Watermarking Proofs with Provenance-Binding ZK-SNARKs

Wenhao Wang
Yale University, IC3

Fan Zhang
Yale University, IC3

Charalampos Papamanthou
Yale University

***Abstract*—The emerging ecosystem of outsourced proving services, decentralized prover networks, and prover marketplaces raises a new requirement for ZK-SNARKs: the need to *watermark* proofs with provenance information, such as the prover identity and a fresh request-time nonce. The prover identity records who should receive credit for the proof, while the nonce binds the proof to a particular request and is intended to prevent the prover from reusing work performed before that request was created. The key security requirement is that a prover must perform fresh work—comparable to computing the proof from scratch—when generating a proof for a fresh provenance, even if the attacker may have generated the same proofs before or performed precomputation.**

Prior attempts, such as Succinct Network, proposed embedding provenance information in the Fiat-Shamir challenges, though without a rigorous security definition or analysis. We show that this approach is not secure and is vulnerable to *preprocessing attacks*, in which an adversary can precompute provenance-independent information that can later be used to reduce the effort required to generate proofs for a fresh provenance. We instantiate these attacks against KZG-PLONK.

To arrive at a secure construction, we propose the first formal definition for *provenance-binding ZK-SNARKs* to capture the above desired property. We present a generic construction that augments code-based polynomial commitments, such as BaseFold and FRI, with provenance binding. Compiling them with Poly-IOPs yields provenance-binding ZK-SNARKs. Our construction incurs minimal overhead.

1. Introduction

Zero-knowledge succinct non-interactive arguments of knowledge (ZK-SNARKs) enable verification of computations from a short proof, with a wide range of applications, including privacy-preserving payments, blockchain scalability, interoperability, verifiable virtual machines, and verifiable ML [10], [11], [13], [15], [26].

Because generating SNARK proofs is computationally expensive, *proof markets* [28] have emerged as platforms that connect provers with computational resources and users who pay for proofs. Real-world examples include Fermah [1], Succinct Network [25], Lagrange's ZK Prover Network [2], RISC Zero's Boundless proving marketplace [4], and Marlin's Kalypso [3].

While the exact market mechanisms differ across these systems, a common goal is to reward provers for the computational effort they spent on generating proofs. Achieving this requires a new SNARK property that has not been studied before.

In particular, for any reward mechanism to work, it is essential to prevent a malicious prover from earning the same reward by spending *less* effort than an honest prover. Prior works have studied the notion of *unstealability* [27] in the context of polynomial commitments, which ensures that a prover cannot steal another prover's proofs and rewards. While necessary, unstealability is not sufficient in proof markets: it does not prevent a malicious prover from creating multiple identities and posing as different provers. This is because in proof markets, identities are usually public keys, making it easy to create Sybils; moreover, markets explicitly reward multiple provers for the same proof task to incentivize participation.

Yet another concern is preventing the reuse of work across proof jobs, where a prover saves computation results from a prior job and repeats them later, framed as the "reused work" problem in [25].

We propose *proof watermarking* to address these concerns and serve as the basis for proof markets. Specifically, we study how to bind a proof to a provenance string ρ, in a way that a malicious prover cannot produce a proof for a fresh provenance string ρ' by performing significantly less work than an honest prover, even though the prover has previously produced proofs for other provenance strings. The provenance string is meant to capture information such as prover identity, which records who should receive credit for the proof, and a fresh request-time nonce, which binds the proof to a particular job. We refer to this new property of SNARKs as *provenance-binding*.

1.1. Provenance-Binding ZK-SNARKs

To capture our proposed provenance-binding property, we formalize *provenance-binding* (ZK)-SNARKs. A provenance-binding (ZK)-SNARK is a ZK-SNARK whose proving and verification algorithms take an additional provenance string ρ, while preserving the properties of completeness, knowledge soundness, and zero-knowledge (optional) in (ZK)-SNARKs. The main extra security requirement is an online-work lower bound. We allow an adversary to run an arbitrary polynomial-time preprocessing algorithm before seeing ρ, a fresh provenance string. This preprocessing may depend on the proving key, verification key, circuit, public

input, and witness. Then, the online adversary receives ρ and must output an accepting proof.

Let t denote the honest online proving time after ρ is known. For a function $\sigma(t) \leq t$, we say a scheme is $\sigma(t)$-provenance-binding if no such adversary can produce a valid proof in less than $\sigma(t)$. It is desired that $\sigma(t)$ is of $\Omega(t)$, since it means that the prover needs to perform linear work proportional to the honest work.

We stress again that this notion is different from unstealability proposed in [27]. Both unstealable proofs and provenance-binding proofs require that an adversary cannot produce a valid proof under an arbitrary provenance string ρ (e.g., prover identity) without redoing the proof computation from scratch. The two notions differ in the adversary's input: In unstealable proofs, the adversary is given just the watermarked proofs, but the internal state of the prover remains hidden. In provenance-binding proofs, the adversary is given the watermarked proofs together with the prover's internal state, including intermediate computation results from generating other proofs (in fact, the adversary is the prover). Provenance binding is therefore a fundamentally different and strictly stronger notion—yet it is exactly what is needed in prover market applications to prevent a malicious prover from being compensated multiple times for the same proof by re-watermarking it under Sybil identities. We elaborate on this in Section 7.

1.2. Adding Provenance String to Fiat-Shamir Transform with Is Not Enough

At first glance, achieving provenance-binding seems easy: one can hash the provenance string ρ into the Fiat-Shamir transcript, as proposed by Succinct Network [25]. However, we show that this approach is insecure for proof systems such as KZG-PLONK due to *preprocessing*. In particular, for a fixed witness and SRS, the prover can preprocess the provenance-independent part in advance and finish the proof quickly for different provenance strings.

Consider the opening of the KZG polynomial commitment scheme [16] at a random point where the random point is derived from the provenance information. For a polynomial f, the opening proof at a point α is $\pi_\alpha = g_1^{\frac{f(\tau)-f(\alpha)}{\tau-\alpha}}$. Equivalently, let $Q(X,Y) := \frac{f(X)-f(Y)}{X-Y}$, then $\pi_\alpha = g_1^{Q(\tau,\alpha)}$. Therefore, given a polynomial f, all possible opening proofs form a group-valued function with α as the variable. If the provenance string only determines α, then the prover preprocesses this function before ρ is known and later evaluates it faster online.

We turn this observation into concrete preprocessing attacks. Let ℓ_p denote the log size of the field and d be the degree of the polynomial. For KZG openings, we express the proof as a function of the opening point α, written as $\pi_\alpha = \sum_{i=0}^{d-1} c_i(\alpha) g_i$, where the coefficients of the functions $c_i(\alpha)$ depend only on the polynomial f. By partitioning these coefficient functions into consecutive groups and preprocessing their evaluations, the online work drops to $O(d\ell_p/\log d)$, while the honest time without preprocessing is $\Omega(d\ell_p)$.

Then we extend the attack to KZG-PLONK and present a preprocessing attack against a real-world SNARK protocol. We apply this idea to the permutation check so that preprocessing makes the online polynomial evaluations, commitments, and openings faster. Our preprocessing attack computes the standard permutation check commitment and opening in online time $O(n\ell_p/(\log n)^{1/3})$ when proving a circuit of size n. Compare to the honest prover time $t_{\mathsf{hon}} = O(n\ell_p)$, the online time is $o(t_{\mathsf{hon}})$. The above attacks show that KZG-PLONK is not $c \cdot t$ provenance binding for any constant c.

In our concrete implementation of the attack described in Section 6.1, preprocessing speeds up KZG commitment and opening at a random point by about $3\times$. This gives initial evidence that concrete preprocessing attacks are possible, and further engineering could improve the speedup (since the online time is bound by any constant speedup factor). Since our positive constructions, presented next, incur only low overhead, even this preliminary evidence motivates adopting the fix. We leave further optimization of the attack to future work.

1.3. Provenance-Binding SNARK from Code-Based Polynomial Commitment Schemes

To obtain provenance binding with low overhead, we turn to code-based polynomial commitments such as FRI [6] and BaseFold [30]. These systems have a different structure from KZG, as the prover commits to long vectors of encoded polynomial coefficients using Merkle trees, and the verifier checks the proof by opening a small number of random positions and checking local consistency constraints.

This commitment structure gives a natural way to force fresh provenance-dependent work. We mask every committed vector entry with a random value derived from the provenance string. For each oracle level i and coordinate j, define a public random mask $\boldsymbol{r}_i^\rho[j]$. If $\boldsymbol{\pi}_i[j]$ is the vector value, the prover commits instead to $\boldsymbol{\omega}_i^\rho[j] := \boldsymbol{\pi}_i[j] + \boldsymbol{r}_i^\rho[j]$. With this, the construction of the Merkle Tree, which takes the majority of the time in the PCS, cannot be preprocessed. The construction adds minimal overhead to the original proof system: the prover adds masks before Merkle commitment, while the verifier subtracts masks only at the queried positions. Based on this, we build **PB-Code**, a provenance-binding SNARK. Theorem 2 shows that **PB-Code** is $c \cdot t$-provenance-binding for some constant c. We show in Section 6.2 that the constant is at least 0.41 when using **PB-Code** to prove a circuit of size 2^{20}; when only the PCS in **PB-Code** is used, the constant is at least 0.83.

1.4. Our Contributions

In summary, we make the following contributions.

1) **Formal definition of provenance-binding ZK-SNARKs** (Section 3): We give a formal definition for provenance-binding (ZK)-SNARKs.

2) **Preprocessing attacks** (Section 4): We show that embedding provenance information into Fiat-Shamir challenges is not sufficient for KZG-PLONK.
3) **A generic construction with code-based polynomial commitments** (Section 5): We give a provenancing-binding construction using code-based PCS with minimal overhead, and prove that it satisfies our definition.

2. Preliminaries

2.1. KZG Polynomial Commitment Scheme

Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ be bilinear groups of prime order p with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, and let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a non-degenerate bilinear map. Let $\mathbb{F}_p$ be the scalar field, let $\tau \xleftarrow{\$} \mathbb{F}_p$ be the structured reference string trapdoor, and let n be the supported degree bound. The powers-of-τ KZG SRS is $\mathsf{SRS} = (g_1^{\tau^0}, g_1^{\tau^1}, \ldots, g_1^{\tau^n}, g_2, g_2^{\tau})$.

For a polynomial $f(X) = \sum_{i=0}^{n} f_i X^i \in \mathbb{F}_p[X]$, the KZG commitment is $\mathsf{Com}(f) := \mathsf{cm}_f := g_1^{f(\tau)} = \prod_{i=0}^{n} (g_1^{\tau^i})^{f_i}$. To open cm_f at $\alpha \in \mathbb{F}_p$, the prover outputs the claimed value $\beta = f(\alpha)$ and the proof $\pi_\alpha := g_1^{q_\alpha(\tau)}$, where $q_\alpha(X) := (f(X) - \beta)/(X - \alpha)$. The verifier accepts iff $e(\mathsf{cm}_f / g_1^{\beta}, g_2) \stackrel{?}{=} e(\pi_\alpha, g_2^{\tau - \alpha})$.

