SmartBean: Transparent, Concretely Efficient, Polynomial Commitment Scheme with Logarithmic Verification and Communication Costs that Runs on Any Group

Frank Y.C. Lu

YinYao

Abstract. We introduce a new, concretely efficient and transparent polynomial commitment scheme with logarithmic verification time and communication cost that can run on any group. Existing group-based polynomial commitment schemes must use less efficient groups, such as class groups of unknown order [9] or pairing-based groups [16] to achieve transparency (no trusted setup), making them expensive to adopt in practice.

We offer the first group based polynomial commitment scheme that can run on any group s.t. it does not rely on expensive pairing operations or require class groups of unknown order to achieve transparency while still providing logarithmic verifier time and communication cost.

The prover work of our work is dominated by $4n\,\mathbb{G}$ multi-exponentiations, the verifier work is dominated by $4\log n\,\mathbb{G}$ exponentiations, and the communication cost is $4\log n\,\mathbb{G}$. Since our protocol can run on fast groups such as Curve25519, we can easily accelerate the multi-exponentiations with Pippenger's algorithm. The concrete performance of our work shows a significant improvement over the current state of the art in almost every aspect.

1 Introduction

Zero-knowledge succinct arguments of knowledge (zkSNARK) is a subject of great research interest in the field of cryptography, and the polynomial commitment scheme is the most important building block of some of the most popular zero-knowledge systems deployed today. The construction of some of the most popular zkSNARKs such as Groth16 [14], Sonic [17], PLONK [13], Marlin[10], and more recently Spartan [18] and Gemini [6] can be generally described in two steps: first, reduce a satisfying assignment to evaluation of a polynomial commitment. Second, apply some polynomial commitment scheme to validate the soundness of the commitment created in step one.

Non-transparent polynomial commitment schemes such as KZG [15] offers great verifier performance and low communication cost, making it the popular choice for many zkSNARK protocols [13] [14]. Since the only reason popular SNARKs such as PLONK require trusted setup is because KZG requires it, a lack of efficient transparent polynomial commitment solutions is one of the biggest well-known setbacks of SNARKs protocols.

One class of transparent schemes is based on the use of Interactive Oracle Proofs on Reed-Solomon codes (RS-IOP), such schemes are adopted by Ligero [1], Aurora [2], Virgo [20], Fractal [11], and Orion [19]. RS-IOP schemes are plausible post-quantum and generally offer a decent prover runtime cost, even though their asymptotic cost is less optimal. However, its communication cost is expensive, and the soundness error is high. The performance advantage diminishes when running the required number of repetitions to reach provable 120bit+ security [11].

Another popular class of transparent polynomial commitment schemes with logarithmic communication complexity is the discrete log-based work (LCC-DLOG) developed by Bootle et al. [5], which is the basis of many later works, including Bulletproofs [8], and is further optimized by the likes of Spartan [18]. More recently, Halo [7] introduced the idea of amortization of verifier computation through recursive composition of proof, which was later generalized by Bunz et al. [3]. However, these group-based schemes generally require linear verifier work, leaving plenty of room for improvement.

More recently, the development of DARK [9] and Dory [16] finally achieved logarithmic verifier work without trusted setup for group-based schemes. DARK is a new class of polynomial commitment schemes based on groups of unknown order (DARK-GUO) and recently expanded by [4]. Unfortunately, group operations on these types of groups are significantly slower than those of curve-based implementation [12] [16]. On the other hand, Dory improved verifier work to log $n\ \mathbb{G}_T$. However, the exponentiation operations on $\mathbb{G}_T$ are more than 10X costlier than those on $\mathbb{G}$, and the communication cost of $\mathbb{G}_T$ is 6X that of $\mathbb{G}$, making them still less optimal in practice.

1.1 Summary of Contributions

We introduce a new transparent polynomial commitment scheme that offers linear prover work, logarithmic verifier work, and logarithmic communication cost. Our protocol is the first group-based polynomial commitment scheme with sub-linear verifier work and communication cost that does not rely on expensive pairing or groups of unknown order, translating to significant performance gains in practice.

We first introduce the base version of our protocol in Section 3 after a brief review of Bulletproofs' inner product argument. The base version gives 4log n verifier work for group exponentiation operations. This is achieved using a pair of transcripts in each round of the recursion: B_L and $B_{\delta L}$, where B_L is used by the prover to provide a pre-computed value and $B_{\delta L}$ is used to check if the B_L sent by the prover is valid. In Section 3, we show the algebraic equation to show each pair of $B_L, B_{\delta L}$ can't both be right at the same time.

However, the base version of our protocol would still require linear $O(n)$ field operations, which is still efficient for small to medium sized polynomials but not so much for large polynomials. In Section 4, we introduce a math trick to bring down the total verifier work to $O(\log n)$. In summary, the prover work of our protocol is dominated by $4n\,\mathbb{G}$ multi-exponentiations, the verifier work is dominated by $4\log n\,\mathbb{G}$ group exponentiations, and the communication cost is dominated by $4\log n\,\mathbb{G}$.

2 Preliminaries

2.1 Assumptions

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.

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

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

2.2 Interactive Arguments

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 $\mathcal{A}$:

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

Definition 3. (Public Coin) All messages sent from $\mathcal{V}$ to $\mathcal{P}$ are chosen uniformly at random and independently of $\mathcal{P}$'s messages.

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

Definition 4. (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:

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

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

Note that there is also a probability aspect to our definition of CWEE (which can also be called statistical-witness extended emulation in some places [5]). Informally, if an adversary can produce an argument that satisfies the verifier with some probability, then there exists an emulator producing an identically distributed argument with the same probability, as well as a witness. The zero-knowledge property requires that the verifier doesn't learn anything about the witness from its interaction with an honest prover.

Generalized Special Soundness We follow the presentation first introduced in Bootle et al. [5] and later enhanced by Bulletproofs [8] and also found in DARK [9] and Dory [16]. Consider a public-coin interactive argument with f rounds and $(n_1, ..., n_f)$ tree of accepting transcripts with challenges sampled from a large message space. The tree has depth f with its root labelled with the statement x. Each node at depth $i < f$ has n_i children, and each children is labelled with a distinct value for the ith challenge. Every path from the root to a leaf corresponds to an accepting transcript, and there are a total of $\prod_{i=1}^{f} n_i$ distinct accepting transcripts.

Lemma 1. (Generalized Forking Lemma) [8] [9] [5] Let $(\mathcal{P}, \mathcal{V})$ be an f-round public-coin interactive argument system for a relation $\mathcal{R}$. Let $\mathcal{T}$ be a tree-finder polynomial time algorithm that has access to a $\boldsymbol{Recorder}()$ with rewinding capabilities outputs an $(n_1, ..., n_f)$-tree of accepting transcripts. Let $\mathcal{X}$ be a deterministic polynomial time extractor that use $\mathcal{T}$'s outputs compute a witness w for the statement x with overwhelming probability. Then $(\mathcal{P}, \mathcal{V})$ has witness-extended emulation.

Definition 5. (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

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

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

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

2.3 Commitment Schemes

Our definitions are based on the polynomial commitments from [9].

Definition 6. (Commitment scheme) A commitment scheme $\mathcal{C}$ is a tuple $\mathcal{C} = (\boldsymbol{Setup}, \boldsymbol{Commit}, \boldsymbol{Open})$ of PPT algorithms where:

- **Setup**$(1^\lambda) \rightarrow pp$ generates public parameters pp.
- **Commit**$(pp; x) \rightarrow (C; \phi)$ takes a secret message x and produces a public commitment C with a secret opening hint ϕ.
- **Open**$(pp, C, x, \phi) \rightarrow b \in \{0, 1\}$ verifies the opening of commitment C to the message x provided with the opening hint ϕ.

$$Pr\left[\begin{array}{ccc} & & pp \leftarrow Setup(1^\lambda) \\ b_0 = b_1 \neq 0 \wedge & : & (P, x_0, x_1, \phi_0, \phi_1) \leftarrow \mathcal{A}(pp) \\ x_0 \neq x_1 & & b_0 \leftarrow \boldsymbol{Open}(pp, P, x_0, \phi_0) \\ & & b_1 \leftarrow \boldsymbol{Open}(pp, P, x_1, \phi_1) \end{array}\right] \leq \boldsymbol{negl}(\lambda)$$

Definition 7. (Polynomial Commitment) A polynomial commitment scheme is a tuple of protocols $\mathcal{C} = (\boldsymbol{Setup}, \boldsymbol{Commit}, \boldsymbol{Open}, \boldsymbol{Eval})$ where $(\boldsymbol{Setup}, \boldsymbol{Commit}, \boldsymbol{Open})$ is a binding commitment scheme for a message space $\mathcal{R}[X]$ of polynomials over some ring $\mathcal{R}$, and function $\boldsymbol{Eval}$ is defined as:

- $\boldsymbol{Eval}(pp, C, z, y, n; f(X), \phi) \rightarrow b \in \{0,1\}$ is an interactive public-coin protocol between a PPT prover $\mathcal{P}$ and verifier $\mathcal{V}$. Both $\mathcal{P}$ and $\mathcal{V}$ have a input commitment C, points $z, y \in \mathbb{Z}_p$, and a degree $n \geq deg(f(X))$. The prover additionally knows the opening of C to a secret polynomial $f(X) \in R[X]$. The protocol convinces the verifier that $f(z) = y$.

$$\mathcal{R}_{Eval(pp)} = \left\{ \begin{array}{r} \langle (C, z, y, n), (f(X), \phi) \rangle : f \in R[X] \wedge \ deg(f(X)) \leq n \ \wedge \ f(z) = y \\ \wedge \ \boldsymbol{Open}(pp, C, f(X), \phi) = 1 \end{array} \right\}$$

Knowledge Soundness : In the $\boldsymbol{Eval}$ protocol, provers must know a polynomial $f(X)$ such that $f(z) = y$ and C is a commitment to $f(X)$. Since $\boldsymbol{Eval}$ is a public-coin interactive argument, we say this knowledge property as a special case of witness-extended emulation for $\boldsymbol{Eval}$, and that a commitment scheme $\mathcal{C}$ has witness-extended emulation if $\boldsymbol{Eval}$ has CWEE for $\mathcal{R}_{Eval(pp)}$.

