Input Transformation Based Zero-Knowledge Argument System for Arbitrary Circuits

Frank Y.C. Lu

YinYao Inc.

Abstract. We introduce a new efficient, transparent, interactive zero-knowledge argument system that is based on the new input transformation concept that we will introduce in this paper. The core of this concept is a mechanism that converts input parameters into a format that can be processed directly by the circuit so that the circuit output can be verified through direct computation of the circuit.
Our benchmark result shows our approach can significantly improve both prover runtime and verifier runtime performance by either close to or more than one order of magnitude over the state of the art while keeping the communication cost competitive with that of the state of the art. Specifically, when processing an deep circuit of 2^{20} sequential multiplication gates with 960 input bits on a single CPU thread, the performance of the BinaryBoost version of our protocol is: 0.8 seconds for the prover runtime cost; 17 milliseconds for the verifier runtime cost; and 55 kilobytes for the communication cost.
Our approach also allows our protocol to be memory-efficient without forcing it to require a designated verifier. The theoretical memory cost of our protocol is $\leq O(m_p{}^{\frac{1}{2}})$ without requiring a designated verifier.

1 Introduction

Ever since the discoveries of interactive proofs (IPs) [22] 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.

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. [23], and Maller et al. improved the SRS size from quadratic to linear in Sonic [26]. More recently developed protocols such as PLONK [20], MARLIN [15] 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 [14], 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 [31], Virgo [36]; MPC-in-the-head class of Kushilevitz, Ostrovsky, and Sahai such as ZKBoo [21] and Ligero/Ligero++ [1] [8] 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 [13]. 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 [25] [19] [24] and Vector Oblivious Linear Evaluation (VOLE) protocols [10] [28] [12] [11] [35] [32] [6] [34] 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 [29] and later Lakonia [30] 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.

There were past attempts that somewhat enabled verifiers to "execute" each multiplication gate on its own, such as Cramer and Damgård [16] and more recent designated-verifier (which is a limitation itself) VOLE (LPKZ in particular) [18] [34] [17] [33] based protocols. In these older strategies, each multiplication gate computation is actually not computed but "confirmed" by the verifier using transcripts tight to each multiplication gate. As a result, the communication/verifier costs of these earlier protocols are generally linear with the number of multiplication gates in a circuit.

On the other hand, the input transformation technique introduced by our protocol allows verifiers to use transformed inputs to directly (one operation for each operation, like we do with clear text data) compute the circuit (important), and the verifier computed output is still bound to the challenge x. This is a first and brings us three direct benefits; 1) After "computing" the whole circuit using transformed inputs, the verifier can now validate sub-linear sized proof transcripts in sub-linear runtime. 2) Since the whole circuit is linearly/directly computed by the verifier, we can break a large circuit into several smaller sub-circuits, minimizing the memory footprint to that of a sub-circuit. 3) Because the circuit is linearly "computed" by both the prover and the verifier, it gives the developer the power to significantly reduce the size of the circuit by combining "other" protocols in the middle and bypassing the "inactive" part of the circuit logic.

In our protocol, we begin by transforming each committed input parameter in $\mathbb{G}$ into its linear polynomial form in $\mathbb{Z}_q$. For 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 $a'_1, a'_2, a'_3 \in \mathbb{Z}_q$ s.t. $a'_i = a_i + X\alpha_i \in \mathbb{Z}_q$ (α_i is its blinding key). Since the transformed inputs are in $\mathbb{F}$, the verifier can plug these values directly into the circuit and just "execute" them to get the output o e.g. $a_1'^d + a_2'^d + a_3'^d = o$. The circuit output $o \in \mathbb{Z}_p$ is the evaluation at point x of a degree d polynomial s.t. $f(x) = o$. The constant term of this polynomial is the circuit output r and all other d coefficients are its blinding keys. If the prover can prove 1) it knows all coefficients of the output polynomial before the evaluation point is given (e.g., using a polynomial commitment) 2) all input transformations are legit, then we say the proof is legit.

The output polynomial in the example above has a degree of d because the transformed inputs (linear polynomials) are of degree 1. Taking to its dth power will give a polynomial with a degree of d. So if the circuit is something like ${a_1}^3 + {a_2}^3 + {a_3}^3 +, ..., + {a_t}^3 = r$, the degree of the output polynomial is 3 regardless of the value of t. Throughout our paper, we use the symbol m_p (short for "multiplication path") to denote the maximum number of multiplications included in the path that leads to the circuit output, which is one less than the degree of the output polynomial (e.g. if the degree of the output polynomial is 3, then $m_p = 2$). m_p is different from "multiplication depth". For example, for a circuit ${a_1}^3 \cdot {a_2}^5 + {a_3}^6 = r$, $m_p = 7$ (${a_1}^3 \cdot {a_2}^5 =$ a polynomial of degree 8), which is bigger than the multiplication depth (5) but smaller than the total number of multiplications (12).

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 deep circuit of 2^{20} sequential multiplication gates ($m_p = n$) with 960 input bits on a single CPU thread, the performance of the BinaryBoost version of our protocol is: 0.8 seconds for the prover runtime cost; 17 milliseconds for the verifier runtime cost; and 55 kilobytes for the communication cost. To the best of our knowledge, our protocol offers the best prover/verifier runtime performance in literature (transparent/non-interactive/high-depth protocols) by a large margin.

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

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.

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

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

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.

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

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

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

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

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

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

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

The zero-knowledge property requires that the verifier doesn't learn anything about the witness from its interaction with an honest prover.

Definition 4. (Perfect Special Honest Verifier Zero Knowledge for Interactive Arguments) An interactive proof is $(\boldsymbol{Setup}, \mathcal{P}, \mathcal{V})$ is a perfect special honest verifier zero knowledge (PHVZK) argument of knowledge for $\mathcal{R}$ if there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that all 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 the adversary chooses a distribution over statements x and witnesses w but is not able to distinguish between the simulated transcripts and the honestly generated transcripts for a valid statement/witnesses pair, and that the simulator has access to the randomness used by the verifier.

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

2.3 Polynomial Commitment Function

As in the case of other popular zero-knowledge protocols that offer succinct proof size, our protocol uses a polynomial commitment evaluation protocol to construct most of our proof transcript. Our protocol uses a version of the polynomial commitment scheme defined by Bootle. et al. [9]. The polynomial commitment function PolyCommitEval is defined as:

- $\boldsymbol{PolyCommitEval}(C, y, x; \vec{\tau}, \phi)$ $\to$ *boolean* $C = u_1^{\tau_1} u_2^{\tau_2}, ..., u_n^{\tau_n}$ is the committed polynomial (vector commitment) in $\mathbb{G}$ where $\vec{u}$ are generators, $\vec{\tau}$ are its coefficients and ϕ is its blinding key. The function returns a boolean value"true" if it can be correctly evaluated at point x s.t. $y = f(x)$.

The polynomial commitment scheme we use in our benchmark testing is based on the one defined by Bootle et al. [9].

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 $s = g^{\kappa} \in \mathbb{G}$. We define the relation for this protocol as $\mathcal{R}_{PoD} = \{(h, s; \kappa) : s = g^{\kappa}\}$. We also define two functions ($\boldsymbol{ProveDL}, \boldsymbol{VerifyDL}$) for provers and verifiers to create and verify proof transcripts:

- $\boldsymbol{ProveDL}(g, \kappa)$ $\to tr_{\kappa}$ generates the proof transcript tr_{κ}, where κ is the witness.
- $\boldsymbol{VerifyDL}(g, s, tr_{\kappa})$ $\to b \in \{0, 1\}$ 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 [27]. We know for a fact that Schnorr'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 than generators are denoted by capital letters. e.g., $C = u_1^{a_1} u_2^{a_2} ... u_n^{a_n} \in \mathbb{G}$ is a commitment committed to a vector $\vec{a}$ denoted by a capital letter, and $B \in \mathbb{G}$ is a random group element also denoted by a capital letter. For generators used as base points to compute other group elements in our protocol, such as $\vec{g}, h \in \mathbb{G}$, we use lower case letters to denote them. Greek letters are used to label hidden key values. e.g., υ is the blinding key for Pedersen commitment P on generator $h \in \mathbb{G}$ s.t. $P = g^a h^{\upsilon}$. Finally, we use standard vector notation $\vec{v}$ to denote vectors. i.e., $\vec{a} \in \mathbb{Z}_p^n$ is a list of n 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.