Additive notation for group operations. In the following sections, we write group operations additively: $[a]_1 := g_1^a$, $[a]_1 + [b]_1 = [a+b]_1$, and $c \cdot [a]_1 = [ca]_1$. When using additive notation, we write the SRS elements as $g_i := [\tau^i]_1$. Thus $\mathsf{cm}_f = \sum_{i=0}^{n} f_i g_i = [f(\tau)]_1$.

2.2. Preprocessing Tables for Fixed-Base Evaluation

Our fixed-base preprocessing tool is a preprocessing table, following the techniques of fixed-base exponentiation with precomputation [8], [18].

Let $V(\mathbf{x}) = \sum_{\mu \in \mathcal{M}} V_\mu m_\mu(\mathbf{x})$ be either a field-valued polynomial over $\mathbb{F}_p$ or a group-valued polynomial over $\mathbb{G}$, where $\mathcal{M}$ is the set of its monomials and $M := |\mathcal{M}|$. For every $\eta \in \{0,1\}^M$, we preprocess the valuations

$$\mathsf{Tab}_V[\eta] := \sum_{\mu \in \mathcal{M}} \eta_\mu V_\mu.$$

After $\mathbf{x}$ is known, we can compute the monomial evaluations $m_\mu(\mathbf{x})$. We further write $m_\mu(\mathbf{x}) = \sum_{\ell=0}^{\ell_p - 1} m_{\mu,\ell} 2^\ell$, where $m_{\mu,\ell} \in \{0,1\}$ and $\ell_p := \lceil \log_2 p \rceil$. For each bit position ℓ, define the mask $\eta_\ell := (m_{\mu,\ell})_{\mu \in \mathcal{M}} \in \{0,1\}^M$. Then we have the evaluation can be written as

$$V(\mathbf{x}) = \sum_{\ell=0}^{\ell_p - 1} 2^\ell \cdot \mathsf{Tab}_V[\eta_\ell].$$

Therefore, after the monomial values are known, the online evaluation uses $O(\ell_p)$ field or group operations. The preprocessing table has size $O(2^M)$.

Costs. Suppose the largest polynomial has $M = O(b^r)$ monomials and there are N/b blocks. The preprocessing costs $O\left(\frac{N}{b} \cdot 2^{O(b^r)}\right)$, while the online work costs $O\left(\frac{N}{b} \cdot \ell_p\right)$. Choosing $b = \Theta((\log N)^{1/r})$ gives preprocessing work polynomial in N, and online work $O\left(\frac{N \ell_p}{(\log N)^{1/r}}\right)$. In Section 4, we use this technique with $r = 1$ for univariate KZG opening at a random point, $r = 2$ for PLONK permutation check commitment, and $r = 3$ for PLONK permutation check opening at a random point.

3. Provenance-Binding ZK-SNARKs

Let $\mathcal{C}$ be an arithmetic circuit over $\mathbb{F}$, and let $\mathcal{R}_\mathcal{C}$ be the associated NP relation. We write $(x, w) \in \mathcal{R}_\mathcal{C}$ if w is a valid witness for public input x. The setup algorithm outputs a provenance space $\mathcal{I}$ from which ρ is sampled.

Definition 1 (Provenance-binding (ZK-)SNARK)**.** *A provenance-binding (ZK-)SNARK for $\mathcal{C}$ is a tuple of algorithms* $\mathsf{PB} := (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *with* $\mathsf{Setup}(1^\lambda, \mathcal{C}) \to (\mathsf{pk}, \mathsf{vk}, \mathcal{I})$, $\mathsf{Prove}(\mathsf{pk}, x, w, \rho) \to \pi_\rho$, *and* $\mathsf{Verify}(\mathsf{vk}, x, \rho, \pi_\rho) \to \{0,1\}$. *The provenance string space $\mathcal{I}$ satisfies $|\mathcal{I}| \geq 2^\lambda$. The provenance string $\rho \in \mathcal{I}$ is supplied to both the prover and the verifier, but it is not part of the NP relation. The algorithms must satisfy completeness, knowledge soundness, succinctness, and, when claimed, zero knowledge, as defined below. In addition, the scheme is required to satisfy an online-time provenance-binding property as defined in Definition 2.*

The size requirements of the provenance string space $\mathcal{I}$ ensure that the fresh provenance string is unpredictable during preprocessing of the adversary. The provenance string ρ binds the proof artifact to a provenance context without changing the underlying circuit.

Completeness. For every λ, every circuit $\mathcal{C}$, every $(x, w) \in \mathcal{R}_\mathcal{C}$, and every $\rho \in \mathcal{I}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, x, \rho, \pi_\rho) = 1 \;\middle|\; \begin{array}{l} (\mathsf{pk}, \mathsf{vk}, \mathcal{I}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{C}), \\ \pi_\rho \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w, \rho) \end{array}\right] = 1.$$

Knowledge soundness. We say PB satisfies knowledge soundness if for every PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}^\mathcal{A}$ such that

$$\Pr\left[\begin{array}{c} \mathsf{Verify}(\mathsf{vk}, x, \rho, \pi) = 1 \\ \wedge (x, w^\star) \notin \mathcal{R}_\mathcal{C} \end{array} \;\middle|\; \begin{array}{l} (\mathsf{pk}, \mathsf{vk}, \mathcal{I}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{C}), \\ (x, \pi, \rho) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{vk}, \mathcal{I}), \\ w^\star \leftarrow \mathcal{E}^\mathcal{A}(x, \pi, \rho) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Equivalently, any adversary that outputs an accepting proof for (x, ρ) must know a witness $w^\star$ such that $(x, w^\star) \in \mathcal{R}_\mathcal{C}$.

Succinctness. As usual for SNARKs, the proof size and verifier running time should be sublinear in the size of the computation. Concretely, we require $|\pi_\rho|$ and the running time of $\mathsf{Verify}(\mathsf{vk}, x, \rho, \pi_\rho)$ to be $\mathsf{poly}(\lambda, |x|, \log |\mathcal{C}|)$, or to satisfy the corresponding succinctness bound of the underlying SNARK model.

Zero knowledge (optional). When zero knowledge is required, we additionally require the standard zero-knowledge property for the identity-indexed proving algorithm. That is,

for every valid $(x, w) \in \mathcal{R}_{\mathcal{C}}$ and every $\rho \in \mathcal{I}$, the proof $\pi_\rho \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w, \rho)$ can be simulated from (vk, x, ρ) without knowing w. We note that zero knowledge is orthogonal to provenance binding.

3.1. Online-Time Provenance Binding

We formalize the provenance binding property by having a lower bound on the online time an adversary has to perform after ρ-independent preprocessing. After preprocessing, the adversary receives a fresh provenance information string ρ and must still spend substantial online time to produce a valid proof.

Single-core online time. Fix a deterministic single-core reference machine M, or equivalently a fixed RAM model with specified costs for field operations, group operations, hashing, memory reads, and memory writes. For an algorithm $\mathcal{A}$, let $\mathsf{cycles}_{\mathsf{M}}(\mathcal{A}; \mathsf{input})$ denote its running time on M, measured in single-core processor cycles. All online-time statements below are with respect to this fixed machine model.

For an honest prover, we define the honest time to generate the proof as $t_{\mathsf{hon}}(\lambda, \mathcal{C}, x, w, \rho) := \mathbb{E}\left[\mathsf{cycles}_{\mathsf{M}}\big(\mathsf{Prove}(\mathsf{pk}, x, w, \rho)\big) \,\middle|\, (\mathsf{pk}, \mathsf{vk}, \mathcal{I}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{C})\right]$, where the expectation is over setup randomness and the randomness during of Prove. When the parameters are clear, we write simply t.

We introduce the provenance-binding game, where an adversary tries to perform preprocessing and then tries to finish the proof in lower time than the honest time t_{hon}.

Provenance-Binding Game $\mathsf{ProvGame}^{\mathsf{PB}}_{\mathcal{A}_0,\mathcal{A}_1}(\lambda, \mathcal{C}, x, w)$

1) The challenger runs $(\mathsf{pk}, \mathsf{vk}, \mathcal{I}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{C})$.
2) Before seeing the provenance string, the preprocessing adversary outputs
$$L \leftarrow \mathcal{A}_0(\mathsf{pk}, \mathsf{vk}, \mathcal{I}, 1^\lambda, \mathcal{C}, x, w).$$
The string L represents arbitrary provenance-independent preprocessing advice.
3) The challenger samples a fresh provenance string $\rho \xleftarrow{\$} \mathcal{I}$ and sends ρ to $\mathcal{A}_1$.
4) The online adversary outputs
$$\pi \leftarrow \mathcal{A}_1(\mathsf{pk}, \mathsf{vk}, \mathcal{I}, x, \rho, L).$$
5) The adversary wins if $\mathsf{Verify}(\mathsf{vk}, x, \rho, \pi) = 1$.

Definition 2 (σ-provenance binding)**.** *Let $\sigma : \mathbb{N} \to \mathbb{N}$ be a function bounding the online time of the adversary. We say* PB *is* σ-provenance-binding *if, for every PPT preprocessing adversary $\mathcal{A}_0$ running in time $\mathsf{poly}(|\mathcal{C}|, \lambda)$ and every online adversary $\mathcal{A}_1$ running in at most*

$$\sigma\left(t_{\mathsf{hon}}(\lambda, \mathcal{C}, x, w, \rho)\right)$$

single-core cycles after receiving ρ, we have

$$\Pr\left[\mathsf{ProvGame}^{\mathsf{PB}}_{\mathcal{A}_0,\mathcal{A}_1}(\lambda, \mathcal{C}, x, w) = 1\right] \le \mathsf{negl}(\lambda),$$

for every valid $(x, w) \in \mathcal{R}_{\mathcal{C}}$. The probability is over setup randomness, the sampled provenance string, and the coins of the adversaries.

In this definition, the adversary may preprocess $(\mathsf{pk}, \mathsf{vk}, \mathcal{I}, \mathcal{C}, x, w)$ before seeing ρ. This models a prover that already knows the witness, has generated the same proof before, or has done substantial preprocessing before the fresh provenance information is sampled. Provenance binding says that such preprocessing should not allow the adversary to produce a valid proof for the fresh provenance more quickly than $\sigma(t_{\mathsf{hon}})$.

Typical parameterization. The main case is $\sigma(t) = c \cdot t$ for a constant $c \in (0, 1)$. Then ct-provenance binding means that, even after arbitrary provenance-independent preprocessing, producing a valid proof for a fresh provenance string still requires a constant fraction of the honest online proving time. Our attack in Section 4 breaks this typical parameterization by having $\sigma(t) = o(t)$.