Zero Knowledge : tuple $(\boldsymbol{Gen}, \boldsymbol{Commit}, \boldsymbol{Open}, \boldsymbol{Eval})$ is a perfect special honest-verifier zero-knowledge, extractable polynomial commitment scheme for polynomials $f(X) \in R[X]$. If $(\boldsymbol{Gen}, \boldsymbol{Commit}, \boldsymbol{Open})$ is a commitment scheme for $f(X) \in R[X]$ then $\boldsymbol{Eval}$ is an PSHVZK interactive argument of knowledge for $\mathcal{R}_{Eval(pp)}$.

2.4 Notations

Let $\mathbb{G}$ denote any type of secure cyclic group of prime order p, and let $\mathbb{Z}_p$ denote an integer field modulo p. A commitment is a group element denoted by capital letters. e.g. $C = g_1^{a_1} g_2^{a_2} ... g_n^{a_n} \ h^{\phi} \in \mathbb{G}$ is a commitment committed to a vector $\vec{a}$. A group element $B \in \mathbb{G}$ is also a group element denoted by a capital letter. For generators used to compute other group elements in our protocol, such as $\vec{g}, h, u \in \mathbb{G}$, we use lower case letters to denote them. We also use bold letters to denote generators created during the initialization phase, e.g. generator set $\vec{\boldsymbol{g}}$ is the SRS of our protocol generated during the initialization phase. Greek letters are used to label hidden key values. e.g. ϕ is the blinding key for commitment C on generator $h \in \mathbb{G}$.

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

Let $\vec{a} \,||\, b$ denote the concatenation of the second element to the first vector, which returns a new vector with length $|\vec{a}|+1$. e.g. $\vec{a} \,||\, b \rightarrow (a_1, ..., a_n, b)$. For $1 \leq l \leq n$, we use the following format to represent a vector divided into two slices:

$$\vec{a}_{[:l]} = \{a_1, ..., a_l\} \in \mathbb{F}^l, \quad \vec{a}_{[l:]} = \{a_l, ..., a_n\} \in \mathbb{F}^l$$

Finally, $\vec{0}^n$ indicates a vector with n zeros. e.g. $\vec{0}^n = \{0_1, ..., 0_n\}$.

3 Transparent Polynomial Commitment Protocol That Supports Any Group

In this section, we begin with a quick review of Bulletproofs' inner product argument, then introduce a new protocol that offers logarithmic verifier work for group exponentiation operations. In Section 4, we will introduce a computation trick that allows our protocol to achieve total logarithmic verification work and communication cost. The interactive protocol we will introduce in this section can be made non-interactive with the Fiat-Shamir transform.

3.1 Bulletproofs' Inner Product Argument Revisited

Let H denote a function that takes four vector inputs and outputs a single group element in $\mathbb{G}$. e.g.:

$$H(\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4, y) = \vec{g}_{[:n']}^{\vec{v}_1} \vec{g}_{[n':]}^{\vec{v}_2} \vec{h}_{[:n']}^{\vec{v}_3} \vec{h}_{[n':]}^{\vec{v}_4} \cdot u^y \in \mathbb{G}$$

Before the protocol runs, the prover initializes $P \in \mathbb{G}$ using two committed vectors and their inner product y s.t.:

$$P = H(\vec{a}_{[:n']}, \vec{a}_{[n':]}, \vec{b}_{[:n']}, \vec{b}_{[n':]}, y) = \vec{g}_{[:n']}^{\vec{a}_{[:n']}} \vec{g}_{[n':]}^{\vec{a}_{[n':]}} \vec{h}_{[:n']}^{\vec{b}_{[:n']}} \vec{h}_{[n':]}^{\vec{b}_{[n':]}} \cdot u^y \in \mathbb{G}$$

In each round of the recursion, the prover computes and sends L, R to the verifier so that the verifier can logarithmically reduce the size of vector P:

$$L = H(\vec{0}, \quad \vec{a}_{[:n']}, \quad \vec{b}_{[n':]}, \quad \vec{0}, \quad \langle \vec{a}_{[:n']}, \vec{b}_{[n':]} \rangle) = \vec{g}_{[n':]}^{\vec{a}_{[:n']}} \vec{h}_{[:n']}^{\vec{b}_{[n':]}} \cdot u^{\langle \vec{a}_{[:n']}, \vec{b}_{[n':]} \rangle} \in \mathbb{G}$$

$$R = H(\vec{a}_{[n':]}, \quad \vec{0}, \quad \vec{0}, \quad \vec{b}_{[:n']}, \quad \langle \vec{a}_{[n':]}, \vec{b}_{[:n']} \rangle) = \vec{g}_{[:n']}^{\vec{a}_{[n':]}} \vec{h}_{[n':]}^{\vec{b}_{[:n']}} \cdot u^{\langle \vec{a}_{[n':]}, \vec{b}_{[:n']} \rangle} \in \mathbb{G}$$

The verifier computes P' (P for the next round) from L, R provided by the prover and the challenge x chosen by the verifier s.t.

$$P' = L^{x^2} \cdot PR^{x^{-2}} = \vec{g}'^{\vec{a}'} \vec{h}'^{\vec{b}'} \cdot u^{y'} \in \mathbb{G}$$

Both the prover and the verifier compute $\vec{g}'$ and $\vec{h}'$ s.t. $\vec{g}' = \vec{g}_{[:n']}^{x^{-1}} \circ \vec{g}_{[n':]}^{x}$ and $\vec{h}' = \vec{h}_{[:n']}^{x} \circ \vec{h}_{[n':]}^{x^{-1}}$. Only the prover can compute $\vec{a}'$ and $\vec{b}'$ s.t. $\vec{a}' = \vec{a}_{[:n']} \cdot x + \vec{a}_{[n':]} \cdot x^{-1}$ and $\vec{b}' = \vec{b}_{[:n']} \cdot x^{-1} + \vec{b}_{[n':]} \cdot x$. It is worth noticing that the challenge x introduced in each round makes the left and right slices, or g, h different so that a malicious prover cannot cheat by swapping generators and coefficients.

We know for a fact that $y' = \langle \vec{a}', \vec{b}' \rangle = y + \langle \vec{a}_{[:n']}, \vec{b}_{[n':]} \rangle \cdot x^2 + \langle \vec{a}_{[n':]}, \vec{b}_{[:n']} \rangle \cdot x^{-2}$. If the prover can provide correct openings $\vec{a}', \vec{b}'$ on generators $\vec{g}', \vec{h}'$, then the verifier can validate the inner product argument by checking whether P' computed from L, R, P can be opened with $\vec{a}', \vec{b}'$. In Bulletproofs, the prover provides the openings when the size of group vectors ($|\vec{g}'|$ and $|\vec{h}'|$) reaches 1.

One of the key contributions of bulletproofs' recursion mechanism is that it offers a way to achieve logarithmic communication cost. However, the verification cost is still linear due to the expensive linear computations required to compute generators $\vec{g}, \vec{h}$ in each round.

3.2 A New Transparent Polynomial Commitment Scheme with Fixed Generators

In our protocol, coefficients $a_1, ..., a_n$ of a polynomial $f(X)$ are committed into a group element $C = g_1^{a_1}, ..., g_n^{a_n} h^{\phi}$. Upon providing $z, y \in \mathbb{Z}_p$ the prover creates a proof for $f(z) = y$. Note that $g_1, ..., g_m \in \mathbb{G}^m$ s.t. $m \geq n$ are public parameters generated when the system is initialized.

We now update the relation $\mathcal{R}_{Eval}$ defined in definition 7 with the inputs we will be using to construct our polynomial commitment scheme in the following format: $\mathcal{R} = \{\langle (Public\ Inputs)\ \ , (Witnesses) \rangle : Relation\}$.

$$\{\langle (\vec{g}, h, u, C \in \mathbb{G}, z, y \in \mathbb{Z}_p), (\vec{a} \in \mathbb{Z}_p^n, \phi \in \mathbb{Z}) \rangle : C = \vec{g}^{\vec{a}} h^{\phi} \wedge f(z) = y\} \quad (1)$$

The most costly operation for verifiers in Bulletproofs' inner product argument is the task of computing base generators $\vec{g}', \vec{h}'$ in each round of a recursion. Although subsequent protocols such as Halo [7] do not need the verifier to compute the second vector $\vec{h}'$, computing $\vec{g}'$ in each round is still expensive and requires both the prover and the verifier to perform $2n$ group exponentiations.

3.3 The Basic Idea

We first simplify how we compute the committed coefficients a' and z'. Instead of multiplying both left and right sets by the challenge x_j and its inverse (subscript j means jth round in the recursion), we only apply the challenge (or its inverse) to the right slice of each vector.

$$\vec{a}\,' = \vec{a}_{[:n']} + \vec{a}_{[n':]} \cdot x_j \in \mathbb{Z}_p^{n'} \tag{2}$$

$$\vec{z}\,' = \vec{z}_{[:n']} + \vec{z}_{[n':]} \cdot x_j^{-1} \in \mathbb{Z}_p^{n'} \tag{3}$$

This simplification is safe since $\vec{a}\,' = \vec{a}_{[:n']}x_j + \vec{a}_{[n':]}x_j^{-1}$ in bulletproofs can be trivially factored to $\vec{a}\,' \cdot x_j^{-1} = \vec{a}_{[:n']} + \vec{a}_{[n':]}x_j^{-2}$ by the verifier (the challenge is only applied to the right slice of $\vec{a}\,'$), and the same applies to vector $\vec{z}\,'$.