3 Protocol for Arbitrary Circuits

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

$$\begin{aligned}\{(g, h, \vec{u}, \vec{P}, R \in \mathbb{G}, E \in \mathcal{C}_{\mathbb{F}}\,;\, \vec{a}, \vec{v}, r, \phi \in \mathbb{Z}_q) :\ E(\vec{a}) = r \\ \wedge\ P_i = g^{a_i} h^{v_i} \forall_i \in [1, |\vec{P}|] \wedge\ R = g^r h^{\phi}\}\end{aligned} \tag{1}$$

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

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

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

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

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

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

In the next two sub-sections, we explain our protocol in two steps: In the first step, we introduce a sub-protocol (Protocol InputMapping) that allows the prover to prove each input in $\mathbb{G}$ is correctly transformed to that in $\mathbb{Z}_q$; In the second step, we introduce the full protocol (Protocol Baseline) that uses the aforementioned sub-protocol to validate transformed inputs and prove the circuit output is correctly computed from circuit inputs as relation 1 states.

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

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

$$\begin{aligned}\{(g, h \in \mathbb{G}, \vec{P} \in \mathbb{G}^l, \vec{a}\,', \in \mathbb{Z}_q^l\,;\, \vec{a}, \vec{\alpha} \in \mathbb{Z}_q^l, \vec{v} \in \mathbb{Z}_p^l) : \\ P_i = g^{a_i} h^{v_i}\ \wedge\ a_i' = a_i + X\alpha_i\ \wedge\ \forall_i \in [1, l]\}\end{aligned} \tag{4}$$

An important requirement for input transformation is that we need to transform the hidden value in Pedersen commitment to a prime field that is friendly to NTT. When multiplying two polynomials of degree m_p, the trivial approach to compute the resultant polynomial's coefficients would require a runtime cost of $O({m_p}^2)$, whereas the NTT based approach would reduce that to $O(m_p \log m_p)$.

NTT 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, so we need to pick a prime q that is NTT friendly. Also note that the prime q is expected to be smaller than p because: 1) computation in p must not overflow 2) the smaller the q value in bits, the lower the communication cost.

To start, the prover commits to a new set of blinding elements to facilitate the transformation:

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

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

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

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

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.

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

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

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

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

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

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

This test also implies the transformation process explained in this section is sound since the soundness of equation 9 is trivial to prove except for a negligible probability. For example, if $a'^*_i = a^*_i +\ x\alpha_i = a_i + \delta +\ x\alpha_i$. Knowing that e_i must be a multiple of q for equation 10 to be true, we have $a'^*_i \cdot x - e_i\ = (a_i + x\alpha_i) \cdot x -\ \omega_i q = x(a_i + x\alpha_i) + x\delta$. In order for equality 8 to be true, the left side of the equality 8 must offset $x\delta$ using committed values T_i and S_i, s.t. $x\delta$ will be removed after applying challenge x to these elements (i.e., $T_i^{x^2}\ \cdot S_i$, note exponents are different powers of x), which only happens for a negligible chance of $1/q$ when the dishonest prover successfully guessed x correctly.

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 where $\omega >> s$. Since every other transcript (in $\mathbb{G}$) is trivially zero-knowledge, we say this transformation process is zero-knowledge if ω is correctly chosen. See full proof in Appendix A.

We have so far skipped the overflow problem. If $a_i + (x\ \alpha_i \bmod q)\ > q$, then we will have an overflow problem in equation 8 9 when computing $a'_i \cdot x - e_i$. To get around this, the prover simply needs to check if $a_i + (x\ \alpha_i) \bmod q$ overflows q, and subtract $q \cdot x$ from e_i if that's the case.

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

The sub-protocol for input validation is defined in two parts: Setup and Verify. We omit the challenge generation part since that is also part of the main protocol.

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

Protocol InputMapping - Setup

The verifier generates challenge x and send it to the prover. After the challenge is received, the prover generates $\vec{a}'$ and sends them to the verifier. Since this challenge generation part is shared with the main protocol, we omit it here. Next, the verify part of the protocol validates the mapping between transformed inputs in field $\mathbb{Z}_q$ to those in group $\mathbb{G}$.

```
$Input : (\vec{P}, \vec{T}, g, h \in \mathbb{G}, \vec{a}' \in \mathbb{Z}_q; \vec{a}, \vec{\alpha} \in \mathbb{Z}_q, \vec{\upsilon} \in \mathbb{Z}_p)$
    $\mathcal{P}'s\,input : (\vec{P}, \vec{T}, g, h; \vec{a}, \vec{\upsilon});\quad \mathcal{V}'s\,input : (\vec{P}, \vec{T} g, h)$
    $\mathcal{P}\ compute :$
        $e_i = ((x\alpha_i \bmod q) - x\alpha_i)x + \omega_i q$                      $i = \{1, ..., l\}$
        if $a_i + (x\alpha_i \bmod q) > q,$ then $e_i = e_i - q \cdot x$    $i = \{1, ..., l\}$
    $\mathcal{P} \rightarrow \mathcal{V} : \vec{e}, \vec{a}'$
    $\mathcal{V} \rightarrow \mathcal{P} : k \xleftarrow{\$} \mathbb{Z}_p$
    $\mathcal{P}\ compute :$
        $\upsilon_t = \sum_{i=1}^{l} \upsilon_i k^i \in \mathbb{Z}_p$
        $tr_{\upsilon_t} = ProveDL((h^x/(g^x u)), \upsilon_t)$
    $\mathcal{P} \rightarrow \mathcal{V} : tr_{\upsilon_t}$
    $\mathcal{V}\ verify\ inputs :$
        if $(e_i \bmod q) \stackrel{?}{=} 0,$ then $continue$                      $i = \{1, ..., l\}$
        else $reject$
        $PK_{\upsilon_t} = \prod_{i=1}^{l} \left( \frac{P_i^x}{g^{a'_i \cdot x - e_i} \cdot T_i^{x^2} \cdot S_i} \right)^{k^i} \in \mathbb{G}$
        if $VerifyDL((h^x/(g^{x^2} u)), PK_{\upsilon_t}, tr_{\upsilon_t}),$ then $accept$
        else $reject$
```

Protocol for InputMapping - Verify

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

3.2 The Baseline Protocol

To prove the circuit output is correctly computed from transformed inputs $\vec{a}'$, the prover needs to show it knows all coefficients of the output polynomial. For example, for a simple circuit that just outputs the sum of two inputs, the prover needs to show it knows the constant term r and the coefficient of the degree 1 term ϵ of the output polynomial :

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

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

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

Where $r = \ a_1 \cdot a_2$, $\epsilon = a_2\alpha_1 \ + a_1\alpha_2$, and $\tau = \alpha_1 \cdot \alpha_2$. We use the label "$o$" to represent the 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 that leads to the circuit output (m_p) increases.

To get the linear polynomial we need from the raw output o, the verifier needs to subtract out terms with degrees higher than one. In the multiplication circuit above, the verifier needs to eliminate the term of degree 2 to get the linear polynomial. To do so, the prover commits to τ before the challenge x is known. When the challenge x is available, the prover sends the evaluation value y to the verifier and proofs to prove $f(x) = X^2\tau = y$. The verifier can subtract y from o to get the output in linear polynomial form:

$$r' = o - y$$

We call value y a "breaker". Breaker(s) subtracts all "noises" (polynomial terms of degree higher than one) from the raw output o. The prover and the verifier can engage in a polynomial commitment evaluation protocol to confirm $f(x) = y$.

$$C = \prod_{i=1}^{m_p} u_i^{\tau_i} \in \mathbb{G} \tag{14}$$

We can compute the whole circuit in one run s.t. the protocol evaluates to one polynomial of m_p degrees as its output and use just one breaker y to get the circuit output r'. However, that would be pretty inefficient because the bigger the polynomial gets, the less efficient it would be to compute its coefficients (even with NTT). Instead, we can break a big circuit into m_b smaller sub-circuits. Each sub-circuit computes a polynomial of at most $b + 1$ degrees, which evaluates to an intermediate result r_i' s.t.:

$$r_i' = o_i - y_i \qquad \text{for} \qquad i = \{1, ..., m_b\} \tag{15}$$

Each o_i is the output of a degree $b + 1$ polynomial at evaluation point x, and each breaker y_i is the breaker for that polynomial.