4. Preprocessing Attacks on Provenance Binding in KZG-PLONK

This section gives preprocessing attacks on KZG-PLONK when the provenance information ρ is included only through Fiat-Shamir challenges. In our attack, the prover work that depends on the Fiat-Shamir challenge can be represented as the evaluation of fixed field- or group-valued polynomials whose coefficients are independent of ρ. These polynomials can be preprocessed before ρ is sampled, so that after ρ is given, they are computed faster online.

We make use of the preprocessing technique from Section 2.2. We use the single-core online-time model from Section 3.1. Let $\ell_p := \lceil \log_2 p \rceil$ be the number of bits needed to represent a field element. Then one field or group operation costs $\Omega(\ell_p)$.

The section is organized as follows. In Section 4.1, we give the preprocessing attack for KZG opening at a random point, and explain how it applies to the PLONK gate check in Section 4.2. Then in Section 4.3, we give the preprocessing attack for the PLONK permutation check.

4.1. KZG with a Random Opening Point

We first consider a single KZG opening, where the opening point is determined by ρ. Let $f(X) = \sum_{i=0}^{d} f_i X^i$ and use the additive notation from Section 2.1, so $g_i = [\tau^i]_1$ and $\mathsf{cm}_f = \sum_{i=0}^{d} f_i g_i$. For an opening challenge α, define $Q(X, Y) := (f(X) - f(Y))/(X - Y)$. The KZG proof is $\pi_\alpha = [Q(\tau, \alpha)]_1$.

Write $q_\alpha(X) = Q(X, \alpha) = \sum_{i=0}^{d-1} c_i(\alpha) X^i$, where $c_i(x) := \sum_{j=0}^{d-1-i} f_{i+j+1} x^j$. Then $\pi_\alpha = \sum_{i=0}^{d-1} c_i(\alpha) g_i$. Thus, once f and the SRS are fixed, all opening proofs form a fixed-base group-valued function of α.

Batching. Fix a block size b. Let $N_{\mathsf{blk}} := d/b$ and set $c_d(x) := 0$. For block $h \in \{0, \ldots, N_{\mathsf{blk}} - 1\}$ and $i \in \{0, \ldots, b-1\}$, we define the polynomial

$$c_{hb+i}(x) = \sum_{j=0}^{b-1-i} f_{hb+i+j+1} x^j + x^{b-i} c_{(h+1)b}(x).$$

In particular, $c_{hb}(x) = \bar{c}_h(x) + x^b c_{(h+1)b}(x)$, where $\bar{c}_h(x) := \sum_{i=0}^{b-1} f_{hb+i+1} x^i$.

Define the polynomial $G_h(x) := \sum_{i=0}^{b-1} c_{hb+i}(x) g_{hb+i}$. The identity above gives $G_h(x) = A_h(x) + c_{(h+1)b}(x) \cdot U_h(x)$, where

$$\begin{cases} A_h(x) := \sum_{j=0}^{b-1} x^j A_{h,j} \\ A_{h,j} := \sum_{i=0}^{b-1-j} f_{hb+i+j+1} g_{hb+i} \\ U_h(x) := \sum_{i=0}^{b-1} x^{b-i} g_{hb+i} \end{cases}$$

Preprocessing and online opening. Before α is known, preprocess evaluation tables for the polynomial $\bar{c}_h$ and the group-valued polynomials A_h, U_h, for every block h. After α is known, compute $1, \alpha, \ldots, \alpha^b$, initialize $z \leftarrow 0$ and $G \leftarrow 0_{\mathbb{G}_1}$, and process blocks from high to low. At the beginning of block h, the invariant is $z = c_{(h+1)b}(\alpha)$.

The prover then evaluates $\bar{c}_h(\alpha)$, $A_h(\alpha)$, and $U_h(\alpha)$ from their evaluation tables, updates $G \leftarrow G + A_h(\alpha) + z \cdot U_h(\alpha)$, and update $z \leftarrow \bar{c}_h(\alpha) + \alpha^b z$. After the last block, $z = c_0(\alpha)$, so the claimed value is $\beta = f_0 + \alpha z$ and the proof is G.

Correctness. The invariant $z = c_{(h+1)b}(\alpha)$ holds initially because $c_d = 0$. If it holds before block h, then the group update adds $G_h(\alpha) = \sum_{i=0}^{b-1} c_{hb+i}(\alpha) g_{hb+i}$, and set $z = c_{hb}(\alpha)$. After all blocks, $G = \sum_{i=0}^{d-1} c_i(\alpha) g_i = [q_\alpha(\tau)]_1$, and $\beta = f(\alpha)$. Thus the algorithm outputs the standard KZG opening.

Cost. Each block polynomial has $O(b)$ monomials. According to Section 2.2, the preprocessing costs $O((d/b)2^b)$, and the online work is $O((d/b)\ell_p)$. Taking $b = C \log d$ for a constant $C > 0$ gives polynomial-size preprocessing $O(d^{C+1}/(C \log d))$ and online work $O(d\ell_p/(C \log d))$.

4.2. PLONK Gate Check

Now we recall the PLONK protocol [13] consisting of witness commitments, gate check, and permutation check. Let $H = \{1, \omega, \ldots, \omega^{n-1}\}$. We encode the left input, right input, and output of the i-th gate as consecutive values of one trace polynomial f: $f(\omega^{3i})$ is the left input, $f(\omega^{3i+1})$ is the right input, and $f(\omega^{3i+2})$ is the output. Let $H_{\mathsf{gate}} := \{\omega^{3i} : i = 0, \ldots, n/3 - 1\}$. Let S_{add} and S_{mul} be fixed selector polynomials such that, on H_{gate}, S_{add} selects addition gates and S_{mul} selects multiplication gates.

For every $x \in H_{\mathsf{gate}}$, the gate constraint is

$$S_{\mathsf{add}}(x)\big(f(\omega x) + f(x)\big) + S_{\mathsf{mul}}(x)\big(f(\omega x) \cdot f(x)\big) = f(\omega^2 x).$$

The prover commits to the trace polynomial f using KZG. The selector polynomials are fixed by the circuit, and their commitments are part of the proving and verification keys.

To prove the gate constraint, the protocol samples a random point ζ. The prover opens f at $\zeta, \omega\zeta, \omega^2\zeta$ and opens the selector polynomials $S_{\mathsf{add}}, S_{\mathsf{mul}}$ at ζ if their values are not already explicit in the verification key. The verifier checks

$$S_{\mathsf{add}}(\zeta)\big(f(\omega\zeta) + f(\zeta)\big) + S_{\mathsf{mul}}(\zeta)\big(f(\omega\zeta) \cdot f(\zeta)\big) \stackrel{?}{=} f(\omega^2\zeta).$$

Preprocessing attack. All polynomials opened in the gate check are fixed before ρ is sampled. Thus the preprocessing adversary preprocesses the KZG opening functions for f, S_{add}, and S_{mul} using the method of Section 4.1. After ζ is known, the online adversary computes the ordinary KZG openings at $\zeta, \omega\zeta, \omega^2\zeta$ using the preprocessed tables and outputs the standard openings expected by the verifier.

Cost. The gate check requires only a constant number of KZG openings of fixed polynomials. By Section 4.1, each such opening has polynomial-size preprocessing and online work $O(n\ell_p/(C \log n))$ for group size $b = C \log n$. This is asymptotically smaller than the permutation-check opening cost $O(n\ell_p/(\log n)^{1/3})$, so the gate check does not change the final online-work bound.

4.3. PLONK Permutation Check

We now apply the same idea to the PLONK permutation check. Recall $H = \{r_0, \ldots, r_{n-1}\} = \{1, \omega, \ldots, \omega^{n-1}\}$ is the PLONK evaluation domain, where we write $r_i = \omega^i$. For Lagrange-basis notation, let $\ell_i(X)$ be the Lagrange polynomial for the point r_i, namely $\ell_i(X) = \prod_{j \neq i}(X - r_j)/(r_i - r_j)$. We set $G_i := [\ell_i(\tau)]_1$. These G_i are derived from the powers-of-τ.

Permutation check recap. The permutation check enforces that the witness values are consistent with the circuit wiring. Let t_i be the witness value at row i, and let P_i be the wiring label for the position to which row i is matched. The verifier samples random permutation challenges $\alpha, \beta \in \mathbb{F}_p$. The prover then compares the two compressed multisets $\{t_i + \alpha P_i + \beta\}_{i=0}^{n-1}$ and $\{t_i + \alpha r_i + \beta\}_{i=0}^{n-1}$ If the wiring is satisfied, these multisets are equal, and therefore the product of the ratios is 1. Randomness of α, β makes an invalid wiring pass this product test only with probability $O(n/p)$.

Define $A_i := t_i + \alpha P_i + \beta$ and $B_i := t_i + \alpha r_i + \beta$. Also define $c_0 := 1$, $c_{i+1} := c_i \cdot \frac{A_i}{B_i}$ for $i = 0, \ldots, n-1$. Therefore, the permutation constraint holds if and only if $c_{i+1} B_i = c_i A_i$, $c_0 = 1$ and $c_n = 1$. Let $C_{\mathsf{perm}}(X)$ be the degree-$< n$ polynomial with $C_{\mathsf{perm}}(r_i) = c_i$ for $i = 0, \ldots, n-1$. Then the permutation check needs to commit to this polynomial $C_{\mathsf{perm}}(X)$ and open it at random points.

4.3.1. Preprocessed Commitment to the Permutation-Check Polynomial. Choose a commitment group size b_{com}. Let $N_{\mathsf{com}} := n/b_{\mathsf{com}}$. For the h-th group, write $L := hb_{\mathsf{com}}$ and $z_h := c_L$. Define $R_{h,0} := 1$ and $R_{h,i} := \prod_{j=0}^{i-1} A_{L+j}/B_{L+j}$ for $i \geq 1$, so $c_{L+i} = z_h R_{h,i}$.

Define $N_h(\alpha,\beta) := \prod_{j=0}^{b_{\mathsf{com}}-1} A_{L+j}$ and $D_h(\alpha,\beta) := \prod_{j=0}^{b_{\mathsf{com}}-1} B_{L+j}$. The boundary update is $z_{h+1} = z_h N_h/D_h$. Define the numerator polynomial

$$E_h(\alpha,\beta) := \sum_{i=0}^{b_{\mathsf{com}}-1} \left(\prod_{j=0}^{i-1} A_{L+j}\right)\left(\prod_{j=i}^{b_{\mathsf{com}}-1} B_{L+j}\right) G_{L+i}.$$

Then $D_h^{-1}E_h = \sum_{i=0}^{b_{\mathsf{com}}-1} R_{h,i}G_{L+i}$, and the prover adds $z_h D_h^{-1}E_h$ to the commitment.