This simplification also provides a marginal reduction of one group exponential operation when computing $g'_i = g_i \cdot g_{n'+i}^{x^{-1}}$ for $i = \{0, ..., n-1\}$ in our protocol instead of $g'_i = g_i^{x^{-1}} \cdot g_{n'+i}^{x}$ in that of bulletproofs and its derivatives.

Still, even computing the simplified $\vec{g}'$ in each round is still expensive. Instead, we want the prover to do the bulk of the work for the verifier in a trustworthy way. We do this by introducing a pair of transcripts B_L and $B_{\delta L}$ in each round of the recursion.

B_L is the transcript that allows the prover to perform the multi-exponentiation operations: B_L is the product of the first half of generators $B_L = \prod_0^{n'-1} g_i$. The verifier can use B_L to get the product of the second half of generators $B_R = \prod_{n'}^{n-1} g_i$ and then apply challenge x to B_R to get B' s.t. $\prod_0^{n'-1} g'_i = B_L \cdot B_R^x$. If this works, it would be a lot more efficient than computing the whole $\vec{g}'$ in each round, because eventually we will shrink the vector to one single generator $B' = B_L \cdot B_R^x$ in the final round.

Unfortunately, it is easy for a dishonest prover to cheat if we just ask the dishonest prover to provide either B_L or B_R because the generators in $\vec{g}$ are not binding. For example, assuming $\vec{g}$ only has two generators g_l, g_r, we can find a pair of invalid B_L, B_R that are generators of $a_1^* \neq a_1$ and $a_2^* \neq a_2$ s.t.

$$B_L{}^{a_1^*}(B/B_L)^{a_2^*} = P = g_l^{a_1} g_r^{a_2}$$

Note that $B_R = B/B_L$. Since $B = g_l g_r$, the honest values are $B_L = g_l, B_R = g_r$. The dishonest prover can use the algebraic formula below to find an invalid B_L s.t. $B_L \neq g_l$:

$$P = B_L{}^{a_1^*} \cdot (g_l g_r / B_L)^{a_2^*}$$

Rewriting the equation above, we get the algebraic formula for finding the cheating $B_L \neq g_l$:

$$B_L = (P/(g_l g_r)^{a_2^*})^{a_2^* - a_1^*} \tag{4}$$

After the challenge x is available, both the prover and the verifier compute P' and B' s.t. $P' = B'^{a'^*}$ where $a'^* \neq a'$ and that:

$$B' = B_L \cdot (B/B_L)^{x^{-1}} \tag{5}$$

This is not good, as there is a readily available formula for adversaries to cheat, because we cannot check whether the B_L value sent by the prover is legit or not.

$B_{\delta L}$ is the transcript used to validate the legitimacy of B_L: We can do that by making the prover provide another transcript $B_{\delta L}$ on another product of generators B_δ s.t.

$$B_\delta = g_l g_r^2 \tag{6}$$

The exponent of g_r in the equation above doesn't need to be 2, we pick 2 because it is the smallest value that does the work. The algebraic formula to find $B_{\delta L}$ is $P = {B_{\delta L}}^{a_1^*} \cdot (g_l g_r^2 / B_{\delta L})^{a_2^*}$, which reduces to:

$$B_{\delta L} = (P/(g_l g_r^2)^{a_2^*})^{a_2^* - a_1^*} \tag{7}$$

and the formula to compute B' from B_δ and $B_{\delta L}$ is similar to that used to compute B' from B and B_L, except we need to factor out the exponent 2 on g_r :

$$B' = B_{\delta L} \cdot (B_\delta / B_{\delta L})^{x^{-1}/2} \tag{8}$$

For B' to be the product of generators of P' (e.g. $B' = g_l g_r$), $B_{\delta L}$ must be the same as B_L except for a negligible probability (correctly guess challenge x) so that both equations 5 and 8 compute the same generator B':

If we replace B_L with the right-hand side of the equation 4 and $B_{\delta L}$ with the right-hand side of the equation 7 we get the following equality:

$$(P/(g_l g_r)^{a_2^*})^{a_2^* - a_1^*} = (P/(g_l g_r^2)^{a_2^*})^{a_2^* - a_1^*}$$

After canceling out all common terms, we get the following equality, which is not possible unless g_r is 1:

$$g_r = g_r^2$$

The above equality holds even for $g_l^* \neq g_l$ and $g_r^* \neq g_r$. We can use the concept above to build a polynomial evaluation protocol that can evaluate committed polynomials of any size.

3.4 Building the Protocol With Logarithmic Verification Cost on Group Exponential Operations

Besides making provers do the work of computing generators for each round (which it has to do anyway), we make two more simplifications to bulletproofs' inner product protocol.

The first area is that we only recursively evaluate one set of generators $\vec{g}$ because only the vector commitment that commits to the coefficients of the polynomial that we are evaluating needs to be concealed. The powers of evaluation point z (e.g. $1, z, z^1, z^2, ..., z^n$) is public information.

Like that of Bulletproofs, our protocol defines and works on a new group element $P \in \mathbb{G}$ created from the vector commitment C of the polynomial $f(X)$ and the result y of the evaluation point z s.t. $f(z) = y$.

$$P = C \cdot u^y \qquad // \text{ equivalent to } \vec{g}^{\vec{a}} h^\phi u^y \tag{9}$$

To make the paper easier to follow, we also define a function H like that in Bulletproofs to help explaining our protocol. H takes coefficients $\vec{a}$ and value $y = f(z)$ as inputs and convert them into a single group element as the following equation shows:

$$H(\vec{a}_{[:n']}, \vec{a}_{[n':]}, y) = \vec{g}_{[:n']}^{\vec{a}_{[:n']}} \cdot \vec{g}_{[n':]}^{\vec{a}_{[n':]}} \cdot u^y \in \vec{G} \tag{10}$$

To make the paper easier to follow, we also define a function H like that in Bulletproofs to help us explain our protocol. H takes coefficients $\vec{a}$ and value $y = f(z)$ as inputs and converts them into a single group element as the following equation shows:

$$L = H(\vec{0}^{n'}, \quad \vec{a}_{[:n']}, \quad \langle \vec{a}_{[n':]}, \vec{z}_{[:n']} \rangle) \tag{11}$$

$$R = H(\vec{a}_{[n':]}, \quad \vec{0}^{n'}, \quad \langle \vec{a}_{[:n']}, \vec{z}_{[n':]} \rangle) \tag{12}$$

$$P = H(\vec{a}_{[:n']}, \quad \vec{a}_{[n':]}, \quad y \quad) \cdot h^{\phi} \quad // \text{ equivalent to } C \cdot u^{y} \tag{13}$$

Both the prover and the verifier use inputs C, y to compute P. L, R are computed by the prover and sent to the verifier. Upon receiving L, R from the prover, the verifier will perform the following steps to compute P':

1 The verifier generates a random challenge $x_j \leftarrow \mathbb{Z}_p^{n'}$ and then send it to the prover
2 Prover computes $\vec{a}' = \vec{a}_{[:n']} + \vec{a}_{[n':]} x_j \in \mathbb{Z}_p^{n'}$ and sends $\vec{a}'$ to the verifier
3 Both prover and verifier compute $\vec{z}' = \vec{z}_{[:n']} + \vec{z}_{[n':]} x_j^{-1} \in \mathbb{Z}_p^{n'}$
4 With $L, R,\ \vec{z}'$, the verifier can compute $P' = L^{x_j^{-1}} P R^{x_j}$, and outputs "accept" if and only if:

$$P' = H(\vec{a}', \quad \vec{a}' x^{-1}, \quad \langle \vec{a}', \vec{z}' \rangle) \tag{14}$$

Note that P' is the P value in the next round of the recursion, which we will explain in the next sub-section.

3.5 Recursive Evaluation

Similar to the inner product argument used in Bulletproofs, we can shrink the polynomial commitment being evaluated here through recursion. Instead of sending $\vec{a}' \in \mathbb{Z}^{n'}$, the prover and the verifier can engage in a recursive protocol to reduce the transcript size by half in each round, until $|\vec{a}| = 1$ in the final round. The full recursion algorithm is shown in Protocol RecursiveEval. Note that the right-hand side of equation 14 is the same as:

$$P' = \vec{g}_{[:n']}^{\vec{a}'} \cdot \vec{g}_{[n':]}^{x_j^{-1} \cdot \vec{a}'} \cdot u^{\langle \vec{a}', \vec{z}' \rangle} \cdot h^{\phi} = (\vec{g}_{[:n']} \circ \vec{g}_{[n':]}^{x_j^{-1}})^{\vec{a}'} \cdot u^{y} \cdot h^{\phi} \tag{15}$$

$(\vec{g}_{[:n']} \circ \vec{g}_{[n':]}^{x_j^{-1}})$ is a list of n' base elements for round $j+1$ in the recursion, and each a_i' is the exponent of base $g_i' = (\vec{g}_i \cdot \vec{g}_{n'+i}^{x_j^{-1}})$ in the next round.