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).

$$\begin{matrix} u_1^{y_1'} \\ . \\ . \\ u_{m_b}^{y_{m_b}'} \end{matrix} = \begin{pmatrix} u_1^{\tau_{1,1}} & u_1^{\tau_{1,2}} & . & . & u_1^{\tau_{1,b}} \\ . & . & . & & . \\ . & . & & . & . \\ u_{m_b}^{\tau_{m_b,1}} & u_{m_b}^{\tau_{m_b,2}} & . & . & u_{m_b}^{\tau_{m_b,b}} \end{pmatrix} \begin{pmatrix} x^2 \\ . \\ . \\ x^{b+1} \end{pmatrix}$$

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 in Bootle et al.'s polynomial commitment evaluation scheme.

$$C_j = \prod_{i=1}^{m_b} u_i^{\tau_{i,j}} \qquad \text{for} \qquad j = \{1, ..., b\} \tag{16}$$

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.

$$\prod_{j=1}^{b} C_j^{x^j} \stackrel{?}{=} \prod_{i=1}^{m_b} u_i^{y'_i} \tag{17}$$

Note that if each breaker y'_i is equal to the sum of products of b terms and each term is a product of $x^j \in \mathbb{Z}_q$ 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 $y'_i = y_i + v \cdot q$ for some v and $|v| \leq |q| + |b|$ (this is important in Protocol BinaryBoost we will cover in the next section). 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 $\vec{\mu} \in \mathbb{Z}_m^{m_b}$. Each y'_i is now computed as:

$$y'_i = \sum_{j=1}^{b} \tau_{i,j}x^j + \mu_i \qquad \text{for} \qquad i = \{1, ..., m_b\} \tag{18}$$

If $|m| >> |\sum_{j=1}^{b} \tau_{i,j}x^j|$, then $\mathbb{Z}_m$ perfectly hides $\sum_{j=1}^{b} \tau_{i,j}x^j$ (130-bit for 61-bit prime and b=2^8) with high probability. For example, if we set m to a randomly chosen 216-bit value, then there is less than a 2^{-85} probability that μ_j does not perfectly hide $\vec{\tau}_j$ (e.g. when y_i overflows m or less than 131 bits). 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:

$$y_i = (y'_i \cdot x) \bmod q \in \mathbb{Z}_q \qquad \text{for} \qquad i = \{1, ..., m_b\} \tag{19}$$

We also need to adjust the blinding key of r' by adding μ_i to each r_i. The updated equality graph is shown in figure 2 below.

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

$$\prod_{j=1}^{b} C_j^{x^j} \cdot M \stackrel{?}{=} \prod_{i=1}^{m_b} u_i^{y'_i} \tag{20}$$

If the commitments $\vec{C}, M$ and vector $\vec{y}'$ satisfy the equation above, then we know breakers $\vec{y}'$ are valid. The verifier applies equation 19 to each y'_i to get the actual breakers y_i used in computing o.

We define two more functions for our protocol. function
computeSubCircuitKeys is used by the prover to compute the keys of each sub-circuit or "row" in Figure 1, and function ***computeSubCircuit*** is used by the verifier to compute the value of a sub-circuit at the evaluation point x:

1. function ***computeSubCircuitKeys***$_i$("input values", "input keys"): for $i = \{1, .., m_b\}$, it takes input values $\vec{a}$ and keys $\vec{\alpha}$ to evaluate the sub-circuit and outputs $r_i, \epsilon_i, \vec{\tau}_i$ (coefficients of o_i).

2. function ***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 sub-circuit.

For example, if the logic of the ith sub-circuit is to return the product of l inputs, then the $computeSubCircuit_i$ function simply performs $o_i = a_1 \times a_2 \times, ..., \times a_l$. Since $a_1, ..., a_l$ are linear polynomials evaluated at point X, o_i is the evaluation of the output polynomial at point X, and the $computeSubCircuitKeys_i$ function computes all coefficients of the output polynomial. We are now ready to introduce Protocol Baseline as follows:

$$Input : (g, h, \vec{u}, \vec{P} \in \mathbb{G}, \vec{a}\,' \in \mathbb{Z}_q^l\,; \, \vec{a}, \vec{v} \in \mathbb{Z}_q) \tag{21}$$

$$\mathcal{P}'s\, input : (g, h, \vec{u}, R, \vec{a}\,'; \, \vec{a}, \vec{\alpha}, \phi); \quad \mathcal{V}'s\, input : (g, h, \vec{u}, R, \vec{a}\,') \tag{22}$$

$$\mathcal{P}\ compute : \tag{23}$$

$$\rho_i \xleftarrow{\$} \mathbb{Z}_m, \qquad\qquad i = \{1, ..., m_b\} \tag{24}$$

$$M = \prod_{i=1}^{m_b} u_i^{\mu_i} \in \mathbb{G} \tag{25}$$

$$r_1, \epsilon_1, \vec{\tau}_1 = computeSubCircuitKeys_1(\vec{a}, \vec{\alpha}); \tag{26}$$

$$\epsilon_1 = (\epsilon_1 - \mu_1) \bmod q \in \mathbb{Z}_q \tag{27}$$

$$\textbf{for} \quad i = 2, ..., m_b \quad \{ \tag{28}$$

$$\quad \vec{a}' = \vec{a}\,||\, r_{i-1}, \qquad \vec{\alpha}' = \vec{\alpha}\,||\,\epsilon_{i-1}; \tag{29}$$

$$\quad r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}\,', \vec{\alpha}'); \tag{30}$$

$$\quad \epsilon_i = (\epsilon_i - \mu_i) \bmod q \in \mathbb{Z}_q \quad \} \tag{31}$$

$$C_j = \prod_{i=1}^{m_b} u_i^{\tau_{i,j}} \in \mathbb{G} \qquad\qquad i = \{1, ..., b\} \tag{32}$$

$$\phi \xleftarrow{\$} \mathbb{Z}_p \tag{33}$$

$$R = g^{r_{m_b}} h^{\phi} \in \mathbb{G} \tag{34}$$

$$\text{InputMapping-Setup}(\vec{P}||R, g; \vec{v}||\phi) \rightarrow \mathcal{P} : \vec{S}, \vec{T} \tag{35}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{C}, M, \vec{S}, \vec{T} \tag{36}$$

$$\mathcal{V} \rightarrow \mathcal{P} : \quad x \xleftarrow{\$} \mathbb{Z}_q \tag{37}$$

$$\mathcal{P}\ compute : \tag{38}$$

$$a_i' = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \qquad\qquad i = \{1, ..., l\} \tag{39}$$

$$y_i' = \sum_{j}^{b} \tau_{i,j} \cdot x^j + \beta_i \in \mathbb{Z}_p \qquad\qquad i = \{1, ..., m_b\} \tag{40}$$

$$\mathcal{P} \rightarrow \mathcal{V} : \vec{y'}, \vec{a'} \tag{41}$$

$$\mathcal{V}\ verify\ final\ output : \tag{42}$$

$$\textbf{if } (\prod_{i=1}^{m_b} u_i^{y_i'} \overset{?}{=} \prod_{j=1}^{b} C_j^{x^j} \cdot M) \textbf{ then } continue \tag{43}$$

$$\textbf{else } reject \tag{44}$$

$$\textbf{for} \quad i = 1, ..., m_b \quad \{ \tag{45}$$

$$\quad o_i = computeSubCircuit_i(\vec{a}\,') \in \mathbb{Z}_q \tag{46}$$

```
        $y_i = (y'_i \cdot x) \bmod q \in \mathbb{Z}_q$                                        (47)
        $r'_i = o_i - y_i \in \mathbb{Z}_q$                                                      (48)
        $\vec{a}' = \vec{a}'||r' \in \mathbb{Z}_q$  }                                            (49)
    if InputMapping-Verify($\vec{P}||R, \vec{S}, \vec{T}, g, h, \vec{a}'||r'_{m_b}; \vec{a}||r_{m_b}, \vec{\alpha}||\epsilon, \vec{v}||\phi$)   (50)
        then *continue*                                                                          (51)
    else *reject*                                                                                (52)
```

Protocol Baseline

Theorem 2. *(The Baseline Protocol). The proof system presented in this section has perfect completeness, PHVZK, and CWEE. The proof for Theorem 2 is presented in Appendix B.*

3.3 Optimization Techniques and Customized Sub-Circuit

We can optimize performance through the use of specially designed sub-circuits. The idea is similar to that of "custom gates" found in SNARKs protocols in principle but very different in implementation. In our case, the goal is to utilize existing algorithms/protocols to handle operations that would otherwise be expensive in our protocol (or any other protocol).

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

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

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

An example usage is as follows:

```
computeSubCircuit : ($\vec{a}' \in \mathbb{Z}^q, \vec{C} \in \mathbb{G}$)
        $c'_1 = a'_1 - a'_2 \in \mathbb{Z}_q$
        if *Protocol RangeProof*($C_1, 0$)
            $c'_2 = a'_3 - a'_4 \in \mathbb{Z}_q$
            if *Protocol RangeProof*($C_2, 0$)
                $o$ = do something
            else
                $o$ = do something else
        else
            $o$ = do something
        if *Protocol InputMapping*($C_1||C_2, c'_1||c'_2$)  return $o$
        else  *reject*
```

Function computeSubCircuit (Customized)

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

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

We believe combining the arithmetic circuit and range-proof protocols is the most efficient way to run a zero-knowledge test in the real world. What makes our protocol unique in such an approach is that we can batch proof all comparison logics using one range proof while still keeping the main line of the circuit process inside one protocol run. If it were done in SNARK protocols, it would require cutting a large circuit into multiple smaller ones, making both the communication cost and verifier runtime cost linear to the number of smaller circuits (or the number of comparisons in a circuit).