Preprocessing and online commitment. Before (α,β) is known, expand N_h, D_h, E_h for every group h. Since every A_i and B_i is affine in (α,β), these polynomials have total degree at most b_{com} and $O(b_{\mathsf{com}}^2)$ monomials. Preprocess evaluation tables for the polynomials N_h, D_h and the group-valued polynomial E_h.

Online, after (α,β) is known, initialize $z \leftarrow 1$ and $\mathsf{cm}_{C_{\mathsf{perm}}} \leftarrow 0_{\mathbb{G}_1}$. For each group h, evaluate N_h, D_h, E_h using the preprocessed tables, update $\mathsf{cm}_{C_{\mathsf{perm}}} \leftarrow \mathsf{cm}_{C_{\mathsf{perm}}} + zD_h^{-1}E_h$, and update $z \leftarrow zN_hD_h^{-1}$. The output is $\mathsf{cm}_{C_{\mathsf{perm}}}$.

Correctness. At the start of group h, $z = c_L$. The update adds $c_L\sum_{i=0}^{b_{\mathsf{com}}-1} R_{h,i}G_{L+i} = \sum_{i=0}^{b_{\mathsf{com}}-1} c_{L+i}G_{L+i}$. The boundary update sets $z = c_{L+b_{\mathsf{com}}}$. Summing over groups gives $\mathsf{cm}_{C_{\mathsf{perm}}} = \sum_{i=0}^{n-1} c_iG_i = [C_{\mathsf{perm}}(\tau)]_1$.

Cost. By the cost template in Section 2.2, preprocessing costs $O((n/b_{\mathsf{com}})2^{O(b_{\mathsf{com}}^2)})$ field and group operations, while online work costs $O((n/b_{\mathsf{com}})\ell_p)$. Taking $b_{\mathsf{com}} = \Theta(\sqrt{\log n})$ gives polynomial-size preprocessing and online commitment work $O(n\ell_p/\sqrt{\log n})$.

4.3.2. Preprocessed Opening of the Permutation-Check Polynomial. Let $\gamma \notin H$ be the KZG opening point and write $\Delta_i(\gamma) := r_i - \gamma$. The claimed value is $y := C_{\mathsf{perm}}(\gamma)$. In the Lagrange basis, the KZG witness is $\pi_\gamma = \sum_{i=0}^{n-1} \frac{c_i - y}{r_i - \gamma} G_i$. Define $S_c := \sum_i c_i\Delta_i(\gamma)^{-1}G_i$ and $S_0 := \sum_i \Delta_i(\gamma)^{-1}G_i$, so $\pi_\gamma = S_c - yS_0$. For the root-of-unity Lagrange basis, $C_{\mathsf{perm}}(\gamma) = -(\gamma^n-1)S_y/n$, where $S_y := \sum_i c_i r_i \Delta_i(\gamma)^{-1}$. Thus the same grouped computation produces both y and π_γ. Choose an opening group size b_{open}. Let $N_{\mathsf{open}} := n/b_{\mathsf{open}}$. For group h, set $L := hb_{\mathsf{open}}$, $z_h := c_L$, and define $R_{h,i}$ as above. Let $N_h := \prod_{j=0}^{b_{\mathsf{open}}-1} A_{L+j}$, $D_h := \prod_{j=0}^{b_{\mathsf{open}}-1} B_{L+j}$, and $Z_h(\gamma) := \prod_{i=0}^{b_{\mathsf{open}}-1} \Delta_{L+i}(\gamma)$.

For $i \in \{0,\dots,b_{\mathsf{open}}-1\}$, define

$$M_{h,i} := \left(\prod_{j=0}^{i-1} A_{L+j}\right)\left(\prod_{j=i}^{b_{\mathsf{open}}-1} B_{L+j}\right)\left(\prod_{\ell\neq i} \Delta_{L+\ell}(\gamma)\right).$$

Then $M_{h,i}/(D_hZ_h) = R_{h,i}/\Delta_{L+i}(\gamma)$. Define $Y_h := \sum_i r_{L+i}M_{h,i}$, $\mathcal{P}_h := \sum_i M_{h,i}G_{L+i}$, and $O_h := \sum_i(\prod_{\ell\neq i}\Delta_{L+\ell}(\gamma))G_{L+i}$. The prover computes the intermediate values, and adds $z_hY_h/(D_hZ_h)$ to S_y, $z_h\mathcal{P}_h/(D_hZ_h)$ to S_c, and O_h/Z_h to S_0.

Preprocessing and online opening. Before (α,β,γ) is known, expand the polynomials N_h, D_h, Z_h, Y_h and the group-valued polynomials $\mathcal{P}_h, O_h$. The largest monomial set has size $O(b_{\mathsf{open}}^3)$, because the relevant monomials have the form $\alpha^u\beta^v\gamma^w$ with $u+v \le b_{\mathsf{open}}$ and $0 \le w \le b_{\mathsf{open}}-1$. Preprocess evaluation tables for all these polynomials.

Online, initialize $z \leftarrow 1$, $S_y \leftarrow 0$, $S_c \leftarrow 0_{\mathbb{G}_1}$, and $S_0 \leftarrow 0_{\mathbb{G}_1}$. For each group h, evaluate $N_h, D_h, Z_h, Y_h, \mathcal{P}_h, O_h$ using evaluation tables, update $S_y \leftarrow S_y + zY_h/(D_hZ_h)$, $S_c \leftarrow S_c + z\mathcal{P}_h/(D_hZ_h)$, $S_0 \leftarrow S_0 + O_h/Z_h$, and update $z \leftarrow zN_h/D_h$. Finally output $y := -(\gamma^n-1)S_y/n$ and $\pi_\gamma := S_c - yS_0$.

Correctness. The identity $M_{h,i}/(D_hZ_h) = R_{h,i}/\Delta_{L+i}(\gamma)$ and the relation $c_{L+i} = z_hR_{h,i}$ imply that the group updates add exactly $\sum_i c_{L+i}r_{L+i}/\Delta_{L+i}(\gamma)$ to S_y, $\sum_i c_{L+i}G_{L+i}/\Delta_{L+i}(\gamma)$ to S_c, and $\sum_i G_{L+i}/\Delta_{L+i}(\gamma)$ to S_0. Summing over all groups gives the definitions of S_y, S_c, S_0 above. Therefore $y = C_{\mathsf{perm}}(\gamma)$ and $\pi_\gamma = \sum_i(c_i-y)(r_i-\gamma)^{-1}G_i = [(C_{\mathsf{perm}}(\tau)-C_{\mathsf{perm}}(\gamma))/(\tau-\gamma)]_1$. The verifier uses the ordinary KZG verification equation.

Cost. By Section 2.2, preprocessing costs $O((n/b_{\mathsf{open}})2^{O(b_{\mathsf{open}}^3)})$ field and group operations, while online work costs $O((n/b_{\mathsf{open}})\ell_p)$. Taking $b_{\mathsf{open}} = \Theta((\log n)^{1/3})$ gives polynomial-size preprocessing and online opening work $O(n\ell_p/(\log n)^{1/3})$.

4.4. Non-Provenance-Binding Theorem

Putting the results in Sections 4.2 and 4.3 together, we get an algorithm that, after preprocessing, is able to generate a KZG-PLONK proof for any circuit of size n in $o(n)$ time. We formalize this result with the following theorem.

Theorem 1. *Consider the KZG-PLONK consisting of witness commitments, the gate check in Section 4.2, and the permutation check in Section 4.3, where the provenance string ρ is used only to derive the Fiat-Shamir challenges. Then there exist PPT adversaries $(\mathcal{A}_0,\mathcal{A}_1)$ in $\mathsf{ProvGame}^{\mathsf{PB}}$ such that $\mathcal{A}_0$ performs provenance-independent polynomial-time preprocessing, $\mathcal{A}_1$ wins with probability $1-\mathsf{negl}(\lambda)$, and its online time is $O\Big(\frac{n\ell_p}{(\log n)^{1/3}}\Big) = o(t_{\mathsf{hon}}(n))$. Consequently, the KZG-PLONK protocol is not $c\cdot t$-provenance-binding for any fixed constant $c \in (0,1)$ in the sense of Definition 2.*

Proof. We describe the adversary pair. The preprocessing adversary $\mathcal{A}_0$ receives $(\mathsf{pk},\mathsf{vk},\mathcal{I},\mathcal{C},x,w)$ before the provenance string ρ is sampled. It fixes the prover randomness and computes the trace polynomial f, the selector polynomials $S_{\mathsf{add}}, S_{\mathsf{mul}}$, the permutation data, the KZG bases, and all provenance-independent coefficients used by the gate and permutation checks.

For the gate check, $\mathcal{A}_0$ preprocesses the KZG openings of the fixed trace polynomial f and the fixed selector polynomials $S_{\mathsf{add}}, S_{\mathsf{mul}}$. After the Fiat-Shamir transcript determines the gate-check point ζ, $\mathcal{A}_1$ opens f at $\zeta, \omega\zeta, \omega^2\zeta$ and opens $S_{\mathsf{add}}, S_{\mathsf{mul}}$ at ζ. It then supplies the ordinary KZG openings required for the verifier to check

$$S_{\mathsf{add}}(\zeta)\big(f(\omega\zeta)+f(\zeta)\big) + S_{\mathsf{mul}}(\zeta)\big(f(\omega\zeta)\cdot f(\zeta)\big) = f(\omega^2\zeta).$$

These openings are standard KZG openings, and by correctness of the preprocessing algorithm they are identical to

the openings an honest prover would produce for the same Fiat-Shamir challenge.

For the permutation-check commitment, $\mathcal{A}_0$ builds evaluation tables for N_h, D_h, E_h in each commitment group. For the permutation-check opening, it builds evaluation tables for $N_h, D_h, Z_h, Y_h, \mathcal{P}_h, O_h$ in each opening group. Choose $b_{\mathsf{com}} = \Theta(\sqrt{\log n})$ and $b_{\mathsf{open}} = \Theta((\log n)^{1/3})$. The preprocessing sizes are polynomial in n, so $\mathcal{A}_0$ is a valid polynomial-time preprocessing adversary.

After the challenger samples ρ, the online adversary $\mathcal{A}_1$ computes the Fiat-Shamir challenges determined by the transcript and ρ. It outputs the precomputed witness commitment to f. It answers the gate check as described above. It computes the permutation-check commitment using the preprocessed bivariate grouped polynomials and computes the opening of C_{perm} using the preprocessed trivariate grouped polynomials. It then outputs the ordinary KZG commitments, claimed values, and KZG opening proofs required by the core verifier.

Correctness of the KZG-opening preprocessing and the permutation-check algorithms shows that every object output by $\mathcal{A}_1$ is exactly the object that an honest prover with the same witness and prover randomness would output for the same Fiat-Shamir challenges. Therefore the verifier accepts the witness commitment, the gate check, the permutation-check commitment, and the permutation-check opening, except with the usual negligible probability that a denominator equals zero. Thus $(\mathcal{A}_0, \mathcal{A}_1)$ wins with probability $1 - \mathsf{negl}(\lambda)$.