In the final round $j = f$, $|\vec{g}| = 1 \wedge \vec{g} = B$, we have $|\vec{a}| = 1$, $|\vec{z}| = 1$, and $P = B^a h^{\phi} u^{a \cdot z}$. We use a generalized Shnorr's protocol similar to that used in Halo [7] to perform the final check of the protocol. That is, the verifier will validate the statement $f(z) = y$ if the prover can prove it has knowledge of the exponents of B, h, u in P. The final validation steps are described below:

1. The prover generates random secrets $\delta, \epsilon \in \mathbb{Z}_p$ and computes $R = (B \cdot u^z)^{\delta} \cdot h^{\epsilon} \in \mathbb{G}$
2. The prover sends R to the verifier, and upon receiving R, the verifier samples a random challenge c and sends it back to the prover
3. The prover applies challenge c and witnesses a, ϕ to compute s_1, s_2 s.t. $s_1 = a \cdot c + \delta \in \mathbb{Z}_p$, $s_2 = \phi \cdot c + \epsilon \in \mathbb{Z}_p$ and sends s_1, s_2 to the verifier
4. The verifier uses committed values P, R, transcripts s_1, s_2, group bases B, h, u, and z to compute the left and right hand sides of the equality below and passes the validation if the equality holds

$$P^c \cdot R \stackrel{?}{=} (g \cdot u^z)^{s_1} \cdot h^{s_2} \in \mathbb{G}$$

Note that Bulletproofs' inner product protocol requires verifiers to compute all base elements $\vec{g}'$ for every round, an expensive task and the primary reason why it would require $O(n)$ group exponential verifier cost.

3.6 Prover Assisted Logarithmic Computation of Base Element

The most costly operation for the verifier in Bulletproofs' inner product argument is the task of computing base elements $\vec{g}\,', \vec{h}\,'$ in each round of a recursion. Although the second vector $\vec{h}\,'$ is no longer needed after "shrinking" it to a polynomial commitment evaluation protocol, computing $\vec{g}'$ in each round is still expensive and requires the verifier to perform n group exponential operations.

With the knowledge of Section 3.3 in mind, we define the following steps to reduce the verifier's computation cost to 4 log n group exponential operations:

1. During the setup phase, both the prover and the verifier compute $B = \prod_{i=0}^{n-1} g_i$, $B_\delta = \prod_{i=0}^{n-1} g_i^{n^i}$. Note that since g_i for $i = \{0, ..., n-1\}$ is known during the protocol initialization phase, this step should be performed before the protocol is put into use.
2. In round j, the prover computes $B_L = \prod_{i=0}^{n'-1} g_i$ and $B_{\delta L} = \prod_{i=0}^{n'-1} g_i^{n^i}$ and sends them to the verifier.
3. In round j, the verifier computes B' and B'_δ for round $j+1$ using challenge x_j generated for this round. For $B_R = (B/B_L)$, we have:

$$B' = B_L \cdot (B/B_L)^{x_j^{-1}} \in \mathbb{G} \tag{16}$$

$$B'_\delta = B_{\delta L} \cdot (B/B_{\delta L})^{x_j^{-1} \cdot / 2^{n'}} \tag{17}$$

4. In round $j+1$, both the prover and the verifier set B, B_δ of round $j+1$ to B', B'_δ of the earlier round j, and restart from step 2.

The rational is that if we set each $g_{\delta i} = g_i^{2^i}$ (which also implies $g'_{\delta i} = g_i'^{2^i}$), then each ith generator $g'_{\delta i}$ used to create B'_δ can be expressed as:

$$g'_{\delta i} = g_{\delta i} \cdot (g_{\delta n'+i})^{x_j^{-1}} = g_i^{2^i} \cdot (g_{n'+i}^{2^{n'+i}})^{x_j^{-1}}$$

If we factor out each $g'_{\delta i}$ by 2^i power, we get $g_{\delta i}'^{1/2^i}$, which can be written as:

$$g'_i = g_i \cdot (g_{n'+i}^{2^{n'}})^{x_j^{-1}}$$

The equality above is equivalent to the equality 6 explained in the earlier sub-section 3.2. Since the multiplication order of generators in computing B_L and $B_{\delta L}$ does not matter due to associativity, we can also infer that both B_L and $B_{\delta L}$ must be correctly computed as specified in step 2.

The protocol that satisfies the first relation defined in this section is illustrated in Protocol PCEval, which calls Protocol RecursiveEval to recursively run down the statement until we get to $n = 1$ (the number of degree terms in a polynomial), where the verifier will perform the final check to decide whether the statement $f(z) = y$ is valid or not.

```
$Input : (\vec{g}, h, u, C \in \mathbb{G}, z, y, n \in \mathbb{Z}_p; \vec{a}, \phi \in \mathbb{Z}_p)$
    $\mathcal{P}'s\, input : (\vec{g}, h, u, C, z, y, n; \vec{a}, \phi)$
    $\mathcal{V}'s\, input : (\vec{g}, h, u, C, z, y, n)$
        $\mathcal{P}, \mathcal{V}\, Initialize :$    // setup phase, run only once
            $B = \prod_{i=0}^{n-1} g_i \in \mathbb{G}$
            $B_\delta = \prod_{i=0}^{n-1} g_i^{n^i} \in \mathbb{G}$
        $\mathcal{V}\ computes :$
            $x \xleftarrow{\$} \mathbb{Z}_p$
        $\mathcal{V} \to \mathcal{P} : x$
        $\mathcal{P}, \mathcal{V}\, computes :$
            $P = C \cdot u^{x \cdot y} \in \mathbb{G}$
            $\vec{z} = z^0, z^1, ..., z^{n-1} \in \mathbb{Z}_p^n$
    $call$ RecursiveEval $(\vec{g}, h, u^x, P, B, B_\delta, \vec{z}, y, n;\ \vec{a}, \phi)$
```

Protocol PCEval

Protocol PCEval initializes all input parameters of our protocol and then calls Protocol RecursiveEval to recursively evaluate committed polynomial C.

```
$Input : (\vec{g}, h, u, P, B, B_\delta \in \mathbb{G}, \vec{z} \in \mathbb{Z}_p^n, y, n \in \mathbb{Z}_p; \vec{a}, \phi \in \mathbb{Z}_p)$
    $\mathcal{P}'s\, input : (\vec{g}, h, u, P, B, B_\delta, \vec{z}, y, n;\ \vec{a}, \phi)$
    $\mathcal{V}'s\, input : (\vec{g}, h, u, P, B, B_\delta, \vec{z}, y, n)$
    **if** $n = 1 :$
        $\mathcal{V}\ computes :$
            **if** $\boldsymbol{ShnorrVerify}(P, B, z; a, \phi) \wedge B \overset{?}{=} B_\delta :$    $accept$
            **else** $: reject$
    $\mathcal{P}\, computes :$
        $n' = n/2 \in \mathbb{Z}_p$
        $c_L = \sum_{i=0}^{n'-1} a_i z_{n'+i} \in \mathbb{Z}_p,$        $c_R = \sum_{i=0}^{n'-1} a_{n'+i} z_i \in \mathbb{Z}_p$
        $L = \prod_{i=0}^{n'-1} g_{n'+i}^{a_i} \cdot u^{c_L} \in \mathbb{G},$        $R = \prod_{i=0}^{n'-1} g_i^{a_{n'+i}} \cdot u^{c_R} \in \mathbb{G}$
    $B_L = \prod_{i=0}^{n'-1} g_i \in \mathbb{G},$        $B_{\delta L} = \prod_{i=0}^{n'-1} {g_i}^{2^i} \in \mathbb{G}$
$\mathcal{P} \rightarrow \mathcal{V} : L, R, B_L, B_{\delta L}$
$\mathcal{V}$ computes :
    $x_j \xleftarrow{\$} \mathbb{Z}_p$
    $B' = B_L \cdot (B/B_L)^{x_j^{-1}} \in \mathbb{G},$     $B'_\delta = B_{\delta L} \cdot (B/B_{\delta L})^{x_j^{-1}/2^{n'}} \in \mathbb{G}$
$\mathcal{V} \rightarrow \mathcal{P} : x_j$
$\mathcal{P}$ computes :
    $\vec{g}' = \vec{g}_{[:n']} \circ (\vec{g}_{[n':]})^{x_j^{-1}} \in \mathbb{G}^{n'}$
$\mathcal{P}, \mathcal{V}$ computes :
    $P' = L^{x_j^{-1}} \cdot P \cdot R^{x_j}$
    $\vec{a}' = \vec{a}_{[:n']} + \vec{a}_{[n':]} \cdot x_j \in \mathbb{Z}_p^{n'}$
call RecursiveEval$(\vec{g}', h, u, P, B, B_\delta, P', U, \vec{z}', y, n';\ \vec{a}, \phi)$
```

Protocol RecursiveEval

Protocol RecursiveEval calls protocol ShnorrCheck to perform the final check by validating knowledge of exponents on B, h, u match those on P.

```
Input :$(P, B \in \mathbb{G}, z \in \mathbb{Z}_p; a, \phi \in \mathbb{Z}_p)$
    $\mathcal{P}'s$ input : $(P, B \in \mathbb{G}, z \in \mathbb{Z}_p; a, \phi \in \mathbb{Z}_p)$
    $\mathcal{V}'s$ input : $(P, B \in \mathbb{G}, z \in \mathbb{Z}_p)$
        $\mathcal{P}$ computes :
            $\delta, \epsilon \xleftarrow{\$} \mathbb{Z}_p$
            $R = (B \cdot u^z)^\delta \cdot h^\epsilon \in \mathbb{G}$
        $\mathcal{P} \rightarrow \mathcal{V} : R$
        $\mathcal{V}$ computes :
            $c \xleftarrow{\$} \mathbb{Z}_p$
        $\mathcal{V} \rightarrow \mathcal{P} : c$
        $\mathcal{P}$ computes :
            $s_1 = a \cdot c + \delta \in \mathbb{Z}_p$
            $s_2 = \phi \cdot c + \epsilon \in \mathbb{Z}_p$
        $\mathcal{P} \rightarrow \mathcal{V} : s_1, s_2$
        $\mathcal{V}$ validates :
            if $P^c \cdot R \stackrel{?}{=} (g \cdot u^z)^{s_1} \cdot h^{s_2} \in \mathbb{G}$
                return true
            else return false
```

Protocol ShnorrCheck

Corollary 1. *(Polynomial Commitment Evaluation). The scheme presented in Protocol PCEval has perfect completeness, honest verifier zero knowledge, and witness-extended-emulation for either extracting a non-trivial discrete logarithm relation between $\vec{g}, h, u$ or extracting a valid witness $\vec{a}$.*

The proof of corollary 1 is defined in theorem 1, which is an extension of corollary 1.