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.

3.4 Memory Efficiency

The memory consumption cost of protocol Baseline is $O(m_p)$. We can improve the memory consumption cost of our protocol to $O(b)$ or approximately $O(m_p^{\frac{1}{2}})$.

Instead of computing $\vec{C}$ after all $\vec{\tau}$ are computed s.t. $|\vec{\tau}| = m_p$. We ask the prover to compute $\vec{C}$ iteratively. When the prover computes $r_i, \epsilon_i, \vec{\tau}_i$ in each loop for $i = \{1, ..., m_b\}$, the prover also updates $\vec{C}$ with each new "row" of coefficients $\vec{\tau}_i$. This can be achieved by replacing lines 28 - 32 with the following lines:

$$C_j = u_i^{\tau_{1,j}} \in \mathbb{G} \qquad i = \{1, ..., b\} \tag{53}$$

$$\textbf{for} \quad i = 2, ..., m_b \quad \{ \tag{54}$$

$$\vec{a}' = \vec{a}\,||\,r_{i-1}, \qquad \vec{\alpha}' = \vec{\alpha}\,||\,\epsilon_{i-1}; \tag{55}$$

$$r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{a}', \vec{\alpha}'); \tag{56}$$

$$\epsilon_i = (\epsilon_i - \mu_i) \bmod q \in \mathbb{Z}_q \quad \} \tag{57}$$

Because the coefficients of each iteration $\vec{\tau}_i$ will get discarded from the memory after each iteration completes, we can therefore achieve a memory cost of $O(b)$. However, we now have to recompute $\vec{\tau}_i$ after challenge x is available so that $\vec{y}'$ can be computed (in line 42).

The obvious caveat is that we can no longer use Pippenger acceleration to compute each C_j as we did in protocol Baseline, and we also have to recompute coefficients for each sub-circuit to get $\vec{y}$.

A meet-in-the-middle approach is to break the computation of $\vec{C}$ into tb segments, and in each segment we compute $\frac{1}{tb}m_p$ coefficients and update $\vec{C}$ after each segment. By doing so, we can achieve a consumption cost of $O(\frac{1}{tb}m_p)$.

4 The Binary-Boost Protocol for Boolean Circuits

For boolean circuits, we can leverage the fact that all input/output values of each gate can only be either 0 or 1 to construct a more efficient protocol-BinaryBoost.

4.1 Protocol for boolean circuit validation

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

$$\{(\vec{b}', \in \mathbb{Z}_q^l\,; \vec{b}, \vec{\beta} \in \mathbb{Z}_q^l) : b'_i = b_i + X\beta_i \;\wedge\; \forall_{b_i} \in [0,1] \;\wedge\; \forall_i \in [1,..,l]\} \tag{58}$$

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

$$(b'_i \cdot b'_i - b'_i) = \delta_{1,i}x + \delta_{2,i}x^2 \tag{59}$$

Where $\delta_{2,i} = \beta_i^2$, and $\delta_{1,i} = \beta_i$ when b_i is 1 and $\delta_{1,i} = -\beta$ when b_i is 0. To prove the correctness for all $b_i \in \{0,1\}$, the prover commits to polynomials D_1, D_2:

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

Where D_1 commits to coefficients on the x term and D_2 commits to coefficients on the x^2 term. The prover can easily join two polynomial commitments into one and sends only one element D to the verifier.

$$D = \prod_{i=1}^{l} u_i^{\delta_{1i}} \cdot \prod_{i=1}^{l} u_{i+l}^{\delta_{2i}} \cdot h^{\rho} \in \mathbb{G} \tag{60}$$

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

$$y_1 \cdot X + y_2 \cdot X^2 = \sum_{j=1}^{l \cdot 32} (b'_i \cdot b'_i - b'_i) \cdot k^i \tag{61}$$

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

$$a'_i = \sum_{j=1}^{32} b'_{i,j} \cdot 2^j \tag{62}$$

We define the protocol BooleanityTest using two sub-protocols:

$$\boxed{\begin{array}{ll}
\textit{Input} : (\vec{b}' \in \mathbb{Z}_q : \vec{b}, \vec{\beta}' \in \mathbb{Z}_q) & \\
\mathcal{P}\ \textit{compute} : & \\
\quad l = |\vec{b}'| \in \mathbb{Z}_q & \\
\quad \rho \xleftarrow{\$} \mathbb{Z}_q & \\
\quad \delta_{1i} = b_i\beta_i + b_i\beta_i - \beta_i \in \mathbb{Z}_q & i = \{1, ..., l\} \\
\quad \delta_{2i} = \beta_i^2 \in \mathbb{Z}_q & i = \{1, ..., l\} \\
\quad D = \prod_{i=1}^{l} u_i^{\delta_{1,i}} \cdot \prod_{i=1}^{l} u_{i+l}^{\delta_{2,i}} \cdot h^{\rho} \in \mathbb{G} &
\end{array}}$$

Protocol BooleanityTest-Setup

After a challenge x is sent from the verifier to the prover, the protocol moves to the verification stage defined by the following sub-protocol.

```
$Input : (\vec{b}' \in \mathbb{Z}_q; \rho, \vec{\delta_1}, \vec{\delta_2}' \in \mathbb{Z}_q)$
  $\mathcal{P}'s\, input : (\vec{b}'; \rho, \vec{\delta_1}, \vec{\delta_2}');\ \ \mathcal{V}'s\, input : (\vec{b}')$
$\mathcal{V} \rightarrow \mathcal{P} : \quad k \xleftarrow{\$} \mathbb{Z}_q$
$\mathcal{P}$ $compute$ :
    $y_1 = \sum_{i=1}^{l} \delta_{1,i} k^i \in \mathbb{Z}_q, \quad y_2 = \sum_{i=1}^{l} \delta_{2,i} k^i \in \mathbb{Z}_q$
$\mathcal{P} \rightarrow \mathcal{V} : y_1, y_2$
$\mathcal{P}, \mathcal{V}$ engate to evaluate :
    if $PolyCommitEval(D, y_1 + y_2 k^l, k; \vec{\delta_1} || \vec{\delta_2}, \rho)$
    $\wedge\ y_1 \cdot x + y_2 \cdot x^2 \overset{?}{=} \sum_{j=1}^{l\cdot 32} (b'_i \cdot b'_i - b'_i) \cdot k^i$
        return true
    else  return false
```

Protocol BooleanityTest-Verify

Theorem 3. *(The Booleanity Test Protocol). The proof system presented in this section has perfect completeness, PHVZK, and CWEE. The proof for Theorem 3 is presented in Appendix C.*

4.2 Batch Validate Subcircuit Rows

In a boolean circuit, we know for a fact that each r_i must be either 0 or 1 s.t.

$$o_i - y_i = r_i + X\epsilon_i \qquad s.t. \quad r_i \in [0, 1]$$

We can confirm the boolean property of $\vec{r}$ with the booleanityTest protocol. This property implies a dishonest prover can only alter each y_i by ± 1 s.t.

$$y_i^* = y_i \pm 1;$$

The equation above also implies $y_i^{*\prime} = y_i' \pm 1$ as far as our protocol is concerned since $y_i^* = (y_i^{*\prime} \cdot x) \bmod q$. Leveraging this fact, we can now batch commit each C_j (committing coefficients $\tau_{1,j}, ..., \tau_{m_b,j}$) with fewer generators by using a simple binary trick: multiply each row i by a power of 2^i.

