$ and verify if they can be mapped $\vec{P}$ s.t. $P_i = g^{a_i} h^{\upsilon_i}$ unless we found a non-trivial relationship among generators g, h.

In the final step, we must be able to re-compute witnesses $r, \epsilon, \vec{\tau}$ from $\vec{a}, \vec{\alpha}$ for equality 184 to be true except for an acceptable probability or we found a non-trivial relationship among generators $g, h, \vec{u}$. We can therefore conclude Protocol 1 has computational witness extended emulation.

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 4. $\mathcal{S}$ uses simulator $\mathcal{S}_S$ to simulate proof transcripts for proof of knowledge (or proof of discrete logarithm) protocols.

The simulator $\mathcal{S}$ generates random group elements to represent $\vec{S}, \vec{T}, \vec{C}, B, R$. After receiving challenge x from the verifier, the simulator generates l random integers to represent $\vec{e}$, l random integers to represent $\vec{a}'$, m_b random integers to represent breakers $\vec{y}$, and the proof of knowledge transcript $tr_{\epsilon,\phi}$. The simulator then sends them to the verifier.

The simulator follows the protocol to compute r' and $PK_{\epsilon,\phi}$, then the simulator $\mathcal{S}$ calls the simulator $\mathcal{S}_S$ to interact with the verifier to randomly generate all the necessary transcripts to prove it knows the value of ϵ and ϕ.

Next, simulator $\mathcal{S}$ simulates transcripts proving the mapping from $\vec{a}'$ to $\vec{P}$. After challenge k is received from the verifier, the simulator randomly generates tr_{υ_t} and then follows the protocol to compute PK_{υ_t}. In the final step, the simulator$\mathcal{S}$ calls the simulator $\mathcal{S}_S$ to prove the knowledge of υ_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 $\mathcal{X}$ that also use extractor $\mathcal{X}_s$ to extract witnesses from proof of knowledge transcripts. Like we did for Protocol 1, we validate the soundness of protocol 2 in three steps. First, we show how to construct an extractor $\mathcal{X}$ for protocol 2 s.t. on input $\vec{P}, R$, it either extracts witnesses $r, \epsilon, \vec{\tau}$ for relation 4, or discovers a non-trivial discrete logarithm relation among $g, h, \vec{u} \in \mathbb{G}$. Second, we show that the extractor $\mathcal{X}$ either extracts witnesses $\vec{a}, \vec{\upsilon}$ s.t. $\vec{\upsilon}$ maps to $\vec{\alpha}$ or discovers a non-trivial discrete logarithm relation between $g, h, \vec{u} \in \mathbb{G}$. Third, check if $r, \epsilon, \vec{\tau}$ can be computed from witnesses $\vec{a}, \vec{\alpha}$.

In the first step, the extractor $\mathcal{X}$ interacts with the prover in protocol 2 and receives $\vec{S}$, $\vec{T}$, $\vec{C}$, B, R from the prover. The extractor $\mathcal{X}$ then generates at least $b+3$ challenges $\vec{x}$ and forwards them to the prover. After receiving $\vec{e}_1, \vec{a}'_1, \vec{y}_1$, the extractor rewinds and repeats this step $b+2$ times to retrieve $\vec{e}_2, ..., \vec{e}_{b+3}$, $\vec{a}'_2, ..., \vec{a}'_{b+3}$, $\vec{y}_2, ..., \vec{y}_{b+3}$, and $tr_{\epsilon,\phi}$. The extractor then follows the protocol and calls extractor $\mathcal{X}_s$ to extract ϵ, ϕ from $tr_{\epsilon,\phi}$ and computes $PK_{\epsilon,\phi}$ from public inputs (R, r'). With any two challenges x_i, x_{i+1}, we can trivially retrieve r, ϵ since $r' = r + x \cdot \epsilon$, which must match the witness r, ϵ retrieved from $R = g^r h^\epsilon u^\phi$ and PK_ϵ using extractor $\mathcal{X}_s$ except with a negligible probability or discover a non-trivial discrete log relation among generators $g, h, u \in \mathbb{G}$. With challenges $x_1, ..., x_{b+3}$ and evaluation (breaker) sets $\vec{y}_1, ..., \vec{y}_{b+3}$, we apply Lagrange polynomial interpolation to retrieve witnesses $\vec{\tau}_1, ..., \vec{\tau}_{m_b}$ and $\vec{\beta}$, coefficients of commitments $\vec{C}, B$. which must be computed from $\vec{a}, \vec{\alpha}$ s.t. $\vec{\alpha}$ maps to blinding keys $\vec{\upsilon}$ except for negligible probability.

In the second step, we validate if witnesses $\vec{a}, \vec{\alpha}$ of $\vec{a}'$ used in computing o map to witnesses $\vec{a}, \vec{\upsilon}$ of $\vec{P}$ by checking if we can extract these witnesses and that $\vec{\alpha}$ map to $\vec{\upsilon}$. The extractor first generates k_1 and then follows the protocol to get $tr_{\upsilon_{t1}}, PK_{\upsilon_{t1}}$, then calls the extractor $\mathcal{X}_s$ to retrieve υ_{t1} . 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 $\{1, ..., l\}$. Dividing dividing P_i by h^{υ_i} we will get:

$$P_i / h^{\upsilon_i} = g^{a_i} \tag{186}$$

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 $\vec{a}, \vec{\alpha}$ for all $i = \{1, ..., l\}$ as equality 185 shows. Each υ_i must map to each α_i and commitments S_i, T_i for equality 67 to be true except for a negligible probability. This also implies that each a_i must be the exponent of g in equality 186 or we find a non-trivial relationship among generators g, h.