It remains to compare online time. The honest prover, after the provenance-dependent permutation challenges are known, computes the permutation-check recurrence $c_{i+1} = c_i A_i / B_i$ over all n rows. This alone requires reading row-wise data and performing a challenge-dependent recurrence, costing $\Omega(n\ell_p)$ online work. The honest prover also performs KZG openings for the gate and permutation checks, which only increases $t_{\mathsf{hon}}(n)$.

The preprocessed gate-check openings cost $O(n\ell_p/(C \log n))$ online work by Section 4.1. The preprocessed permutation-check commitment costs $O(n\ell_p/\sqrt{\log n})$. The preprocessed permutation-check opening costs $O(n\ell_p/(\log n)^{1/3})$, which dominates. Hence

$$T_{\mathsf{pre}}(n) = O\left(\frac{n\ell_p}{(\log n)^{1/3}}\right).$$

Since $t_{\mathsf{hon}}(n) = \Omega(n\ell_p)$, we have $T_{\mathsf{pre}}(n)/t_{\mathsf{hon}}(n) = O(1/(\log n)^{1/3}) = o(1)$. Therefore, for any fixed $c \in (0,1)$ and all sufficiently large n, $\mathcal{A}_1$ runs in online time less than $c \cdot t_{\mathsf{hon}}(n)$ while winning with probability $1 - \mathsf{negl}(\lambda)$. □

4.5. Extension to SumCheck and Multilinear KZG

The preceding theorem focuses on KZG-PLONK, which uses univariate polynomials. The similar preprocessing idea also applies to multilinear PCS and Poly-IOPs, such as SumCheck [19] and multilinear KZG [22].

Let $B^\mu := \{0,1\}^\mu$ and let $T := 2^\mu$ be the multilinear table size. For $a \in B^\mu$, write $\chi_a(X) := \prod_{j=1}^{\mu}(a_j X_j + (1 - a_j)(1 - X_j))$. For a function $f : B^\mu \to \mathbb{F}_p$, its multilinear extension is $\widetilde{f}(X) = \sum_{a \in B^\mu} f(a)\chi_a(X)$.

Preprocessing SumCheck messages. Consider a fixed multilinear SumCheck target $F(X_1, \ldots, X_\mu)$. In round i, after challenges $\alpha_1, \ldots, \alpha_{i-1}$ are fixed, the prover sends $r_i(Z) := \sum_{w \in B^{\mu-i}} F(\alpha_1, \ldots, \alpha_{i-1}, Z, w)$. Since F is multilinear, $r_i(Z)$ has degree at most 1.

For $a \in B^{i-1}$, define $R_{i,a,0}, R_{i,a,1} \in \mathbb{F}_p$ by $\sum_{w \in B^{\mu-i}} F(a_1, \ldots, a_{i-1}, Z, w) = R_{i,a,0} + R_{i,a,1} Z$. Let $L_a(X_1, \ldots, X_{i-1})$ be the Boolean tensor-product Lagrange basis polynomial for a, namely $L_a(X_1, \ldots, X_{i-1}) = \prod_{j=1}^{i-1}(a_j X_j + (1 - a_j)(1 - X_j))$. Thus $L_a(b) = 1$ if $b = a$ and $L_a(b) = 0$ otherwise.

The coefficient of Z^k in $r_i(Z)$ is $\theta_{i,k}(\alpha_1, \ldots, \alpha_{i-1}) := \sum_{a \in B^{i-1}} R_{i,a,k} L_a(\alpha_1, \ldots, \alpha_{i-1})$ for $k \in \{0,1\}$. Therefore $r_i(Z) = \theta_{i,0}(\alpha_1, \ldots, \alpha_{i-1}) + \theta_{i,1}(\alpha_1, \ldots, \alpha_{i-1}) Z$. Each SumCheck message coefficient is a fixed multilinear polynomial in the previous verifier challenges.

The number of coefficient entries is $N_{\mathrm{coef}} := \sum_{i=1}^{\mu} 2 \cdot 2^{i-1} = 2(T-1) = O(T)$. We preprocess these entries directly in the Boolean Lagrange basis, rather than expanding the coefficient functions as monomials in $\alpha_1, \ldots, \alpha_{i-1}$. This is important because the number of variables can be $\Theta(\log T)$.

Choose an integer $s \geq 0$ and set $b := 2^s$. For round i, let $s_i := \min\{s, i-1\}$ and write a prefix index $a \in B^{i-1}$ as $a = (u, v)$, where $u \in B^{i-1-s_i}$ and $v \in B^{s_i}$. For every triple (i, k, u) with $k \in \{0,1\}$, define the coefficient list $\big(R_{i,(u,v),k}\big)_{v \in B^{s_i}}$. The preprocessing algorithm stores all evaluations of this list: $\mathsf{Tab}_{i,k,u}[\eta] := \sum_{v \in B^{s_i}} \eta_v R_{i,(u,v),k}$, for $\eta \in \{0,1\}^{2^{s_i}}$.

For early rounds with $i - 1 < s$, this is simply the table for the whole coefficient list. For later rounds, each table has $2^{2^s} = 2^b$ entries. The total number of tables over all rounds and both coefficients is $O(T/b)$, so the preprocessing size is $O((T/b) \cdot 2^b)$ field elements.

At online time in round i, after $\alpha_1, \ldots, \alpha_{i-1}$ are known, split the challenge vector as $\alpha^{\mathsf{pre}} := (\alpha_1, \ldots, \alpha_{i-1-s_i})$ and $\alpha^{\mathsf{suf}} := (\alpha_{i-s_i}, \ldots, \alpha_{i-1})$. For each $v \in B^{s_i}$, compute the weight of the suffix

$$\lambda_v := L_v(\alpha^{\mathsf{suf}}) = \textstyle\prod_{j=1}^{s_i}\big(v_j \alpha_{i-s_i+j-1} + (1 - v_j)(1 - \alpha_{i-s_i+j-1})\big).$$

These $2^{s_i} \leq b$ weights are shared across all prefix indices u in round i.

Write each λ_v in binary as $\lambda_v = \sum_{\ell=0}^{\ell_p - 1} \lambda_{v,\ell} 2^\ell$, where $\lambda_{v,\ell} \in \{0,1\}$. For each bit position ℓ, define the mask $\eta_\ell := (\lambda_{v,\ell})_{v \in B^{s_i}}$. Then for every (i, k, u), the suffix contribution is recovered from the preprocessed table by

$$W_{i,k,u} := \sum_{v \in B^{s_i}} R_{i,(u,v),k} \lambda_v = \sum_{\ell=0}^{\ell_p - 1} 2^\ell \cdot \mathsf{Tab}_{i,k,u}[\eta_\ell].$$

Finally, compute the prefix Lagrange weights $L_u(\alpha^{\mathsf{pre}})$ and output $\theta_{i,k}(\alpha_1, \ldots, \alpha_{i-1}) = \sum_{u \in B^{i-1-s_i}} L_u(\alpha^{\mathsf{pre}}) \cdot W_{i,k,u}$. Correctness follows from the equality that

$$L_{(u,v)}(\alpha_1, \ldots, \alpha_{i-1}) = L_u(\alpha^{\mathsf{pre}}) \cdot L_v(\alpha^{\mathsf{suf}}).$$

The online cost per table is $O(\ell_p)$, and the number of tables used over all rounds is $O(T/b)$. Therefore, the total online work is $O(T\ell_p/b)$. Choosing $b = C \log T$ for a constant $C > 0$ gives polynomial-size preprocessing and online SumCheck-message work $O(T\ell_p/(C \log T))$.

Multilinear KZG protocol. We next recall the multilinear KZG commitment and opening protocol [22]. Let $g : B^\mu \to \mathbb{F}_p$ be a committed multilinear polynomial. Let $U_a := [\chi_a(\tau)]_1$ be the multilinear KZG basis for $a \in B^\mu$, where $\tau = (\tau_1, \ldots, \tau_\mu)$. The commitment is $\mathsf{Com}(g) = \sum_{a \in B^\mu} g(a) U_a = [\widetilde{g}(\tau)]_1$.

To open at $\alpha = (\alpha_1, \ldots, \alpha_\mu)$, the ordinary prover outputs the value $z := \widetilde{g}(\alpha)$ and proof elements $\pi_1, \ldots, \pi_\mu$ satisfying

$$\widetilde{g}(X) - \widetilde{g}(\alpha) = \sum_{i=1}^{\mu} (X_i - \alpha_i) q_i(X_{i+1}, \ldots, X_\mu).$$

The proof is $\pi_i = [q_i(\tau_{i+1}, \ldots, \tau_\mu)]_1$. The verifier checks

$$e(\mathsf{Com}(g) - [z]_1, g_2) \stackrel{?}{=} \prod_{i=1}^{\mu} e(\pi_i, [\tau_i - \alpha_i]_2).$$

Preprocessing attack for multilinear KZG opening. The claimed value is $z = \widetilde{g}(\alpha) = \sum_{a \in B^\mu} g(a) \chi_a(\alpha)$. An honest prover computes it in $O(T)$ field operations. We preprocess this evaluation with the following protocol.

Choose $s \le \mu$, set $b := 2^s$, and write each Boolean index as $a = (u, v) \in B^{\mu - s} \times B^s$. For every prefix u, store all evaluations of the suffix coefficient list $(g(u, v))_{v \in B^s}$, which are $\mathsf{Tab}^{\mathsf{val}}_u[\eta] := \sum_{v \in B^s} \eta_v g(u, v)$ for $\eta \in \{0, 1\}^b$. The preprocessing size is $O((T/b) \cdot 2^b)$ field elements.

Online, compute the suffix weights $\lambda_v := \chi_v(\alpha^{\mathsf{suf}})$ for all $v \in B^s$. These weights are shared across all prefixes u. For each bit position ℓ, let $\eta_\ell := (\lambda_{v,\ell})_{v \in B^s}$. For every prefix u, recover $W_u := \sum_{v \in B^s} g(u, v) \lambda_v$ by $W_u = \sum_{\ell=0}^{\ell_p - 1} 2^\ell \cdot \mathsf{Tab}^{\mathsf{val}}_u[\eta_\ell]$. Then compute the prefix weights $\rho_u := \chi_u(\alpha^{\mathsf{pre}})$ and output $z = \sum_{u \in B^{\mu - s}} \rho_u W_u$. The online work is $O(b + T\ell_p/b)$. Taking $b = C \log T$ gives polynomial-size preprocessing and online work $O(T\ell_p/(C \log T))$.