Instead of committing $\tau_{1,j}, .., \tau_{m_b,j}$ using m_b generators $u_1, .., u_{m_b}$, the prover now uses $\frac{m_b}{tb}$ generators $u_1, .., u_{t_b}$ to commit each C_j, where tb is the size of each batch i'. Each i' covers tb "rows" (See Figure 2). Let $st = (i' - 1) \cdot tb$ we have:

$$\gamma_{i',j} = \sum_{i=1}^{tb} \tau_{st+i,j} \cdot 2^i \qquad\qquad j = \{1, ..., b\} \tag{63}$$

$$C_j = \prod_{i'=1}^{m_b/tb} u_{i'}^{\gamma_{i',j}} \in \mathbb{G} \qquad\qquad i' = \{1, ..., m_b/tb\} \tag{64}$$

This can be visualized in Figure 2. For example, the exponent of generator u_1 in C_j is $\gamma_{1,j} = \sum_{i=1}^{32} \tau_{st+i,j} \cdot 2^i$, and its (batched) blinding key is $\mu_{i'} = \sum_{i=1}^{tb} \mu_{st+i} \cdot 2^i$. After applying evaluation point x, the opening of each u_i ("row") is the summation of tb values:

$$\sum_{i=1}^{tb} y'_{st+i} \cdot 2^i = \sum_{i'=1}^{m_b/tb} (\gamma_{i',j} \cdot x^j + \mu_{i'}) \tag{65}$$

A dishonest prover cannot convince verifiers $\vec{y}'^* \neq \vec{y}'$ by committing to $\vec{\gamma}^* \neq \vec{\gamma}$ s.t. $\vec{r}'^* \neq \vec{r}'$ still passes the booleanity test. For example, let's say $y_i'^* = y_i' + d_i$ s.t. $d_i \in \{0, \pm 1\}$ and $\gamma^*_{i',j} = \gamma_{i',j} + f_{i'}$ where $f_{i'} \in \{0, \delta_i\}$. To convince $\vec{r}^*$ the dishonest prover needs to find a pair of sets $\vec{d}, \vec{f}$ that satisfies the following equality before committing $\vec{\gamma}^*$:

$$\sum_{i=1}^{tb} d_{st+i} \cdot 2^i = \sum_{i'=1}^{m_b/tb} f_{i',j} \cdot x^j \tag{66}$$

Which cannot be done without prior knowledge of x. Note that although $\vec{y}'^*$ is provided by the prover after x is known, the prover needs to set $\vec{d}$, which defines committed values $\vec{r}^*, \vec{\epsilon}^*, \vec{\gamma}^*$, before x is available. The binary trick (multiplying each "row" by 2^i) guards against the case where a dishonest prover finds a $\vec{y}'^*$ set s.t. $\sum_{i=1}^{tb} y'^*_{st+i} = \sum_{i=1}^{tb} y'_{st+i}$, in which case the dishonest prover would not need to find a set $\vec{f}$ since $\sum_{i=1}^{tb} d_{st+i} = 0$.

To protect our protocol from the overflow attack, the tb value (we default to 32) must be smaller than $|q|$, or else equation 19 would not be safe (e.g., a dishonest prover can find a set $\vec{d}$ s.t. $(\sum_{i=1}^{tb} d_{st+i}) \bmod q = 0$).

$$\begin{matrix} u_1^{y'_1 \cdot 2^1} \\ \cdot \\ u_1^{y'_{32} \cdot 2^{32}} \\ u_2^{y'_{33} \cdot 2^1} \\ \cdot \\ u_2^{y'_{64} \cdot 2^{32}} \\ u_3^{y'_{65} \cdot 2^1} \\ \cdot \\ \cdot \\ u_{m_b/Tb}^{y'_{m_b} \cdot 2^{32}} \end{matrix} = \begin{pmatrix} u_1^{\tau_{1,1} \cdot 2^1} & \cdot & & u_1^{\tau_{1,b} \cdot 2^1} \\ \cdot & & \cdot & \cdot \\ u_1^{\tau_{32,1} \cdot 2^{32}} & \cdot & \cdot & u_1^{\tau_{32,b} \cdot 2^{32}} \\ u_2^{\tau_{33,1} \cdot 2^1} & \cdot & \cdot & u_2^{\tau_{33,b} \cdot 2^1} \\ \cdot & & \cdot & \cdot \\ u_2^{\tau_{64,1} \cdot 2^{32}} & \cdot & \cdot & u_2^{\tau_{64,b} \cdot 2^{32}} \\ u_3^{\tau_{65,1} \cdot 2^1} & \cdot & \cdot & u_2^{\tau_{65,b} \cdot 2^1} \\ \cdot & \cdot & & \cdot \\ \cdot & & \cdot & \cdot \\ u_{m_b/tb}^{\tau_{m_b,1} \cdot 2^{32}} & \cdot & \cdot & u_{m_b/tb}^{\tau_{m_b,b} \cdot 2^{32}} \end{pmatrix} \begin{pmatrix} x \\ x^2 \\ x^3 \\ \cdot \\ \cdot \\ x^b \end{pmatrix} \cdot \begin{pmatrix} u_1^{\mu_1 \cdot 2^1} \\ \cdot \\ u_1^{\mu_{32} \cdot 2^{32}} \\ u_2^{\mu_{33} \cdot 2^1} \\ \cdot \\ u_2^{\mu_{64} \cdot 2^{32}} \\ u_3^{\mu_{65} \cdot 2^1} \\ \cdot \\ \cdot \\ u_{m_b/tb}^{\mu_{m_b} \cdot 2^{32}} \end{pmatrix}$$

Figure 2

The verifier can now batch commit tb "rows" by using their "binary sum" $z_{i'}$:

$$z_{i'} = \sum_{i=1}^{tb} y'_{st+i} \cdot 2^i \qquad \text{for} \qquad i' = \{1, ..., m_b/tb\} \tag{67}$$

The equation 20 is now updated to batch validate tb "rows" at once.

$$\prod_{j=1}^{b} C_j^{x^j} \cdot M \overset{?}{=} \prod_{i'}^{m_b/tb} u_{i'}^{z_{i'}} \tag{68}$$

4.3 Leveraging Batched Rows to Shrink Communication Cost

We can leverage the batch validation approach to reduce the communication cost. To do so, the prover now computes and sends $\vec{y} \in \mathbb{Z}_q$ to the verifier instead of $\vec{y}'$. Since $|y_i| \approx \frac{1}{4}|y_i'|$, sending $\vec{y}$ will save approximately $24 \times m_b$ bytes in communication cost.

The equation 19 will now be performed by the prover instead of the verifier. From equation 19, we can infer that $(y_i \cdot x^{-1}) \mod q = (y_i') \mod q$. We can therefore conclude that each y_i' can be represented as:

$$y_i' = (y_i \cdot x^{-1}) \bmod q + v_i \cdot q$$

Note that the equation above must not create an overflow in $\mathbb{Z}_p$ (e.g. $y_i' < P$). Rearranging the equation above, we get an equation that computes v_i:

$$v_i = (y_i' - (y_i') \bmod q)/q$$

Since we want to prove the committed value in batch, we need $v_{i'} = \sum_{i=1}^{tb} v_{st+i} \cdot 2^i$ instead, which we can compute directly from:

$$v_{i'} = (\sum_{i=1}^{m_b/tb} (y'_{st+i} - (y'_{st+i}) \bmod q) \cdot 2^i)/q \tag{69}$$

Since the prover now passes $\vec{y}, \vec{v}$ instead of $\vec{y}'$ to the verifier, the verifier needs to compute $z_{i'}$ with the following equation instead of equation 67:

$$z_{i'} = (\sum_{i=1}^{m_b/tb} (y_{st+i} \cdot x^{-1}) \bmod q \cdot 2^i) + v_{i'} \cdot q \tag{70}$$

Lastly, $v_{i'}$ must not overflow p because it can be used as an attacking surface:

$$z_{i'} < p \tag{71}$$

We now define the updated protocol for the boolean circuit - BinaryBoost:

$$Input : (g, h, \vec{u}, \vec{P} \in \mathbb{G}, \vec{a}' \in \mathbb{Z}_q^l \,;\, \vec{a}, \vec{v} \in \mathbb{Z}_q) \tag{72}$$