4 Full Protocol with Complete Logarithmic Verification Time

The protocol introduced in Section 3 achieved logarithmic verifier cost for group exponentiation operations. However, the total asymptotic verification time is still linear because verifiers need to compute $\vec{z}'$ for every iteration. Although field operations are usually considered cheap, they add up when the polynomial gets large. In this section, we will introduce a computation trick that will bring the asymptotic verification cost to $O(\text{log n})$.

4.1 Achieving Logarithmic Verifier for Field Operations

As the degree of a polynomial grows toward 2^{20}, the concrete verifier work will increasingly be dominated by $1n$ + log n field multiplications and $1n$ field additions in protocol PVEval. Furthermore, computing $\vec{z}$ from one z input would require another $1n$ field multiplication operations in $\mathbb{Z}_p$, so there are a total of $2n$ + log n field multiplications the verifier needs to compute in protocol PVEval. In this section we will introduce a computation trick to reduce the verifier field computation cost from $2n + \log n$ in protocol PVEval to $O(\log n)$.

While our focus is on the univariate polynomial where the pattern is well known (the variable in each term has one more degree than that of the previous term), similar techniques may exist for other multi-variate polynomials that have an identifiable pattern.

In each round of the recursion, the verifier needs to compute z'_i for $i = \{1, 2, ..., n'\}$. Computing $|\vec{z}'| = n'$ elements requires n' field addition and n' field multiplication operations. Each z'_i is defined as:

$$z'_1 = z_1 + z_{n'+1} \cdot x_j^{-1} \tag{18}$$
$$z'_2 = z_2 + z_{n'+2} \cdot x_j^{-1} \tag{19}$$
$$... \tag{20}$$
$$... \tag{21}$$
$$z'_{n'} = z_{n'} + z_n \cdot x_j^{-1} \tag{22}$$

Break vector $\vec{z}'$ to left and right sets s.t. left set $= z'_1, ..., z'_{n'/2}$ and right set $= z'_{n'/2+1}, ..., z'_n$. We can observe that the elements of the right set can be computed directly from the elements of the left set by multiplying each element of the left set by $z_{n'/2}$:

$$z'_{n'/2+1} = z'_1 \cdot z_{n'/2} \tag{23}$$
$$z'_{n'/2+2} = z'_2 \cdot z_{n'/2} \tag{24}$$
$$... \tag{25}$$
$$... \tag{26}$$
$$z'_{n'/2+n'/2} = z'_{n'/2} \cdot z_{n'/2} \tag{27}$$

If this trick can be applied recursively, we can reduce the total field operations to $O(\log n)$. In our new setting, we define $z_l,\ z_r$ to represent the leading elements of the left set and the right set, such that in round 1 we have:

$$z_l = z^0 \tag{28}$$

$$z_r = z_l \cdot z^{n/2} \tag{29}$$

Note that z^0 is “1”, which is the constant term in a polynomial f(x). In each of the subsequent rounds, new z_l', z_r' are computed from z_l, z_r of the previous round $j-1$ s.t.:

$$z_l' = z_l + z_r \cdot x_j^{-1} \tag{30}$$

$$z_r' = z_l' \cdot z^{n'/2} \tag{31}$$

This approach allows the verifier to only compute two values in each round. We can visually observe this process in table 1.

Instead of computing all z^k values in $f(X)$ for $k = \{z^0, z^1 ..., z^{n-1}\}$ as in Protocol PCE-val/RecursiveEval, the updated protocol only needs verifiers to pre-compute $z^{n'/2}$ used in each round, then store them in $\vec{e}$ during the initialization phase using equations below:

$$e_1 = z \in \mathbb{Z}_p \tag{32}$$

$$e_{i+1} = e_i^2;\ \text{for } i = \{1, ..., \log n\} \in \mathbb{Z}_p^{\log n} \tag{33}$$

Using Table 1 we can visualize how the final z is computed from this recursion process: We use the subscript j to indicate the round number s.t. $z_{j,i}$ is element z_i in round j. Bold symbol $\boldsymbol{z_{j,0}}$ is the z_l' value of round j, and underlined $\underline{z_{j,n'/2}}$ is the z_r' value of round j.

To make table 1 easier to discern, we don't show the negative exponent on each x_j (e.g. x_j means x_j^{-1} in table 1) for clarity reasons. In round $j = 0$, we have $z_i = z^{i-1}$ for $i \in \{1, ..., n\}$.

Table 1: Logarithmic Computation of z

Recursive Iterations to compute $\boldsymbol{z}$			
$j=1$	$j=2$	$j=3$	j $=4$
$\boldsymbol{z_{1,1}' = z_1 + z_9 x_1}$	$\boldsymbol{z_{2,1}' = z_{1,1}' + z_{1,5}' x_2}$	$\boldsymbol{z_{3,1}' = z_{2,1}' + z_{2,3}' x_3}$	$\boldsymbol{z_{4,1} = z_{3,1}' + z_{3,2}' x_4}$
$z_{1,2}' = z_2 + z_{10} x_1$	$z_{2,2}' = z_{1,2}' + z_{1,6}' x_2$	$\underline{z_{3,2}' = z_{2,2}' + z_{2,4}' x_3}$	
$z_{1,3}' = z_3 + z_{11} x_1$	$\underline{z_{2,3}' = z_{1,3}' + z_{1,7}' x_2}$		
$z_{1,4}' = z_4 + z_{12} x_1$	$z_{2,4}' = z_{1,4}' + z_{1,8}' x_2$		
$\underline{z_{1,5}' = z_5 + z_{13} x_1}$			
$z_{1,6}' = z_6 + z_{14} x_1$			
$z_{1,7}' = z_7 + z_{15} x_1$			
$z_{1,8}' = z_8 + z_{16} x_1$			
$n_1' = 8$	$n_2' = 4$	$n_3' = 2$	$n_4' = 1$

By observing table 1, we can visually see how “missing” values from round $j = 0$ are eventually being captured in the final round:

In round $j = 1$, we only computed $z_{1,1}'$ as z_l' and $z_{1,5}'$ as z_r' (or $z_{1,n'/2+1}$). However, the rest of the uncomputed elements of this round $z_{1,2}', z_{1,3}', z_{1,4}', z_{1,6}', z_{1,7}', z_{1,8}'$ will be counted to the computation of the final value in subsequent rounds.

In round $j = 2,$ computing $z_r'(z_{2,3}')$ from $z_l'(z_{2,1}')$ will count $z_{1,3}', z_{1,7}'$ ($z_{1,(n_2'/2+1)}$, $z_{1,(n_2'/2+1)+n_2'}$) to the computation.

In round $j = 3$, computing $z'_r(z'_{3,2})$ will count $z'_{1,2}, z'_{1,4}, z'_{1,6}, z'_{1,8}$ ($z_{1,(n'_3/2+1)}$, $z_{1,(n'_3/2+1)+1\cdot n'_3}$, $z_{1,(n'_3/2+1)+2\cdot n'_3}$, $z_{1,(n'_3/2+1)+3\cdot n'_3}$) to the recursive computation, which is then used to compute $\boldsymbol{z_{4,1}}$ in the next round ($j = 4$). $\boldsymbol{z_{4,1}}$ is the final z we use in the protocol SchnorrCheck to validate the relation $y = f(z)$.

We are now ready to define the complete protocol in Protocol PCEvalFull/RecursiveEvalFull in the next subsection, and define theorem 1 as follows:

4.2 Full Protocol with Complete Logarithmic Verification Cost

We now present the full protocol for polynomial evaluation with asymptotic logarithmic verifier cost. Note that prover's work is identical to that defined in protocol PCEval and protocol RecursiveEval.

```
$\mathit{Input} : (\vec{g}, h, u, C \in \mathbb{G}, z, y, n \in \mathbb{Z}_p; \vec{a}, \phi \in \mathbb{Z}_p)$
    $\mathcal{P}'s\ \mathit{input} : (\vec{g}, h, u, C, z, y, n; \vec{a}, \phi)$
    $\mathcal{V}'s\ \mathit{input} : (\vec{g}, h, u, C, z, y, n)$
        $\mathcal{P}, \mathcal{V}\ \mathit{Initialize} :$    // setup phase, run only once
            $B = \prod_{i=0}^{n-1} g_i \in \mathbb{G}$
            $B_\delta = \prod_{i=0}^{n-1} g_{\delta i} \in \mathbb{G}$
        $\mathcal{V}\ \mathit{computes} :$
            $e_1 = z \in \mathbb{Z}_p$
            $e_{i+1} = e_i^2$; for $i = \{1, ..., \log n\} \in \mathbb{Z}_p^{\log n}$
            $z_l = z \in \mathbb{Z}_p$
            $z_r = z_l \cdot \vec{e}_{[\log n]} \in \mathbb{Z}_p$
            $x \xleftarrow{\$} \mathbb{Z}_p$
        $\mathcal{V} \to \mathcal{P} : x, z_l, z_r, \vec{e}$
        $\mathcal{P}, \mathcal{V}\ \mathit{computes} :$
            $P = C \cdot u^{x \cdot y} \in \mathbb{G}$
    $\mathit{call}$ RecursiveEvalFull$(\vec{g}, h, u^x, B, B_\delta, P, \vec{e}, z_l, z_r, y, n, 1;\ \vec{a}, \phi)$
```