We now preprocess the proof vector $(\pi_1, \ldots, \pi_\mu)$. For each i and prefix $a \in B^{i-1}$, define the group coefficient

$$\Pi_{i,a} := \left[\sum_{u \in B^{\mu - i}} (g(a, 1, u) - g(a, 0, u)) \chi_u(\tau_{i+1}, \ldots, \tau_\mu) \right]_1.$$

Then $\pi_i(\alpha) = \sum_{a \in B^{i-1}} \chi_a(\alpha_1, \ldots, \alpha_{i-1}) \Pi_{i,a}$. The total number of group coefficients over all proof entries is $\sum_{i=1}^{\mu} 2^{i-1} = T - 1$.

Choose a suffix size s and set $b := 2^s$. For proof coordinate i, let $s_i := \min\{s, i-1\}$ and write $a = (u, v)$ with $u \in B^{i-1-s_i}$ and $v \in B^{s_i}$. For every pair (i, u), store all evaluations of the suffix group coefficient list $(\Pi_{i,(u,v)})_{v \in B^{s_i}}$, which are $\mathsf{Tab}^{\mathsf{grp}}_{i,u}[\eta] := \sum_{v \in B^{s_i}} \eta_v \Pi_{i,(u,v)}$ for $\eta \in \{0, 1\}^{2^{s_i}}$. The total preprocessing stores $O((T/b) \cdot 2^b)$ group elements.

Online, for each proof coordinate i, compute suffix weights $\lambda_v := \chi_v(\alpha_{i-s_i}, \ldots, \alpha_{i-1})$ and prefix weights $\rho_u := \chi_u(\alpha_1, \ldots, \alpha_{i-1-s_i})$. For each prefix u, form the suffix weights $\sigma_{u,v} := \rho_u \lambda_v$. Write $\sigma_{u,v} = \sum_{\ell=0}^{\ell_p - 1} \sigma_{u,v,\ell} 2^\ell$, and let $\eta_{u,\ell} := (\sigma_{u,v,\ell})_{v \in B^{s_i}}$. Let

$$W_{i,u} := \sum_{v \in B^{s_i}} \rho_u \lambda_v \Pi_{i,(u,v)} = \sum_{\ell=0}^{\ell_p - 1} 2^\ell \cdot \mathsf{Tab}^{\mathsf{grp}}_{i,u}[\eta_{u,\ell}].$$

Finally, the prover outputs $\pi_i = \sum_u W_{i,u}$.

The online work for computing the full vector $\pi_1, \ldots, \pi_\mu$ is $O(T\ell_p/b)$. Taking $b = C \log T$ gives polynomial-size preprocessing and online work $O(T\ell_p/(C \log T))$.

5. Provenance-Binding SNARKs from Code-Based Multilinear PCS

In this section, we give a provenance-binding construction based on code-based multilinear polynomial commitments. Examples of such multilinear PCS include BaseFold and FRI IOPP-based PCS when used to commit to multilinear evaluation oracles [6], [30]. The common structure is that the prover encodes the evaluations of the multilinear polynomial into a vector, commits to the vector using Merkle trees, and later proves evaluation by answering a small number of random local consistency queries.

Our technique that achieves $c \cdot t$ provenance binding on such schemes adds minimal overhead. Before building each Merkle tree, every oracle coordinate is authenticated together with a random value derived from ρ. In this way, the Merkle leaf hashes cannot be computed before ρ is known. On the verifier side, it recomputes the random value only at queried positions, so the verification remains succinct.

In Section 5.1, we first introduce our code-based multilinear PCS construction that enables provenance binding. Then, in Section 5.2, we introduce the SNARK that builds on top of the provenance-binding code-based PCS, and prove that it satisfies Definition 2 with $\sigma(t) = c \cdot t$ for some constant c.

Recap: code-based multilinear PCS. Let $f : \mathbb{F}_p^\mu \to \mathbb{F}_p$ be a multilinear polynomial, and write $T := 2^\mu$. Equivalently, f is specified by its Boolean-hypercube evaluation vector $\boldsymbol{f} := (f(b))_{b \in \{0,1\}^\mu} \in \mathbb{F}_p^T$. A code-based multilinear PCS commits to f by encoding this evaluation vector into a vector (i.e., the "encoding"). We write the encoded vector abstractly as $\boldsymbol{\pi}_L := \mathsf{Enc}(\boldsymbol{f}) \in \mathbb{F}_p^{n_L}$.

In BaseFold, $\boldsymbol{\pi}_L$ is obtained by applying the BaseFold encoding to $\boldsymbol{f}$. In a FRI-style multilinear PCS, $\boldsymbol{\pi}_L$ can be viewed as a codeword or low-degree evaluation oracle derived from the multilinear evaluation vector. The precise encoding differs across protocols. What matters for us is that the committed object is represented by an explicit vector oracle and authenticated by a Merkle root.

The PCS commitment to f is $\mathsf{rt}_L := \mathsf{mt.Commit}(\boldsymbol{\pi}_L)$. To open the commitment at $z \in \mathbb{F}_p^\mu$, the prover claims $y = f(z)$ and runs the ordinary code-based opening or proximity protocol. During this protocol, the prover may compute additional vectors $\boldsymbol{\pi}_{L-1}, \ldots, \boldsymbol{\pi}_0$, where $\boldsymbol{\pi}_i \in \mathbb{F}_p^{n_i}$, and commit to each by a Merkle root $\mathsf{rt}_i := \mathsf{mt.Commit}(\boldsymbol{\pi}_i)$.

The verifier then samples queries. Each query opens a constant number of Merkle leaves from each level and checks the consistency of the leaves, which we write abstractly as $\mathsf{Pred}_\ell(\boldsymbol{\pi}_{i_1}[j_1], \ldots, \boldsymbol{\pi}_{i_s}[j_s]; z, y, \mathsf{chal}) = 1$, where $s = O(1)$ and chal denotes the Fiat-Shamir challenges.

5.1. Code-Based Multilinear PCS with Provenance Binding

Provenance-dependent leaf hashes. For every committed oracle level $i \in \{0, \ldots, L\}$ and coordinate $j \in [n_i]$, define the random value

$$\boldsymbol{r}_i^\rho[j] := \mathsf{Mask}(\mathsf{pp}, x, \rho, i, j) \in \{0,1\}^\lambda.$$

Here Mask is a public hash function modeled as a random oracle in the proof. Note that the verifier can recompute $\boldsymbol{r}_i^\rho[j]$ for every queried entry.

Let $\boldsymbol{\pi}_i$ be the ordinary vector at level i. Instead of adding $\boldsymbol{r}_i^\rho[j]$ to the field element, the prover authenticates the value and the provenance-dependent randomness together. The leaf hash at coordinate j is

$$\boldsymbol{h}_i^\rho[j] := \mathsf{H}_{\mathsf{leaf}}\big(\rho \| i \| j \| \mathsf{Encode}(\boldsymbol{\pi}_i[j]) \| \boldsymbol{r}_i^\rho[j]\big).$$

When the construction is applied to a SNARK with multiple committed polynomials, we additionally include the polynomial label in the leaf hash, as described in Section 5.2.

Internal nodes are computed using the ordinary domain-separated Merkle hash on the ordered child digests. Since every leaf already depends on ρ, all internal nodes and the Merkle root also depend on ρ. We write mt_ρ for the ordinary Merkle commitment scheme instantiated with these provenance-dependent leaf hashes.

Including the coordinate identifier prevents a single leaf-hash computation from being reused at different positions, even if their oracle values are identical. Because $\boldsymbol{\pi}_i[j]$ and $\boldsymbol{r}_i^\rho[j]$ are concatenated inside the hash, an adaptively chosen witness or oracle value cannot manipulate the provenance-dependent randomness. As required in Definition 2, the challenger samples the fresh ρ after preprocessing from a provenance space satisfying $|\mathcal{I}| \geq 2^\lambda$.

Commitment. On input provenance string ρ, the prover first computes the encoded vector $\boldsymbol{\pi}_L := \mathsf{Oracle}_L(f)$. It then computes the provenance-dependent leaf hashes $\boldsymbol{h}_L^\rho[j]$ for all $j \in [n_L]$ and generates the Merkle commitment

$$\mathsf{cm}_f^\rho := \mathsf{rt}_L := \mathsf{mt}_\rho.\mathsf{Commit}(\boldsymbol{\pi}_L).$$

Opening. To open at z with claimed value $y = f(z)$, the prover runs the plain code-based PCS opening protocol, except that every committed oracle uses the provenance-dependent leaf hashes above. Whenever the plain protocol would compute and commit to an oracle $\boldsymbol{\pi}_i$, the prover sends $\mathsf{rt}_i := \mathsf{mt}_\rho.\mathsf{Commit}(\boldsymbol{\pi}_i)$. All Fiat-Shamir challenges are derived with ρ, for example $\mathsf{chal} := \mathsf{H}_{\mathsf{FS}}(\rho \| \mathsf{transcript})$.

Consistency checks. Suppose the ordinary verifier would open $\boldsymbol{\pi}_{i_1}[j_1], \ldots, \boldsymbol{\pi}_{i_s}[j_s]$ and check $\mathsf{Pred}_\ell(\boldsymbol{\pi}_{i_1}[j_1], \ldots, \boldsymbol{\pi}_{i_s}[j_s]; z, y, \mathsf{chal}) = 1$. The verifier receives the ordinary values $\boldsymbol{\pi}_{i_1}[j_1], \ldots, \boldsymbol{\pi}_{i_s}[j_s]$ and their Merkle authentication paths. For every opened value, it recomputes $\boldsymbol{r}_{i_t}^\rho[j_t]$ and the corresponding provenance-dependent leaf hash, and verifies the Merkle path against rt_{i_t}. It then runs the ordinary consistency check on the authenticated values.

Lemma 1. *Fix ρ, the committed oracle vectors, the verifier challenges, and the query indices. The opening proof of the provenance-bound code-based multilinear PCS is accepted by an honest verifier if and only if the corresponding ordinary oracle values pass the consistency checks of the plain code-based multilinear PCS.*

Proof. The construction changes only the Merkle leaf hash used to authenticate each oracle value. After authentication, every consistency check is evaluated on exactly the same values as in the ordinary protocol. □

Verifier protocol. The verifier does not need to compute the provenance-dependent values for the entire oracle, since that would make the verification no longer succinct. The verifier only opens specific locations in each query, so during query verification it computes $\boldsymbol{r}_i^\rho[j] = \mathsf{Mask}(\mathsf{pp}, x, \rho, i, j)$ only for the entries it accesses. If the ordinary verifier opens Q_{cell} oracle cells, then the revised verifier performs Q_{cell} additional evaluations of Mask. The proof contains the same number of Merkle roots, opened field elements, and authentication paths as the ordinary proof.

In typical code-based PCS, if the verifier uses $q_{\mathsf{qry}} = \Theta(\lambda)$ query repetitions and there are $L = O(\mu)$ levels of encoded vectors, then $Q_{\mathsf{cell}} = O(q_{\mathsf{qry}} \cdot L)$. For $q_{\mathsf{qry}} = \Theta(\lambda)$ and $L = O(\log n_L)$, the additional verifier work is $O(\lambda \log n_L)$ hash operations, which is $o(n_L)$.