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

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

$$\mu_i \xleftarrow{\$} \mathbb{Z}_m, \qquad i = \{1, ..., m_b\} \tag{75}$$

$$\mu_{i'} = \sum_{i=1}^{tb} \mu_{(i'-1)tb+i} \cdot 2^i \qquad i' = \{1, ..., m_b/tb\} \tag{76}$$

$$M = \prod_{i=1}^{m_b} u_i^{\mu_{i'}} \in \mathbb{G} \tag{77}$$

$$b_i \in \mathbb{Z}_2\,, \quad \beta_i \xleftarrow{\$} \mathbb{Z}_q \qquad i = \{1, ..., l \cdot 32\} \tag{78}$$

$$r_1, \epsilon_1, \vec{\tau}_1 = computeSubCircuitKeys_1(\vec{b}, \vec{\beta}); \tag{79}$$

$$\epsilon_1 = (\epsilon_1 - \mu_1) \bmod q \in \mathbb{Z}_q \tag{80}$$

$$\textbf{for} \quad i = 2, ..., m_b \quad \{ \tag{81}$$

$$\vec{b} = \vec{b} \,||\, r_{i-1}, \qquad \vec{\beta} = \vec{\beta} \,||\, \epsilon_{i-1}; \tag{82}$$

```
        $r_i, \epsilon_i, \vec{\tau}_i = computeSubCircuitKeys_i(\vec{b}, \vec{\beta});$   (83)
        $\epsilon_i = (\epsilon_i - \mu_i) \bmod q \in \mathbb{Z}_q \quad \}$   (84)
    $\phi \xleftarrow{\$} \mathbb{Z}_p$   (85)
    $R = g^r h^\phi \in \mathbb{G}$   (86)
    $\text{InputMapping-Setup}(\vec{P}||R, g; \vec{v}||\phi) \rightarrow P : \vec{S}, \vec{T}$   (87)
    $\textbf{for} \quad i' = 1, ..., m_b/tb \quad \{$   (88)
        $st = (i' - 1) \cdot tb$   (89)
        $\gamma_{i',j} = \sum_{i=1}^{tb} \tau_{st+i,j} \cdot 2^{i-1} \in Z_p \qquad j = \{1, ..., b\}$   (90)
        $C_{i',j} = u_{i'}^{\gamma_{i',j}} \in \mathbb{G} \qquad j = \{1, ..., b\} \quad \}$   (91)
    $C_j = \prod_{i'=1}^{m_b/tb} C_{i',j} \in \mathbb{G} \qquad i' = \{1, ..., m_b/tb\}$   (92)
    $\text{Booleanity-Setup}(\vec{b}||\vec{r}, \vec{\beta}||\vec{\epsilon}) \rightarrow \mathcal{P} : D, \rho, \vec{\delta_1}, \vec{\delta_2}$   (93)
$\mathcal{P} \rightarrow \mathcal{V} : \vec{C}, M, D, \vec{S}, \vec{T}$   (94)
$\mathcal{V} \rightarrow \mathcal{P} : x \xleftarrow{\$} \mathbb{Z}_q$   (95)
$\mathcal{P} \ compute :$   (96)
    $a'_i = a_i + x \cdot \alpha_i \in \mathbb{Z}_q \qquad i = \{1, ..., l\}$   (97)
    $y'_i = \sum_{j}^{b} \tau_{i,j} \cdot x^j + \beta_i \in \mathbb{Z}_p \qquad i = \{1, ..., m_b\}$   (98)
    $y_i = (y'_i \cdot x) \bmod q \in \mathbb{Z}_q \qquad i = \{1, ..., m_b\}$   (99)
    $\textbf{for} \quad i' = 1, ..., m_b/tb \quad \{$   (100)
        $st = (i' - 1) \cdot tb$   (101)
        $v_{i'} = ( \sum_{i=1}^{m_b/tb} (y'_{st+i} - (y'_{st+i}) \bmod q) \cdot 2^i)/q \quad \}$   (102)
$\mathcal{P} \rightarrow \mathcal{V} : \vec{a}', \vec{b}', \vec{y}, \vec{v}$   (103)
$\mathcal{V} \ verify \ final \ output :$   (104)
    $\textbf{for} \quad i' = 1, ..., m_b/tb \quad \{$   (105)
        $st = (i' - 1) \cdot tb$   (106)
        $z_{i'} = ( \sum_{i=1}^{m_b/tb} (y_i \cdot x^{-1}) \bmod q \cdot 2^i) + v_{i'} \cdot q \quad \}$   (107)
    $\textbf{if} \ ( \prod_{i'}^{m_b/tb} u_{i'}^{z_{i'}} \overset{?}{=} \prod_{j=1}^{b} C_j^{x^j} \cdot M) \ \textbf{then} \ continue$   (108)
    $\textbf{else} \ reject$   (109)
    $\textbf{for} \quad i = 1, ..., m_b \quad \{$   (110)
        $o_i = computeSubCircuit_i(\vec{b}') \in \mathbb{Z}_q$   (111)
        $r'_i = o_i - y_i \in \mathbb{Z}_q$   (112)
        $\vec{a}' = \vec{a}'||r'_i \in \mathbb{Z}_q \quad \}$   (113)
```

$$\textbf{if}\ \text{InputMapping-Verify}(\vec{P}||R, \vec{S}, \vec{T}, g, h, \vec{a}'||r'; \vec{a}||r, \vec{\alpha}||\epsilon, \vec{v}||\phi) \tag{114}$$

$$\wedge\ \text{Booleanity-Verify}(\vec{b}'; \rho, \vec{\delta_1}, \vec{\delta_2}') \tag{115}$$

$$\wedge\ a'_i = \sum_{j=1}^{32} b'_{i,j} \cdot 2^j \qquad i = \{1, ..., l\} \tag{116}$$

$$\textbf{then}\ \mathit{accept} \tag{117}$$

$$\textbf{else}\ \mathit{reject} \tag{118}$$

Protocol BinaryBoost

Theorem 4. *(Input Transformation Protocol with Binary Boost). The proof system presented in this section has perfect completeness, PHVZK, and CWEE. The proof for Theorem 4 is presented in Appendix D.*

4.4 The Asymptotic Cost

The prover runtime of our baseline protocol (Protocol Baseline) is dominated by $O(\iota\, m_p \log m_p + l)$ field operations and $O(m_p + l)$ group exponentiations. The value of ι depends on how the circuit is wired. For the sequential multiplication test case that we benchmark against, $\iota = m_b \sum_{i=1}^{\log b} i$; the verifier runtime is dominated by $O(n + m_p^{\frac{1}{2}} + l)$ field operations and $O(m_p{}^{\frac{1}{2}} + l)$ group exponentiations; and the communication cost is dominated by $O(m_p{}^{\frac{1}{2}} + l)$ group elements and $O(m_p{}^{\frac{1}{2}} + l)$ field elements.

The binary-boost protocol improves the asymptotic group exponentiation cost of the prover runtime to $O(\iota\, m_p \log m_p + l)$ field operations and $O(\frac{1}{tb} m_p + m_p{}^{\frac{1}{2}} + l)$ group exponentiations, where tb is set to 32. While the asymptotic cost of the communication cost is the same, the concrete communication cost is reduced by approximately 20%, which is achieved by sending the smaller $\vec{y}$ instead of the larger $\vec{y}'$.

Our protocol is also natively faster than its asymptotic cost indicates because group exp. operations of our protocol operate mostly in q (61-bits or 93-bits), which is significantly smaller than p in ECC. This gives us approximately 2-2.5X performance gains when performing multi-exponentiations. If the circuit is shallow (e.g., for a circuit with the final output $r = \sum_{i=1}^{n} a \cdot b$ where 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 inexpensive $O(n/2)$ field addition operations.

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 5241902353849032101525979137, 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 $m_b = 2^{14}$.

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.

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
Baseline(61 bit)	0.008	0.02	0.05	0.18	0.7	2.5
Baseline(93 bit)	0.06	0.12	0.3	0.6	1.7	4.9
B-Boost(61 bit)	0.01	0.01	0.02	0.06	0.2	0.8

Table 1. Prover performance comparison (seconds)

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
Baseline($61bit$)	16	17	21	27	39	65
Baseline($93bit$)	20	22	26	33	48	77
B-Boost($61bit$)	15	16	19	24	34	55

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.

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
Baseline($61bit$)	2	2	3	4	7	16
Baseline($93bit$)	5	5	6	8	15	37
B-Boost($61bit$)	2	2	3	4	6	17

Table 3. Verifier performance comparison (milliseconds)

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

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 Conclusion

We believe the best way to run our protocol in the real world is to use protocol Baseline to run an arithmetic circuit and embed an existing range-proof protocol to perform comparison operations as explained in section 3.3. We also present an ultra-efficient boolean circuit protocol BinaryBoost just in case a boolean circuit is necessary.

Appendix

A. Proof for Theorem One

Proof. Perfect completeness follows from the fact that protocol InputMapping Validation is trivially complete.

To prove PHVZK for relation 4, we define a simulator $\mathcal{S}_{input}$. To start, simulator $\mathcal{S}_{input}$ randomly generates $\vec{S}, \vec{T}$ and sends them to the verifier. After receiving challenge x from the verifier, the simulator randomly generates and sends $\vec{a}\,', \vec{e}$ according to the protocol specification (s.t. $(e_i) \bmod q = 0$) and sends them to the verifier.

When challenge k is received, the simulator $\mathcal{S}_{input}$ calls simulator $\mathcal{S}_{dlog}$ to generate transcript tr_{υ_t} and send it to the verifier.