Finally, we validate the proof by checking if $r, \epsilon, \vec{\tau}$ can be computed from witnesses $\vec{a}, \vec{\alpha}$. This must be true for equality 184 to be true except for an acceptable probability (soundness error) or we find a non-trivial relationship among generators $g, h, \vec{u}$. We can therefore conclude protocol 2 has computational witness extended emulation.

C. Extended Schnorr Protocol

An example implementation of Protocol ProveEDL and VerifyEDL:

$Input: (g_1, g_2, PK \in \mathbb{G}; \alpha, \beta \in \mathbb{Z}_p)$

$\mathcal{P}'s\ input: (g_1, g_2, PK \in \mathbb{G}; \alpha, \beta \in \mathbb{Z}_p)$

$\mathcal{V}'s\ input: (g_1, g_2, PK \in \mathbb{G})$

$\mathcal{P}\ computes:$

$\delta, \epsilon \xleftarrow{\$} \mathbb{Z}_p$

$R = (g_1)^{\delta} \cdot g_2^{\epsilon} \in \mathbb{G}$

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

$\mathcal{V}\ computes:$

$c \xleftarrow{\$} \mathbb{Z}_p$

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

$\mathcal{P}\ computes:$

$s_1 = \alpha \cdot c + \delta \in \mathbb{Z}_p$

$s_2 = \beta \cdot c + \epsilon \in \mathbb{Z}_p$

$\mathcal{P} \rightarrow \mathcal{V}: s_1, s_2$

$\mathcal{V}\ validates:$

$\boldsymbol{if}\ \ PK^c \cdot R \stackrel{?}{=} g_1^{s_1} \cdot g_2^{s_2} \in \mathbb{G}$

$\boldsymbol{return}\ true$

$\boldsymbol{else\ return}\ false$

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. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Cryptology ePrint Archive, Report 2014/595 (2014), `https://eprint.iacr.org/2014/595`

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. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKS and proof-carrying data. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC. pp. 111–120. ACM Press, Palo Alto, CA, USA (Jun 1–4, 2013). https://doi.org/10.1145/2488608.2488623
11. Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo infinite: Proof-carrying data from additive polynomial commitments. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part I. LNCS, vol. 12825, pp. 649–680. Springer, Heidelberg, Germany, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84242-$0_2$3
12. 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
13. Bowe, S., Grigg, J., Hopwood, D.: Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021 (2019), `https://eprint.iacr.org/2019/1021`
14. 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
15. 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
16. 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
17. Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part I. LNCS, vol. 12825, pp. 681–710. Springer, Heidelberg, Germany, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84242-$0_2$4
18. Bünz, B., Chiesa, A., Mishra, P., Spooner, N.: Recursive proof composition from accumulation schemes. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 1–18. Springer, Heidelberg, Germany, Durham, NC, USA (Nov 16–19, 2020). https://doi.org/10.1007/978-3-030-64378-2_1
19. 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
20. 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
21. 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
22. Chiesa, A., Ojha, D., Spooner, N.: Fractal: Post-quantum and transparent recursive proofs from holography. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 769–793. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-$1_2$7
23. Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: Yao, A.C.C. (ed.) ICS 2010. pp. 310–331. Tsinghua University Press, Tsinghua University, Beijing, China (Jan 5–7, 2010)
24. 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
25. 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

26. 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
27. 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)
28. 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
29. Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-SNARKs. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 698–728. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2018). https://doi.org/10.1007/978-3-319-96878-$0_2$4
30. Heath, D., Kolesnikov, V.: Stacked garbling for disjunctive zero-knowledge proofs. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 569–598. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45727-$3_1$9
31. Kiayias, A., Tang, Q.: How to keep a secret: leakage deterring public-key cryptosystems. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 943–954. ACM Press, Berlin, Germany (Nov 4–8, 2013). https://doi.org/10.1145/2508859.2516691
32. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 359–388. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 15–18, 2022). https://doi.org/10.1007/978-3-031-15985-$5_1$3
33. 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
34. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-OLE: Improved constructions and implementation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1055–1072. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3363228
35. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 704–737. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2020). https://doi.org/10.1007/978-3-030-56877-$1_2$5
36. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), https://eprint.iacr.org/2020/1275
37. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 1–18. Springer, Heidelberg, Germany, San Francisco, CA, USA (Mar 19–21, 2008). https://doi.org/10.1007/978-3-540-78524-8_1
38. 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
39. Wahby, R.S., Tzialla, I., shelat, a., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. In: 2018 IEEE Symposium on Security and Privacy. pp. 926–943. IEEE Computer Society Press, San Francisco, CA, USA (May 21–23, 2018). https://doi.org/10.1109/SP.2018.00060
40. 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
41. 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
42. 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
43. 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