Costs. Let $N_{\mathsf{code}} := \sum_{i=0}^{L} n_i$ be the total number of committed vector entries. Write T_0 for the time needed to compute the ordinary vectors $\boldsymbol{\pi}_L, \ldots, \boldsymbol{\pi}_0$ in the plain PCS. Write T_{mask} for the time needed, after ρ is known, to compute the provenance-dependent values, compute the leaf hashes, and commit to the vectors using Merkle trees.

The construction adds one evaluation of Mask per committed entry and computes Merkle commitments with provenance-dependent leaf hashes. The Merkle-tree layout and the number of internal-node hashes are unchanged. Therefore, in the fixed machine model, we have $T_{\mathsf{mask}} = \Theta(N_{\mathsf{code}})$ hash computations, up to implementation-dependent constant factors.

Note that we do not require the ordinary vector computation T_0 to be provenance-dependent. An adversary may be able to precompute the ordinary oracle vectors before ρ is sampled. Our construction does not prevent this. It instead enforces that the provenance-dependent leaf hashes and their Merkle roots cannot be computed before ρ is known.

Consequently, the theorem below directly provides a lower bound on T_{mask}. If the honest online prover time is $t_{\mathsf{hon}} = T_0 + T_{\mathsf{mask}} + T_{\mathsf{other}}$, then the construction gives a full $c \cdot t_{\mathsf{hon}}$ provenance-binding guarantee whenever $T_{\mathsf{mask}} \geq c \cdot t_{\mathsf{hon}}$ for some constant $c > 0$. As shown in Section 6.2, the Merkle-tree layer is a substantial part of the honest prover

time in our implementation. The measured fraction is an honest-prover runtime breakdown and does not by itself establish the optimal adversarial constant.

After the provenance string ρ is sampled, every committed leaf becomes provenance-dependent. The ordinary oracle $\boldsymbol{\pi}_i$ may be completely precomputable, but the leaf hash $\boldsymbol{h}_i^{\rho}[j]$ cannot be computed before ρ is known. Therefore, a Merkle tree for the oracle cannot be precomputed during the preprocessing phase for the fresh ρ. A Merkle root computed without ρ, or for a different provenance string, does not help in the online proof generation.

After ρ is revealed, an online adversary may try to save work by preparing openings for only a subset of coordinates rather than building the full provenance-dependent Merkle tree. Since an honest verifier makes random queries, this shortcut succeeds only if the prepared set covers almost all queried positions, as we show next.

Coverage of Random Merkle Queries. We use the following coverage property of code-based PCS. In each query repetition, for every committed level i, the verifier opens cells that include an entry index distributed uniformly over $[n_i]$. Equivalently, there is a constant $\kappa > 0$ such that, for every subset $S_i \subseteq [n_i]$, one query repetition touches a coordinate outside S_i with probability at least $\kappa \cdot \left(1 - \frac{|S_i|}{n_i}\right)$. For the PCS queries in FRI and BaseFold, this condition holds because the verifier samples a random entry to query at each level.

Fix a committed level i and a Merkle root rt_i. After the root is fixed and before the entries that the verifier will query are known, let $S_i \subseteq [n_i]$ be the set of coordinates for which the online prover has already made the required provenance-dependent leaf-hash query and determined an authentication path consistent with rt_i.

If the verifier later queries a coordinate outside S_i, then the prover must produce an opening for a leaf whose provenance-dependent hash was not determined when the root was fixed. In the random-oracle model, producing such a path requires predicting an unqueried hash output or finding a hash collision, except with negligible probability.

If $|S_i| < (1-\delta)n_i$, then one query repetition touches a coordinate outside S_i with probability at least $\kappa\delta$. With q_{qry} independent query repetitions, the probability that all repetitions avoid such coordinates is at most $(1-\kappa\delta)^{q_{\mathsf{qry}}}$, up to negligible random-oracle error.

The adversary may also try polynomially many Fiat-Shamir transcripts. If Q_{FS} denotes the number of such attempts, its success probability is at most

$$Q_{\mathsf{FS}}(1-\kappa\delta)^{q_{\mathsf{qry}}} + \mathsf{negl}(\lambda).$$

Taking $q_{\mathsf{qry}} = \Theta(\lambda)$ and $\delta = \log^2(\lambda)/\lambda$ makes this probability negligible. Therefore, any prover that succeeds with non-negligible probability must have $|S_i| = (1-o(1))n_i$ for every committed vector at level i.

Since $N_{\mathsf{code}} := \sum_{i=0}^{L} n_i$, any successful online prover must make provenance-dependent leaf-hash queries for $(1-o(1))N_{\mathsf{code}}$ coordinates. The coordinate identifiers ensure that these are distinct hash inputs even when different positions contain the same field element.

5.2. Provenance-Binding SNARK Construction

We now introduce **PB-Code**, a provenance-binding SNARK obtained by compiling the HyperPlonk Poly-IOP [10] with the provenance-bound code-based multilinear PCS described above. Let $\mathsf{HP}^{\mathsf{code}} = (\mathsf{Setup}_0, \mathsf{Prove}_0, \mathsf{Verify}_0)$ denote the ordinary HyperPlonk protocol compiled with the plain code-based multilinear PCS. During proving, $\mathsf{HP}^{\mathsf{code}}$ commits to multilinear-polynomial evaluation vectors using Merkle commitments. We index every committed PCS vector by a label a and a folding level i, and write the corresponding vector as $\boldsymbol{\pi}_{a,i} \in \mathbb{F}_p^{n_{a,i}}$. Let $N_{\mathsf{code}} := \sum_{a,i} n_{a,i}$ be the total number of committed vector entries in one honest proof.

The protocol **PB-Code** keeps the HyperPlonk Poly-IOP unchanged. The only change is at the code-based PCS part: every Merkle leaf authenticates the ordinary vector entry together with provenance-dependent randomness and its unique coordinate identifier. We describe the protocol as follows.

Protocol PB-Code

1) $\mathsf{Setup}(1^\lambda, \mathcal{C})$: Run $(\mathsf{pk}_0, \mathsf{vk}_0) \leftarrow \mathsf{Setup}_0(1^\lambda, \mathcal{C})$ for HyperPlonk compiled with the plain code-based multilinear PCS. Fix an efficiently samplable provenance space $\mathcal{I}$ satisfying $|\mathcal{I}| \geq 2^\lambda$, a random-access function Mask, a Merkle commitment instantiation mt_ρ, and a Fiat-Shamir hash H_{FS}. Output $\mathsf{pk} := (\mathsf{pk}_0, \mathsf{Mask}, \mathsf{mt}_\rho, \mathsf{H}_{\mathsf{FS}})$, $\mathsf{vk} := (\mathsf{vk}_0, \mathsf{Mask}, \mathsf{mt}_\rho, \mathsf{H}_{\mathsf{FS}})$, and $\mathcal{I}$.
2) $\mathsf{Prove}(\mathsf{pk}, x, w, \rho)$: Run the HyperPlonk prover $\mathsf{Prove}_0(\mathsf{pk}_0, x, w)$, except that every PCS commitment uses the provenance-dependent leaf hashes described in Section 5.1. Concretely, whenever the ordinary prover would commit to a vector $\boldsymbol{\pi}_{a,i} \in \mathbb{F}_p^{n_{a,i}}$, compute
$$\boldsymbol{r}_{a,i}^{\rho}[j] := \mathsf{Mask}(\mathsf{pk}_0, x, \rho, a, i, j)$$
and the leaf hash
$$\boldsymbol{h}_{a,i}^{\rho}[j] := \mathsf{H}_{\mathsf{leaf}}\big(\rho \| a \| i \| j \| \mathsf{Encode}(\boldsymbol{\pi}_{a,i}[j]) \| \boldsymbol{r}_{a,i}^{\rho}[j]\big).$$
Construct the Merkle root $\mathsf{rt}_{a,i} := \mathsf{mt}_\rho.\mathsf{Commit}(\boldsymbol{\pi}_{a,i})$. Derive all Fiat-Shamir challenges using ρ, e.g., $\mathsf{chal} := \mathsf{H}_{\mathsf{FS}}(\rho \| \mathsf{transcript})$. For every queried oracle coordinate, output the ordinary value and its Merkle authentication path. Output the resulting proof π_ρ.
3) $\mathsf{Verify}(\mathsf{vk}, x, \rho, \pi_\rho)$: Recompute all Fiat-Shamir challenges using ρ. For every opened oracle coordinate (a, i, j), recompute $\boldsymbol{r}_{a,i}^{\rho}[j]$ and $\boldsymbol{h}_{a,i}^{\rho}[j]$, and verify the Merkle path against $\mathsf{rt}_{a,i}$. Then send the authenticated ordinary values to the ordinary HyperPlonk verifier Verify_0. Accept iff Verify_0 accepts.

We show in Theorem 2 that **PB-Code** is $c \cdot t$ provenance binding for some constant c.

Theorem 2 (**PB-Code** is a provenance-binding SNARK)**.** *Assume that* Mask*, the Merkle hashes, and* H_{FS} *are independent domain-separated random oracles. Assume that the*

*underlying code-based multilinear PCS is sound, and that the HyperPlonk Poly-IOP compiled with this PCS is complete, knowledge-sound, and succinct. Assume also that the PCS verifier uses $q_{\mathsf{qry}} = \Theta(\lambda)$ independent query repetitions and satisfies the query-coverage property from Section 5.1. If the encoding time in the plain code-based PCS takes at most an ϵ fraction of the total PCS commit time for a constant $0 \le \epsilon < 1$, then **PB-Code** is a provenance-binding SNARK as defined in Definition 1. Specifically, **PB-Code** is $c' \cdot t_{\mathsf{hon}}$-provenance-binding for some constant $c' > 0$.*

Proof. We first show that **PB-Code** is a SNARK for the same relation as HyperPlonk. Completeness follows from the equivalence of the local consistency checks. For an honest proof, every authenticated value is the same ordinary oracle value that the plain code-based PCS verifier would see. By Lemma 1, the verifier therefore runs exactly the same algebraic checks on these values. Since the underlying HyperPlonk protocol is complete, **PB-Code** is complete.

Succinctness is preserved because the transformed proof contains the same number of Merkle roots, opened values, and authentication paths as the ordinary code-based PCS proof. The verifier performs the ordinary HyperPlonk verification, plus one evaluation of Mask per opened oracle cell. As discussed in Section 5.1, this adds only $O(Q_{\mathsf{cell}})$ hash operations, which is succinct for the query complexity of the underlying PCS.