The verifier then follows the protocol to compute PK_{υ_t} using transcripts $\vec{S}, \vec{T}, \vec{a}\,', \vec{e}$ and calls the $VerifyDL$ function to test it against the input transcript tr_{υ_t}, We already know for a fact that simulator $\mathcal{S}_{dlog}$ can extract witnesses from any discrete-log relation and that simulators $\mathcal{S}_{input}$ and $\mathcal{S}_{dlog}$ choose all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or compute them directly as described in the protocol. Since all elements in proof transcripts are either independently randomly distributed or their relationship is fully defined by the verification equations, we can conclude that protocol InputMapping is PHVZK.

To prove CWEE, we construct an extractor $\mathcal{X}$ that also uses extractor $\mathcal{X}_{dlog}$ to extract witnesses from proof of knowledge transcripts. To start, the extractor $\mathcal{X}$ interacts with the prover and receives $\vec{S}$, $\vec{T}$ from the prover. The extractor $\mathcal{X}$ then generates a challenge x_1 and forwards it to the prover. After receiving $\vec{e}_1, \vec{a}'_1$, the extractor rewinds and repeats this step with another challenge x_2 to retrieve $\vec{e}_2$, $\vec{a}'_2$.

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

Using any two different challenges x_i, x_{i+1} we mentioned earlier, the extractor gets $\vec{a}'_1$ and $\vec{a}'_2$ from the prover, which we can trivially retrieve $\vec{\alpha}$ for all $i = \{1, ..., l\}$ using the equation below.

$$a'_{1_i} - a'_{2_i} = \alpha_i(x_1 - x_2) \tag{119}$$

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

$$(h^x/(g^{x^2}u))^{\sum_i^l \upsilon_i \cdot k_i} = PK_{\upsilon_t} = \prod_{i=1}^{l} \left(\frac{g^{a_i \cdot x} h^{\upsilon_i \cdot x}}{g^{a'_i \cdot x - e_i + (\upsilon_i - \alpha_i)\cdot x^2 + \omega_i \cdot q} \cdot u^{\upsilon_i}} \right)^{k^i} \in \mathbb{G} \tag{120}$$

The equality above must be true for a computationally bounded prover, or we find a non-trivial relationship between generators g, h, u.

B. Proof for Theorem Two

Proof. Perfect completeness follows from the fact that the protocol Baseline is trivially complete.

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

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

Simulator $\mathcal{S}$ randomly generates committed transcripts $\vec{C}$, M and sends them to the verifier. After receiving challenge x from the verifier, the prover rewinds and regenerates M with a randomly generated $\vec{y}'$ s.t. the updated M^* is:

$$M^* = \frac{\prod_i^{m_b} u^{y_i'}}{\prod_{j=1}^{b} C_j^{x^j}} \in \mathbb{G} \tag{121}$$

The simulator then randomly generates $\vec{a}$ before sending them to the verifier. Since all elements in proof transcripts are either independently randomly distributed or their relationship is fully defined by the verification equations, we can conclude that the protocol Baseline is PHVZK.

To prove CWEE, we define extractor $\mathcal{X}$ that calls on extractors $\mathcal{X}_{input}$ defined earlier to extract witnesses for the two sub-protocols used in the protocol Baseline.

We already know $\mathcal{X}$ can extract $\vec{a}, \vec{\alpha}, \vec{v}$ and $\vec{r}, \vec{\epsilon}$ using extractor $\mathcal{X}_{input}$ from committed transcripts $\vec{S}, \vec{T}$. Using input witnesses, we can use the function computeSubCircuitKeys to compute circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$.

Next, we verify the validity of circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$ by checking if we can extract the same circuit witnesses from commitments $\vec{C}, M$. The extractor $\mathcal{X}$ interacts with the prover and receives $\vec{C}, M$ from the prover. The extractor $\mathcal{X}$ then generates $b+1$ challenges $\vec{x}$ and forwards them to the prover. After receiving circuit inputs $\vec{a}'$ and circuit evaluation transcripts $\vec{y}_1'$ computed from the first challenge x, the extractor rewinds and repeats this step b times to generate challenges $x_2, ..., x_{b+1}$ and retrieve witnesses $\vec{y}_2', ..., \vec{y}_{b+1}'$ from interpolation. The retrieved witnesses must pass equality 17 for a computationally bounded prover, or else we find a non-trivial discrete log relationship between generators $\vec{u}$.

We then apply polynomial interpolation to retrieve circuit witnesses $\vec{\tau}_1, ..., \vec{\tau}_{m_b}$ and $\vec{\mu}$ from $\vec{y}_1', ..., \vec{y}_{b+1}'$, The extractor then follows the protocol to compute $\vec{r}'$, and then trivially extracts $\vec{r}, \vec{\epsilon}$ with any pair of challenge x_1, x_2.

Finally, we check if the extracted circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$ match that computed from input witnesses $\vec{a}, \vec{\alpha}$ using computeSubCircuitKeys functions, which must be true for a computationally bounded prover except with negligible probability s.t. the dishonest prover made a correct guess on challenge x or else we find a non-trivial discrete log relationship between generators g, h (for input witnesses) and/or between generators $\vec{u}$ (for circuit witnesses).

C. Proof for Theorem Three

Proof. Perfect completeness follows from the fact that the protocol BooleanityTest is trivially complete.

To prove PHVZK for relation 58, we define a simulator $\mathcal{S}_{b-test}$. To start, $\mathcal{S}_{b-test}$ randomly generates a group element D, which represents the committed polynomial. After challenge x is received from the verifier,

$\mathcal{S}_{b-test}$ uses a simulator $\mathcal{S}_p$ to simulate proof transcripts needed for polynomial commitment evaluation, which we know exist for a fact [9].

The simulators $\mathcal{S}_{b-test}$ and $\mathcal{S}_p$ choose all proof elements and challenges according to the randomness supplied by the adversary from their respective domains or compute them directly as described in the protocol. 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 BooleanityTest is PHVZK.

To prove this protocol has CWEE, we first define an extractor $\mathcal{X}_{b-test}$ for Protocol Booleanity that extracts witnesses $\vec{b}, \vec{\beta}$.

The extractor first receives a vector commitment D from the prover. The extractor then generates $2l+1$ challenges $k_1, ..., k_{2l+1}$ and retrieves $2l+1$ $\vec{y_1}$ and $2l+1$ $\vec{y_2}$ through repeated rewinding. The extractor $\mathcal{X}_{b-test}$ then calls on an extractor for the polynomial commitment evaluation protocol (which we know exists for a fact [9]) to extract witnesses $\vec{\delta_1}, \vec{\delta_2}, \phi$ or else we find a non-trivial relationship between elements in $\vec{u}, h$.

It is trivial to extract witnesses b_i, β_i from $\delta_1.\delta_2$ for $i = \{1, ..., l\}$ s.t. they must pass the equality test defined in equation 61 except with negligible probability of a dishonest prover making the right guess on x

D. Proof for Theorem Four

Proof. Perfect completeness follows from the fact that the protocol Binary Boost is trivially complete.

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

We have already shown $\mathcal{S}_{input}$ can simulate all interactions needed in sub-protocol Input-Mapping, which includes generating transcripts $\vec{S}, \vec{T}$. We have also shown that $\mathcal{S}_{b-test}$ can simulate all interactions needed in the sub-protocol Booleanity Test, which covers generating transcripts D. We now show $\mathcal{S}$ generates the rest of the transcripts according to the randomness supplied by the adversary from their respective domains or computes them directly as described in the protocol.

Simulator $\mathcal{S}$ randomly generates committed transcripts $\vec{C}, D, M$ and sends them to the verifier. After receiving challenge x from the verifier, the prover rewinds and regenerates M^* with a randomly generated $\vec{z_{i'}}$ s.t. M^* is:

$$M^* = \frac{\prod_{i'}^{m_b/tb} u^{z_{i'}}}{\prod_{j=1}^{b} C_j^{x^j}} \in \mathbb{G} \tag{122}$$

The simulator then reversely computes $\vec{y}, \vec{v}$ from generated $\vec{z_{i'}}$ and randomly generates $\vec{a}'$ and $\vec{b}'$ according to protocol specification before sending them to the verifier. Since all elements in proof transcripts are either independently randomly distributed or their relationship is fully defined by the verification equations, we can conclude that the protocol Binary Boost is PHVZK.

To prove CWEE, we define an extractor $\mathcal{X}$ that calls on extractors $\mathcal{X}_{input}$ and $\mathcal{X}_{b-test}$ defined earlier to extract witnesses for the two sub-protocols used in the BinaryBoost protocol.