Protocol PCEvalFull

Protocol PCEvalFull initializes parameters of our protocol using its inputs and then calls Protocol RecursiveEvalFull to recursively evaluate committed polynomial C at evaluation point z s.t. $y = f(z)$.

```
$Input : (\vec{g}, h, u, B, B_\delta, P \in \mathbb{G}, \vec{e}, z_l, z_r, y, n, j \in \mathbb{Z}_p; \vec{a}, \phi \in \mathbb{Z}_p)$
    $\mathcal{P}'s\,input : (\vec{g}, h, u, B, B_\delta, P, \vec{e}, z_l, z_r, y, n, j;\ \vec{a}, \phi)$
    $\mathcal{V}'s\,input : (\vec{g}, h, u, B, B_\delta, P, \vec{e}, z_l, z_r, y, n, j)$
    **if** $n = 1$ :
        $\mathcal{V}$ *computes* :
            **if** $\boldsymbol{ShnorrVerify}(P, B, z; a, \phi) \wedge B \overset{?}{=} B_\delta$ :    *accept*
            **else** : *reject*
    $\mathcal{P}$ *computes* :
        $n' = n/2 \in \mathbb{Z}_p$
        $c_L = \sum_{i=0}^{n'-1} a_i z_{n'+i} \in \mathbb{Z}_p, \qquad c_R = \sum_{i=0}^{n'-1} a_{n'+i} z_i \in \mathbb{Z}_p$
        $L = \prod_{i=0}^{n'-1} g_{n'+i}^{a_i} \cdot u^{c_L} \in \mathbb{G}, \qquad R = \prod_{i=0}^{n'-1} g_i^{a_{n'+i}} \cdot u^{c_R} \in \mathbb{G}$
        $B_L = \prod_{i=0}^{n'-1} g_i \in \mathbb{G}, \qquad B_{\delta L} = \prod_{i=0}^{n'-1} {g_i}^{2^i} \in \mathbb{G}$
    $\mathcal{P} \rightarrow \mathcal{V} : L, R, B_L, B_{\delta L}$
    $\mathcal{V}$ *computes* :
        $x_j \xleftarrow{\$} \mathbb{Z}_p$
        $B' = B_L \cdot (B/B_L)^{x_j^{-1}} \in \mathbb{G}, \qquad B'_\delta = B_{\delta L} \cdot (B/B_{\delta L})^{x_j^{-1}/2^{n'}} \in \mathbb{G}$
        $z'_l = z_l + z_r \cdot x_j^{-1} \in \mathbb{G}$
        **if** $n' \neq 1$ :
            $z'_r = z'_l \cdot \vec{e}_{[\log\, n-j]} \in \mathbb{Z}_p$
        $j' = j + 1 \in \mathbb{Z}_p$    // keeping track of recursion rounds
    $\mathcal{V} \rightarrow \mathcal{P} : x_j$
    $\mathcal{P}$ *computes* :
        $\vec{g}' = \vec{g}_{[:n']} \circ (\vec{g}_{[n':]})^{x_j^{-1}} \in \mathbb{G}^{n'}$
        $\vec{a}' = \vec{a}_{[:n']} + (\vec{a}_{[n':]})x_j \in \mathbb{Z}_p^{n'}$
    $\mathcal{P}, \mathcal{V}$ *computes* :
        $P' = L^{x_j^{-1}} \cdot P \cdot R^{x_j}$
    *call* RecursiveEvalFull$(\vec{g}', h, u, B', B'_\delta, P', \vec{e}, {z_l}', {z_r}', y, n', j';\ \vec{a}, \phi)$
```

Protocol RecursiveEvalFull

Theorem 1. *(Polynomial Commitment Evaluation with Logarithmic Verification Cost). The argument presented in Protocol PCEvalFull/RecursiveEvalFull has perfect completeness, perfect special honest verifier zero knowledge, and witness-extended-emulation for either extracting a non-trivial discrete logarithm relation between $\vec{g}, h, u$ or extracting a valid witness $\vec{a}$.*

The proof of theorem 1 is in Appendix A

Multi-linear/multi-variate Polynomial: The trick we introduced in this section is only designed to accelerate uni-variate polynomial commitment schemes which power most of the popular SNARK protocols. Multi-linear and multi-variate polynomials can have many variable placement patterns, and it is unlikely that one algorithm can fit them all. However, if the variable placement pattern of a multi-linear or multi-variate polynomial can be identified, then there is a non-negligible chance that a sub-linear computation trick can be found.

5 Complexity Analysis and Benchmark

The biggest advantage of our polynomial commitment scheme is that it does not need to use expensive pairing-based or group-of-unknown-order groups to achieve transparency while still providing logarithmic verifier cost and communication cost. While the asymptotic performance of our protocol is comparable to the current state of the art, its concrete verifier and communication costs are almost one order of magnitude more efficient than the current state-of-the-art schemes.

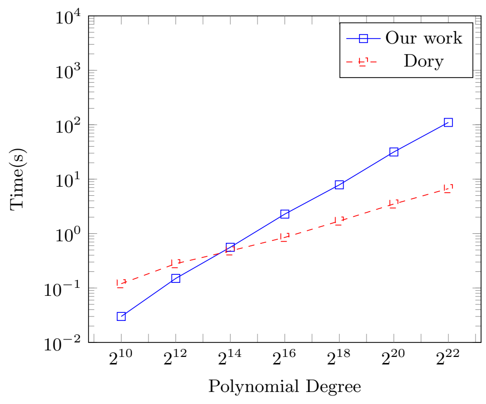
Techniques we use can be easily configured to support batch proof, as mentioned earlier, and can be used to improve the inner product argument in Bulletproofs' Zero Knowledge Range Proof and therefore significantly improve the verifier performance of Bulletproofs' ZKRP.

Table 2: Performance Comparison with Other Polynomial Commitment Schemes

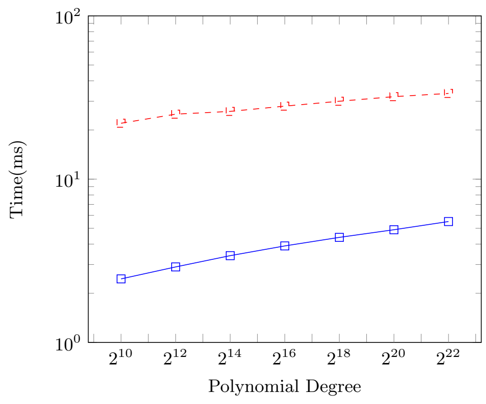
Scheme	Transparent	Prover	Verfier	Proof Size
DARK	●	$O(n \log n)\,\mathbb{G}_u$ exp	$O(n \log n)\,\mathbb{G}_u$ exp	$O(\log n)\mathbb{G}_u$
Bulletproofs	●	$O(n)\,\mathbb{G}$ exp	$O(n)\,\mathbb{G}$ exp	$O(\log n)\,\mathbb{G}$
KATE	○	$O(n)\,\mathbb{G}_1$ exp	2 Pairing	$O(1)\,\mathbb{G}_1$
RS-IOP	●	$O(\lambda n)$ H	$O(\lambda \log^2 n)$ H	$O(\lambda \log^2 n)$ H
Dory	●	$O(n^{1/2})\,P$	$O(\log n)\mathbb{G}_T$ exp	$O(\log n)\mathbb{G}_T$
This Work	●	$O(n)\mathbb{G}$ exp	$O(\log n)\,\mathbb{G}$ exp	$O(\log n)\,\mathbb{G}$

The symbol $\mathbb{G}_u$ denotes a group of unknown order, $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are the first, second, and third groups of a bilinear map (pairing), and H is either the size of a hash output, or the time it takes to compute a hash based on context. Compared to the $\mathbb{G}$ (curve25519 implementation) that our protocol uses, $\mathbb{G}_T$ is approximately 6 times more expensive in size and 10 times more expensive in group exponential operation, and $\mathbb{G}_U$ is approximately 20+ times more expensive in size and 600+ times more expensive in group exponential operation [16]. We neglect the impact of Pippenger style savings in the comparison table.

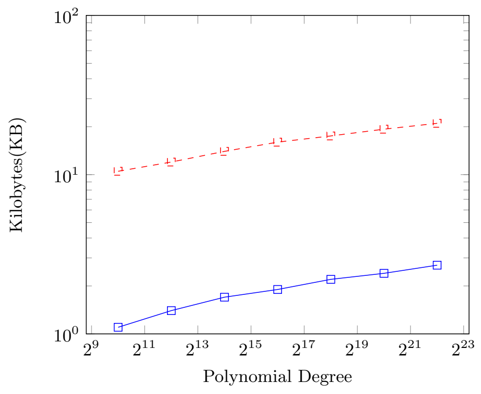
The concrete cost of our protocol is dominated by $4n\,\mathbb{G}$ exponential prover cost, $4\log n\,\mathbb{G}$ exponential verifier cost, and $4\log n\,\mathbb{G}$ communication cost.



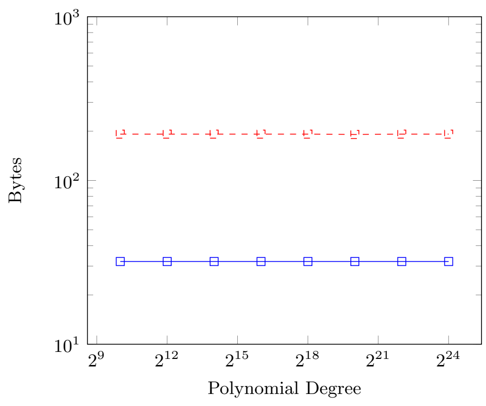
(a) Prover Runtime



(b) Verifier Runtime



(c) Communication Cost



(d) Commitment Size

Now we show the result of our benchmark testing. The test is performed on an Intel (R) Core(TM) i7-9750H CPU @2.60GHz. We wanted to test against Dory (which we believe is the current state of art), but we couldn't find any open sourced code for Dory. Instead, we copied the benchmark numbers from Dory paper and marked them with a red dashed line. While Dory's benchmark is performed on multilinear polynomials, the concrete cost is the same for both multi-linear and univariate polynomials in Dory is $9m + O(1)\,\mathbb{G}_T$ s.t. $m = \frac{1}{2}\log\, n + O(1)$ in verifier cost [16], communication cost is both at 6 log $n\,\mathbb{G}_T$ [16], and prover cost is both dominated by $n^{1/2}P$ [16].