We already know we can extract input witnesses $\vec{a}, \vec{\alpha}, v$ and $\vec{r}, \vec{\epsilon}$ using extractor $\mathcal{X}_{input}$ from $\vec{a}', \vec{r}'$ and committed transcripts $\vec{S}, \vec{T}$. We also know that we can extract boolean witnesses $\vec{b}, \vec{\beta}$ from $\vec{b}'$ and committed value D using extractor $\mathcal{X}_{b-test}$. Since we can extract witnesses for $\vec{a}'$ and that of their boolean constituents $\vec{a}'$, equality (line TBD-115 of the protocol) 62 must be true for a computationally bounded prover except for the negligible chance of making the right guess on x.

Using these boolean witnesses, we can use the function computeSubCircuitKeys to compute circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$. We validate the CWEE of the protocol by testing if the witnesses computed from input witnesses match those committed by the prover using commitments $\vec{C}, M$.

The extractor $\mathcal{X}$ interacts with the prover and receives $\vec{C}, M, D$ from the prover, then generates $b+1$ challenges $\vec{x}$ and forwards them to the prover. After receiving circuit inputs $\vec{a}', \vec{b}'$ and circuit evaluation transcripts $\vec{y}_1, \vec{v}_1$ computed from the first challenge x, the extractor rewinds and repeats this step b times to retrieve $\vec{y}_2, ..., \vec{y}_{b+1}$, $\vec{v}_2, ..., \vec{v}_{b+1}$. We then apply polynomial interpolation to retrieve circuit witnesses $\vec{\tau}$ and $\vec{\mu}$ from $\vec{y}_1, ..., \vec{y}_{b+1}$, and retrieve batched circuit witnesses $\vec{\gamma}$ and $\vec{\mu}'$ from $\vec{y}_1, ..., \vec{y}_{b+1}$ and $\vec{v}_1, ..., \vec{v}_{b+1}$. Where each element $\gamma_{i',j}$ in $\vec{\gamma}$ is the sum of the products of tb elements in $\vec{\tau}_j$ as defined in equation 63.

Apply any challenge x to each $\gamma_{i',j}$ and masking key μ_i we get the exponents of the left hand side generators of equality 68, which are exponents of $\vec{C}(\vec{\tau})$ multiplied by powers of x plus exponents of $M(\vec{\mu})$. Apply any challenge x to $\vec{y}$ and multiply each element in $\vec{v}$ by q, we get the exponents of the right-hand side of equality 68 $\vec{u}(z_{i'})$ (defined in equation 70):

$$\sum_{j=1}^{b} \gamma_{i',j} x^j + \mu = (\sum_{i=1}^{m_b/tb} (y_i \cdot x^{-1}) \bmod q \cdot 2^i) + v_{i'} \cdot q$$

The equality above must be true for each $i' = \{1, ..., m_b/tb\}$ unless we find a non-trivial discrete log relation between generators $\vec{u}$. After checking that the that the exponents of both sides do not overflow p and $tb < |q|$, we multiply both sides by x and then apply mod q to both sides we get:

$$\sum_{j=1}^{b} \gamma_{i',j} x^{j+1} + \mu_{i'} x = \sum_{i=1}^{tb} y_i \cdot 2^i \in \mathbb{Z}_q \tag{123}$$

We know that $y_i = \sum_{j=1}^{b} \tau_{i,j} x^{i+1} + \mu_i x \in \mathbb{Z}_q$ by protocol definition, so we can replace y_i by using in $\vec{\tau}, \vec{mu}$ received from the prover.

$$\sum_{j=1}^{b} \gamma_{i',j} x^{j+1} + \mu_{i'} x = \sum_{i=1}^{tb} (\sum_{j=1}^{b} \tau_{i,j} x^{i+1} + \mu_i x) \cdot 2^i \in \mathbb{Z}_q \tag{124}$$

Assuming the dishonest prover can only alter y_i^* by $y_i \pm 1$, the right hand side of the equality above can only be altered to $\sum_{i=1}^{tb} (\sum_{j=1}^{b} \tau_{i,j} x^{i+1} + \mu_i x) \cdot 2^i \pm 2^i$. Since $\pm 2^i$ is in the constant term and no combination of $\sum_{i=1}^{tb} \pm 2^i$ can possibly compute to 0, we say that $\vec{\tau}$ is legit except for a negligible probability of making a correct guess on challenge x.

This (± 1) property can be validated by checking if we can successfully extract witnesses $\vec{r}, \vec{\epsilon}$ from transcripts $\vec{r}'$ (computed by the verifier following the protocol) and D using extractor $\mathcal{X}_{b-test}$ s.t. each $r_i \in \{0, 1\}$. If we can confirm each $r_i \in \{0, 1\}$, then we know y_i^* can only be altered by ± 1 since $o_i = r_i + y_i$.

Finally, we check if the extracted circuit witnesses $\vec{r}, \vec{\epsilon}, \vec{\tau}$ match that computed from input witnesses $\vec{b}, \vec{\beta}$ using computeSubCircuitKeys functions, which must be true for a computationally bounded prover except with negligible probability s.t. the dishonest prover made a correct guess on challenge x or else we find a non-trivial d-log relationship between generators g, h (for input witnesses) and/or between generators $\vec{u}$ (for circuit witnesses).

References

1. Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: Lightweight sublinear arguments without a trusted setup. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 2087–2104. ACM Press, Dallas, TX, USA (Oct 31 – Nov 2, 2017). https://doi.org/10.1145/3133956.3134104
2. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and hardness of approximation problems. In: 33rd FOCS. pp. 14–23. IEEE Computer Society Press, Pittsburgh, PA, USA (Oct 24–27, 1992). https://doi.org/10.1109/SFCS.1992.267823
3. Arora, S., Safra, S.: Probabilistic checking of proofs; A new characterization of NP. In: 33rd FOCS. pp. 2–13. IEEE Computer Society Press, Pittsburgh, PA, USA (Oct 24–27, 1992). https://doi.org/10.1109/SFCS.1992.267824
4. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: 23rd ACM STOC. pp. 21–31. ACM Press, New Orleans, LA, USA (May 6–8, 1991). https://doi.org/10.1145/103418.103428

5. Babai, L., Fortnow, L., Lund, C.: Non-deterministic exponential time has two-prover interactive protocols. In: 31st FOCS. pp. 16–25. IEEE Computer Society Press, St. Louis, MO, USA (Oct 22–24, 1990). https://doi.org/10.1109/FSCS.1990.89520
6. Baum, C., Malozemoff, A.J., Rosen, M.B., Scholl, P.: Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 92–122. Springer, Heidelberg, Germany, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84259-8$_4$
7. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 701–732. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_2$3
8. Bhadauria, R., Fang, Z., Hazay, C., Venkitasubramaniam, M., Xie, T., Zhang, Y.: Ligero++: A new optimized sublinear IOP. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2025–2038. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3417893
9. Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 327–357. Springer, Heidelberg, Germany, Vienna, Austria (May 8–12, 2016). https://doi.org/10.1007/978-3-662-49896-5$_1$2
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243868
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354255
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2019). https://doi.org/10.1007/978-3-030-26954-8$_1$6
13. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 677–706. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-1$_2$4
14. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part II. LNCS, vol. 14005, pp. 499–530. Springer, Heidelberg, Germany, Lyon, France (Apr 23–27, 2023). https://doi.org/10.1007/978-3-031-30617-4$_1$7
15. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.P.: Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 738–768. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-1$_2$6
16. Cramer, R., Damgård, I.: Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In: Krawczyk, H. (ed.) CRYPTO'98. LNCS, vol. 1462, pp. 424–441. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 23–27, 1998). https://doi.org/10.1007/BFb0055745
17. Dittmer, S., Ishai, Y., Lu, S., Ostrovsky, R.: Improving line-point zero knowledge: Two multiplications for the price of one. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 829–841. ACM Press, Los Angeles, CA, USA (Nov 7–11, 2022). https://doi.org/10.1145/3548606.3559385
18. Dittmer, S., Ishai, Y., Ostrovsky, R.: Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446 (2020), `https://eprint.iacr.org/2020/1446`
19. Frederiksen, T.K., Nielsen, J.B., Orlandi, C.: Privacy-free garbled circuits with applications to efficient zero-knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 191–219. Springer, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46803-6$_7$
20. 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`

21. 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)
22. 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
23. 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
24. 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
25. 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
26. 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
27. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 239–252. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 1990). https://doi.org/10.1007/0-387-34805-$0_2$2
28. 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
29. 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
30. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020), https://eprint.iacr.org/2020/1275
31. 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
32. 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
33. Weng, C., Yang, K., Yang, Z., Xie, X., Wang, X.: AntMan: Interactive zero-knowledge proofs with sublinear communication. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 2901–2914. ACM Press, Los Angeles, CA, USA (Nov 7–11, 2022). https://doi.org/10.1145/3548606.3560667
34. 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
35. 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
36. 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