Please note that since Dory's benchmark numbers are presented in graphs, we have to do our best approximation here, and we don't believe the error gap is significant enough to impact our analysis. Also note that Dory's test is performed on an AMD Ryzen 5 3600 CPU @3.60Ghz. Since both tests are run in single-threaded mode, we believe the differences in processing power should be minor not impacting our analysis.

Our test shows our work (blue line) is better in almost all categories except the prover cost for large circuits $n > 2^{12}$. This is expected since Dory offers square root asymptotic prover cost. However, our potorocol offers consistent $\geq$6X improvement on verifier cost, $\approx$8X improvement on communication cost, and $\approx$6X improvement on commitment size.

$\geq$6X improvement on verifier cost is comparable to the difference between $9m + O(1)\,\mathbb{G}_T$ of Dory for univariate polynomial and $4\log n + 2\,\mathbb{G}$ of our work. (assuming the group exponential cost of $\mathbb{G}_T$ is $\geq$ 10X the group exponential cost of $\mathbb{G}$ in curve25519, our protocol is about 16X more efficient than Dory).

$\approx$8X improvement on communication cost is strictly constant to the difference between $(6m + 7)\,\mathbb{G}_T + (3m+3)(\mathbb{G}_2 + \mathbb{G}_1) + 8\mathbb{F}$ of Dory for univariate polynomial and $(3\log n + 1)\,\mathbb{G} + 2\mathbb{F}$ of our work,

Finally, the $\approx$6X commitment size saving is exactly the size difference between $\mathbb{G}$ and $\mathbb{G}_t$.

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. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for R1CS. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 103–128. Springer, Heidelberg, Germany, Darmstadt, Germany (May 19–23, 2019). https://doi.org/10.1007/978-3-030-17653-2_4
3. Block, A.R., Holmgren, J., Rosen, A., Rothblum, R.D., Soni, P.: Time- and space-efficient arguments from groups of unknown order. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 123–152. Springer, Heidelberg, Germany, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84259-8_5
4. Block, A.R., Holmgren, J., Rosen, A., Rothblum, R.D., Soni, P.: Time- and space-efficient arguments from groups of unknown order. Cryptology ePrint Archive, Report 2021/358 (2021), `https://eprint.iacr.org/2021/358`
5. 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
6. Bootle, J., Chiesa, A., Hu, Y., Orrù, M.: Gemini: Elastic SNARKs for diverse environments. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 427–457. Springer, Heidelberg, Germany, Trondheim, Norway (May 30 – Jun 3, 2022). https://doi.org/10.1007/978-3-031-07085-$3_1$5
7. Bowe, S., Grigg, J., Hopwood, D.: Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021 (2019), `https://eprint.iacr.org/2019/1021`
8. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy. pp. 315–334. IEEE Computer Society Press, San Francisco, CA, USA (May 21–23, 2018). https://doi.org/10.1109/SP.2018.00020
9. 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
10. 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
11. Chiesa, A., Ojha, D., Spooner, N.: Fractal: Post-quantum and transparent recursive proofs from holography. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 769–793. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-$1_2$7
12. Dobson, S., Galbraith, S.D., Smith, B.: Trustless groups of unknown order with hyperelliptic curves. Cryptology ePrint Archive, Report 2020/196 (2020), `https://eprint.iacr.org/2020/196`

13. 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`
14. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg, Germany, Vienna, Austria (May 8–12, 2016). https://doi.org/10.1007/978-3-662-49896-5$_1$1
15. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg, Germany, Singapore (Dec 5–9, 2010). https://doi.org/10.1007/978-3-642-17373-8$_1$1
16. Lee, J.: Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part II. LNCS, vol. 13043, pp. 1–34. Springer, Heidelberg, Germany, Raleigh, NC, USA (Nov 8–11, 2021). https://doi.org/10.1007/978-3-030-90453-1$_1$
17. 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
18. 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$5
19. Xie, T., Zhang, Y., Song, D.: Orion: Zero knowledge proof with linear prover time. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 299–328. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 15–18, 2022). https://doi.org/10.1007/978-3-031-15985-5$_1$1
20. Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: 2020 IEEE Symposium on Security and Privacy. pp. 859–876. IEEE Computer Society Press, San Francisco, CA, USA (May 18–21, 2020). https://doi.org/10.1109/SP40000.2020.00052

Appendiex

A. Proof for Theorem 1

Proof. Perfect completeness follows because Protocol PCEvalFull is an instance for the relation $\mathcal{R}_{Eval}$ defined in definition 7 and refined in Section 3.2 that calls Protocol RecursiveEvalFull to recursively process the polynomial $f(z)$ and evaluation point z, which is trivially complete.

To prove PSHVZK, we define a simulator $\mathcal{S}$ to prove that it can simulate all transcripts indistinguishable from those created by a valid prover. We also define another simulator $\mathcal{S}_{DL}$ to simulate transcripts for the Proof of Discrete Log protocol, which we use Schnorr's protocol to implement in our system.

Once the recursion starts, simulator $\mathcal{S}$ randomly generates proof elements $L, R, B_L B_{\delta L} \in \mathbb{G}$ for each round regardless of what challenge x_j received from the verifier.

Once the process reaches the final round when $n = 1$, the simulator first sends some random R to the verifier to receive the challenge c and then rewind. Using challenge c the simulator obtains R_d, R_e from randomly generated s_1, s_2 s.t.

$$R_d = (B \cdot u^z)^{s_1}/(B \cdot u^z)^{a \cdot c} \in \mathbb{G}$$
$$R_e = h^{s_2}/h^{\phi \cdot c} \in \mathbb{G}$$

R_d is equivalent to $(B \cdot u^z)^{\delta}$ for some unknown δ and R_e is equivalent to h^{ϵ} for some unknown ϵ. The simulator then reconstructs new $R^* = R_d \cdot R_e$ and send it to the verifier. The verifier then use R^* to pass the validation test since:

$$P^c \cdot R^* = (B \cdot u^z)^{ac+\delta} \cdot h^{\phi c+\epsilon} \in \mathbb{G}$$

With challenge c known, we can simulate transcripts R^*, s_1, s_2 indistinguishable from any real prover. We therefore conclude Protocol PCEvalFull/RecursiveEvalFull is PSHVZK.

To prove knowledge soundness, we first construct an extractor $\mathcal{X}_p$ for Protocol RecursiveEvalFull and show that it either extracts witnesses $\vec{a}, \phi$ or discovers a non-trivial discrete log relation among $\vec{g}, h, u$. For each recursive step, we demonstrate that on inputs $(\vec{g}, h, u, P, y, z, n)$, the extractor can either efficiently extract witness $\vec{a}$ from the prover or show a non-trivial discrete log relation among $\vec{g}, h, u$.

In the final round of the recursion, when $|\vec{g}| = 1 \wedge n = 1$, the process reaches protocol ShnorrVerify. After receiving R from the prover, the extractor $\mathcal{X}_p$ generates challenges c_1 and gets the first pair s_{11}, s_{12} from the prover and then rewinds to get the second pair s_{21}, s_{22} from the prover using the second challenge c_2. It is trivial to retrieve witness a using challenges c_1, c_2 and transcripts s_{11}, s_{21} since $s_{11} - s_{21} = (ac_1 + \delta) - (ac_2 + \delta) = a(c_1 - c_2)$, and to extract witness ϕ from c_1, c_2 and s_{12}, s_{22} since $s_{12} - s_{22} = (\phi c_1 + \epsilon) - (\phi c_2 + \epsilon) = \phi(c_1 - c_2)$. With a, ϕ we can just check if the equality holds:

$$P = B^a h^\phi u^{a \cdot z}$$

If it is not true, then we get a non-trivial discrete log relation among B, h, u.

For each of the recursive steps, the extractor $\mathcal{X}_p$ communicates with the prover and gets L, R, B_R in each round. By rewinding the prover four times with four different challenges $x_{j1}, x_{j2}, x_{j3}, x_{j4}$ in which $x_{ji} \neq x_{jk}$ for $1 \leq i < k \leq 4$, the extractor obtains four pairs of $\vec{a}'_i \in \mathbb{Z}_p$ that satisfies the equation:

$$L^{x_{ji}^{-1}} \cdot P \cdot R^{x_{ji}} = \prod_{i=0}^{n'-1} (\, g_i \cdot g_{n'+i}^{x_{ji}^{-1}})^{\vec{a}\,'} \cdot u^{\langle \vec{a},\, \vec{z} \rangle} \cdot h^\phi \tag{34}$$

We can use the first three challenges x_{j1}, x_{j2}, x_{j3} to compute $w_1, w_2, w_3 \in \mathbb{Z}_p$

$$v_1 = \sum_{i=1}^{3} w_i \cdot x_{ji}^{-1} = 1, \quad v_2 = \sum_{i=1}^{3} w_i = 0 \quad v_3 = \sum_{i=1}^{3} w_i \cdot x_{ji} = 0 \tag{35}$$

Taking the linear combination of w_1, w_2, w_3 and transcripts L, R received from the protocol, the extractor can compute $\vec{a}_L$ and c_L s.t. $L = \vec{g}^{\vec{a}_L} u^{c_L}$. From equation 11, we see that $\vec{a}_L = (\,\vec{0}^{n'} \,||\, \vec{a}_{[:n']})$. With different choices of w_1, w_2, w_3 for $v_1 = 0, v_2 = 1, v_3 = 0$ and $v_1 = 0, v_2 = 0, v_3 = 1$, the extractor also gets $\vec{a}_R, c_R$ and $\vec{a}_P, c_P$ s.t. $R = \vec{g}^{\vec{a}_R} u^{c_R}$, and $P = \vec{g}^{\vec{a}_P} u^{c_P} h^\phi$. With equation 11, we see that:

$$\begin{aligned} \vec{a}_L &= (\quad \vec{0}^{n'} \quad || \quad \vec{a}_{[:n']} \quad) \\ \vec{a}_R &= (\quad \vec{a}_{[n':]} \quad || \quad \vec{0}^{n'} \quad) \\ \vec{a}_P &= (\quad \vec{a}_{[:n']} \quad || \quad \vec{a}_{[n':]} \quad) \end{aligned}$$

We now rewrite the equation above to:

$$\vec{g}^{\vec{a}_L \cdot x_j^{-1} + \vec{a}_P + \vec{a}_R \cdot x_j} \cdot u^{c_L \cdot x_j^{-1} + c_P + c_R \cdot x_j} \cdot h^\phi = \vec{g}_{[:n']}^{\vec{a}'} \vec{g}_{[n':]}^{x_j^{-1} \cdot \vec{a}'} \cdot u^{\langle \vec{a}\,',\ \vec{z}\,' \rangle} \cdot h^\phi \tag{36}$$

The exponents of the right-hand side of equation **??** is :

$$\begin{aligned} \vec{a}\,' &= \vec{a}_{L,[:n']} \cdot x_j^{-1} + \vec{a}_{P,[:n']} + \vec{a}_{R,[:n']} \cdot x_j \\ \vec{a}\,' \cdot x_j^{-1} &= \vec{a}_{L,[n':]} \cdot x_j^{-1} + \vec{a}_{P,[n':]} + \vec{a}_{R,[n':]} \cdot x_j \\ \langle \vec{a}\,',\ \vec{z}\,' \rangle &= c_L \cdot x_j^{-1} + c_P + c_R \cdot x_j \end{aligned} \tag{37}$$

The first line of equation **??** is the exponents of base $\vec{g}_{[:n']}$, the second line is the exponents of base $(\vec{g}_{[n':]})^{x_{ji}^{-1}}$, and the third line is the exponent of base u, all of which are the bases of the equation on the right hand side of equality 36. If any of these equalities do not hold, then we obtain a non-trivial discrete logarithm relation among generators $(g_0, ..., g_{n-1}, u)$.

Since $\vec{a}'$ is a shared exponent, we can rewrite the right-hand side of equality 36 to:

$$\vec{g}'^{\vec{a}'} \cdot u^{\langle \vec{a}', \vec{z}' \rangle} \cdot h^{\phi} = (\vec{g}_{[:n']} \circ \vec{g}_{[n':]}^{x_{ji}^{-1}})^{\vec{a}'} \cdot u^{\langle \vec{a}', \vec{z}' \rangle} \cdot h^{\phi} \tag{38}$$

Where as each $g'_i = g_i \cdot g_{i+n'}{}^{x_{ji}^{-1}}$ for $i \in \{1, ..., n'\}$. The verifier must be able to compute (the product of) generators $\vec{g}'$ from transcript B_L as equation 16 specified s.t.

$$\prod_{i=0}^{n'-1} \vec{g}' = B' = B_L \cdot (B_R)^{x_{ji}^{-1}} \tag{39}$$

Rewriting the right hand side of the equality above, we can see each ith term in B' is equivalent to g'_i as shown below.

$$\prod_{i=0}^{n'-1} \vec{g}' = \prod_{i=0}^{n'-1} g_i \cdot g_{n'+i}{}^{x_{ji}^{-1}} \tag{40}$$

Where each g'_i maps to the product of the ith item of $\vec{g}$ and $n' + i$th of $\vec{g}$, this implies the ith term of B_L must be g_i and the ith term of B_R is $g_{n'+i}$:

$$g'_i = g_i \cdot (g_{n'+i})^{x_{ji}^{-1}} \tag{41}$$

However, this also implies that if the dishonest prover found a list $\vec{g}^*$ that also satisfies the right-hand side of equality 38, then setting the ith term of B_L to g^*_i and the ith term of B_R to $g^*_{n'+i}$ would also pass the validation.

The verifier computes B'_δ from $B_{\delta L}$ according to equation 17 as follows:

$$\prod_{i=0}^{n'-1} \vec{g}'^{2^i} = B'_\delta = B_{\delta L} \cdot (B_\delta / B_{\delta L})^{x_{ji}^{-1}/2^{n'}} \tag{42}$$

Rewriting the right-hand side of the equality above, we can see each ith term in B'_δ is equivalent to $g'^{2^i}_i$ as shown below.

$$\prod_{i=0}^{n'-1} \vec{g}'^{2^i} = \prod_{i=0}^{n'-1} g_i^{2^i} \cdot (g_{n'+i}^{2^{n'+i}})^{x_{ji}^{-1}/2^{n'}} \tag{43}$$

For each ith term, we factor out the exponent 2^i to get the following:

$$g'_i = g_i \cdot (g_{n'+i}^{2^{n'}})^{x_{ji}^{-1}/2^{n'}} \tag{44}$$

The only way for both the equality 41 and the equality 44 to compute to the same g'_i is when the ith term of both B_L and $B_{\delta L}$ (after factoring out the exponent) is g_i (not g^*_i) because such g^*_i cannot be found as we explained in Section 3.2. Since the multiplication order of generators in B_L and $B_{\delta L}$ does not matter due to associativity, we can also infer that both B_L and $B_{\delta L}$ must be legit for equalities 39 and 42 to be true.

From the first two lines of equations **??**, we can conclude that for each challenge $\{x_{j1}, x_{j2}, x_{j3}, x_{j4}\}$ that:

$$\vec{a}_{L,[:n']} \cdot x_j^{-1} + (\vec{a}_{R,[:n']} - \vec{a}_{P,[n':]}) \cdot x_j + (\vec{a}_{P,[:n']} - \vec{a}_{L,[n':]}) + \vec{a}_{R,[n':]} \cdot x_j^2 = 0 \tag{45}$$

Here we can conclude that the only way for the equality in 36 to hold for all challenges $x_{j1}, x_{j2}, x_{j3}, x_{j4} \in \mathbb{Z}_p$ is if:

$$\begin{aligned} \vec{a}_{L,[:n']} &= \vec{a}_{R,[n':]} = 0 \\ \vec{a}_{R,[:n']} &= \vec{a}_{P,[n':]} \\ \vec{a}_{P,[:n']} &= \vec{a}_{L,[n':]} \end{aligned} \tag{46}$$

By applying the relations above to the first equality defined in 37 , we can see that for every $x_j \in \{x_{j1}, x_{j2}, x_{j3}, x_{j4}\}$ that $\vec{a}' = \vec{a}_{P,[:n']} + \vec{a}_{P,[n':]} \cdot x_j$.

Using these values, we see that the last equality of 37 can be represented as:

$$\begin{aligned} c_L \cdot x_j^{-1} + c_P + c_R \cdot x_j = \langle \vec{a}',\ \vec{z}' \rangle \\ &= \langle \vec{a}_{P,[:n']},\ \vec{z}_{[:n']} \rangle + \langle \vec{a}_{P,[n':]},\ \vec{z}_{[n':]} \rangle + x \cdot \langle \vec{a}_{P,[n':]},\ \vec{z}_{[:n']} \rangle \\ &\quad + x_j^{-1} \cdot \langle \vec{a}_{P,[:n']},\ \vec{z}_{[n':]} \rangle \\ &= \langle \vec{a}_P,\ \vec{z} \rangle + x_j \cdot \langle \vec{a}_{P,[n':]},\ \vec{z}_{[:n']} \rangle + x_j^{-1} \cdot \langle \vec{a}_{P,[:n']},\ \vec{z}_{[n':]} \rangle \end{aligned}$$

This equation holds for all $x_j \in \{x_{j1}, x_{j2}, x_{j3}, x_{j4}\}$, and we can conclude that $\langle \vec{a}_P,\ \vec{z}\ \rangle = c_P = \langle \vec{a},\ \vec{z}\ \rangle$, or we obtain a non-trivial discrete logarithm relation among generators $(g_0, ..., g_{n-1}, u)$.

Finally, after exiting recursion, we show that using Protocol PCEvalFull we can construct an extractor $\mathcal{X}$ that uses the extractor $\mathcal{X}_p$ of Protocol RecursiveEvalFull. The behavior of $\mathcal{X}$ is similar to that of Bulletproofs [**?**]. $\mathcal{X}$ runs the prover with a challenge x_1 and then uses the extractor $\mathcal{X}_p$ to extract witness $\vec{a}_1, \phi$ such that $C \cdot u^{x_1 \cdot y} = \vec{g}^{\vec{a}_1} h^{\phi} u^{x_1 \cdot y}$. Rewinding the prover with a new challenge x_2 and running the extractor $\mathcal{X}_p$ again to extract a second witness $\vec{a}_2$ such that $C \cdot u^{x_2 \cdot y} = \vec{g}^{\vec{a}_2} h^{\phi} u^{x_2 \cdot y}$. The soundness implies that we can either compute $u^{(x_1 - x_2) \cdot y} = \vec{g}^{\vec{a}_1 - \vec{a}_2}\ u^{x_1 \cdot y - x_2 \cdot y}$, or we get a non-trivial discrete logarithmic relation among $\vec{g}$ and u, or we get $u^{(x_1 - x_2) \cdot y}$ such that $y = \langle \vec{a}, \vec{z} \rangle$. Therefore, $\vec{a}$ is a valid witness to the relation. The extractor $\mathcal{X}$ is efficient because it only uses extractor $\mathcal{X}_p$ twice, therefore, we can conclude that the protocol has witness-extended emulation based on the forking lemma.