Knowledge soundness reduces to knowledge soundness of the underlying HyperPlonk protocol. Given an accepting proof for **PB-Code**, the Merkle paths authenticate ordinary oracle values under the committed roots. If the prover opens two inconsistent values at the same position, then it finds a collision in the leaf or internal-node hash, except with negligible probability. Otherwise, the authenticated values satisfy the ordinary PCS consistency checks by Lemma 1. Therefore, the resulting transcript is an accepting execution of HyperPlonk compiled with a binding code-based PCS, and the extractor for the underlying protocol extracts a witness $w^\star$ such that $(x, w^\star) \in \mathcal{R}_{\mathcal{C}}$, except with negligible probability. This argument uses knowledge soundness only to establish the ordinary SNARK property; the online-work lower bound below is proved separately.

It remains to prove provenance binding. We first make the honest cost explicit. Let $n := |\mathcal{C}|$ be the circuit size. For HyperPlonk over multilinear polynomials, the prover performs $O(n)$ work for the HyperPlonk Poly-IOP computation, and the witness polynomials have size $O(n)$ [10]. When it is compiled with a constant-rate code-based multilinear PCS, the total length of all committed PCS vectors satisfies $N_{\mathsf{code}} = \Theta(n)$, up to protocol-dependent constants.

Let T_{enc} denote the time to compute the ordinary encoded PCS vectors in the plain PCS, and let T_{mt} denote the time to Merkle-commit to these vectors. The plain PCS commit time is $T^0_{\mathsf{commit}} := T_{\mathsf{enc}} + T_{\mathsf{mt}}$. By assumption, $T_{\mathsf{enc}} \le \epsilon T^0_{\mathsf{commit}}$ for some constant $\epsilon < 1$. Equivalently, $T_{\mathsf{mt}} \ge (1-\epsilon) T^0_{\mathsf{commit}}$.

In **PB-Code**, after ρ is known, the prover computes one provenance-dependent value and one provenance-dependent leaf hash per committed coordinate, and then constructs the Merkle trees. Let T_{mask} be the time spent on this layer. In the fixed single-core model, $T_{\mathsf{mask}} = \Theta(T_{\mathsf{mt}})$, because the construction uses Merkle trees of the same size and adds only a constant amount of work per leaf. Therefore, the provenance-dependent Merkle layer is a constant fraction of the plain PCS commit time, with a constant depending on ϵ.

The remaining HyperPlonk work is $O(n)$. Since $N_{\mathsf{code}} = \Theta(n)$ and $T_{\mathsf{mask}} = \Theta(N_{\mathsf{code}})$ in the fixed cost model, there exists a constant $C_{\mathsf{hon}} > 0$ such that the honest online prover time satisfies $t_{\mathsf{hon}} \le C_{\mathsf{hon}} \cdot T_{\mathsf{mask}}$.

Now fix any adversary $(\mathcal{A}_0, \mathcal{A}_1)$ in the provenance-binding game. The preprocessing phase occurs before the challenger samples ρ. Suppose $\mathcal{A}_0$ makes random-oracle queries involving at most Q_{pre} candidate provenance strings. The probability that the fresh ρ is one of these candidates is at most $Q_{\mathsf{pre}}/|\mathcal{I}|$, which is negligible. Condition on the event that no provenance-dependent leaf hash for the sampled ρ was computed during preprocessing.

The online adversary may choose its witness and oracle values after receiving ρ. For each committed vector labeled by (a, i), it outputs a Merkle root $\mathsf{rt}_{a,i}$ before the corresponding Fiat-Shamir query positions are determined. Let $S_{a,i} \subseteq [n_{a,i}]$ be the set of positions for which the adversary has made the required provenance-dependent leaf-hash query and determined an authentication path consistent with $\mathsf{rt}_{a,i}$.

If the verifier queries a position outside $S_{a,i}$, the adversary must produce an authentication path for a leaf hash that was not determined when the root was fixed. In the random-oracle model, this requires predicting an unqueried random-oracle output or finding a collision, except with negligible probability.

Suppose that $|S_{a,i}| < (1-\delta) n_{a,i}$. By the query-coverage property, one query repetition touches a position outside $S_{a,i}$ with probability at least $\kappa\delta$. With q_{qry} independent repetitions and Q_{FS} Fiat-Shamir grinding attempts, the probability of avoiding all such positions is at most $Q_{\mathsf{FS}}(1-\kappa\delta)^{q_{\mathsf{qry}}} + \mathsf{negl}(\lambda)$. Since the adversary is polynomial time, $Q_{\mathsf{FS}} \le \mathsf{poly}(\lambda)$. Taking $q_{\mathsf{qry}} = \Theta(\lambda)$ and $\delta = \log^2(\lambda)/\lambda$ makes this probability negligible. Thus any adversary that succeeds with non-negligible probability must satisfy $|S_{a,i}| = (1-o(1)) n_{a,i}$ for every committed vector (a, i).

Summing over all committed vectors gives $\sum_{a,i} |S_{a,i}| = (1-o(1)) N_{\mathsf{code}}$. Every position in $S_{a,i}$ requires a distinct fresh leaf-hash query because the input contains (ρ, a, i, j). This remains true even when the adversary chooses the witness adaptively or different positions contain identical values. Therefore, any successful online adversary performs $\Omega(N_{\mathsf{code}})$ provenance-dependent hash work.

In the fixed machine model, there is therefore a constant $C_{\mathsf{lb}} > 0$ such that every successful online adversary spends at least $C_{\mathsf{lb}} \cdot T_{\mathsf{mask}}$ cycles, except with negligible probability. Combining this with $t_{\mathsf{hon}} \le C_{\mathsf{hon}} \cdot T_{\mathsf{mask}}$, every successful online adversary spends at least $(C_{\mathsf{lb}}/C_{\mathsf{hon}}) \cdot t_{\mathsf{hon}}$ cycles.

Let $c_{\mathsf{PB}} < C_{\mathsf{lb}}/C_{\mathsf{hon}}$ be any fixed constant. Then no adversary running in at most $c_{\mathsf{PB}} \cdot t_{\mathsf{hon}}$ online time can win with non-negligible probability. Therefore **PB-Code** satisfies the provenance-binding definition in Definition 2 with $\sigma(t) = c_{\mathsf{PB}} \cdot t$. □

5.3. Comparison with KZG

Here we discuss why we cannot build a similar protocol with the KZG PCS. In a code-based PCS, the prover commits to a vector, the verifier opens a few coordinates, and the Merkle authentication of every coordinate can be made provenance-dependent. Changing ρ therefore changes every leaf hash and the committed root.

However, the KZG PCS does not have an analogous protocol structure. A KZG commitment is a single group element $[f(\tau)]_1$. Hashing the provenance information together with this constant-size group element does not force a linear number of provenance-dependent operations. Therefore, the provenance-dependent Merkle-leaf construction presented in this section does not directly apply to KZG commitments.

6. Evaluation

We evaluate two aspects of our results. First, we benchmark the preprocessing attack against KZG, showing that provenance-independent preprocessing can reduce the online time needed after the provenance string is known. Second, we benchmark **PB-Code** and measure the time of computation that can only be done after provenance information ρ is given. Recall that the definition in Definition 2 is parameterized by an online-time bound $\sigma(t)$, and the main case is $\sigma(t) = c \cdot t$ for a constant $c \in (0, 1)$. For **PB-Code**, we benchmark the time it takes the prover to go through the Merkle tree operations, which is the lower bound estimate of the online prover work that has to be performed after ρ is known.

We run the benchmarks on an AWS `c7i.16xlarge` instance. All experiments were conducted using a single thread without parallelism, matching the single-core online-time definition in Section 3.1. Our code artifact is publicly accessible online on this page.

6.1. Evaluation of the KZG Preprocessing Attack

We first benchmark the KZG preprocessing attack from Section 4.1 for one KZG commitment and one KZG opening. We note that the commitment is naturally preprocessed, since no randomness is involved. In the KZG opening at a random point, for a degree-d polynomial, we set the block size to $b = \log_2 d$ and build preprocessing tables during the provenance-independent preprocessing phase. The online benchmark compares the ordinary KZG commit-and-open time with the preprocessed online algorithm. Preprocessing time is reported separately.

The results are shown in Table 1. At the largest measured size, $d = 2^{16}$, the honest online time is approximately 1.733 seconds, while the preprocessed online time is approximately 0.59 seconds.

TABLE 1. KZG PREPROCESSING ATTACK AND COMPARISON WITH THE HONEST TIME t_{hon} ACROSS DIFFERENT POLYNOMIAL SIZES. ALL TIMES ARE IN SECONDS.

Polynomial Size	Preprocessing	t_{hon}	Attack Time	Speedup
2^{10}	1.23	0.05	0.01	3.57×
2^{11}	2.81	0.09	0.03	3.38×
2^{12}	6.61	0.15	0.05	3.03×
2^{13}	16.21	0.28	0.09	3.14×
2^{14}	41.97	0.51	0.18	2.91×
2^{15}	117.16	0.94	0.34	2.79×
2^{16}	353.71	1.73	0.59	2.93×

These savings require provenance-independent preprocessing. At $d = 2^{16}$, preprocessing takes approximately 353.71 seconds. With $b = \log_2 d$, the subset-sum tables contain $O((d/b)2^b) = O(d^2/\log d)$ entries, illustrating the storage and preprocessing costs of the tradeoff.

Scaling of the concrete speedup. The speedup remains relatively stable over the measured range and does not exhibit a sustained increasing trend. The honest time is $t_{\mathsf{hon}} = a_F d\tau_F + a_G \frac{d\ell_p}{\log d}\tau_G$ and the attack time is $t_{\mathsf{attack}} \approx b_F d\tau_F + b_G \frac{d\ell_p}{b}\tau_G$ where τ_F and τ_G denote representative costs of field and group operations, respectively, and a_F, b_F, a_G, b_G are implementation-dependent constants. Since we choose $b = \log_2 d$, the dominant terms of group operations in both algorithms scale as $d\ell_p/\log d$. The difference is that in the attack algorithm, the field operation terms also scale with $d/\log d$, while in the honest algorithm the field operation terms scale with d.

Over the range of parameters that we measured, these group operation terms dominate the field operation terms in practice. For example, at $d = 2^{16}$, the honest prover spends 0.02 seconds on field operations and 1.75 seconds on group operations, while the attack spends 0.04 seconds on field operations and 0.55 seconds on group operations. Still, when d is large enough, we expect that the field operations will dominate the group operations. .

6.2. Evaluation of PB-Code PB-Time Share

We now evaluate the provenance-dependent work share of **PB-Code**. In the tables below, *PB Time* denotes the time spent in the Merkle tree computations. We benchmark two settings. First, we benchmark the standalone PCS in **PB-Code** that is instantiated with BaseFold. Second, we benchmark the full SNARK proof generation time of **PB-Code** compiled with HyperPlonk and the BaseFold PCS. In both settings, we use code rate 8.

PCS benchmark. We benchmark the standalone BaseFold PCS across input sizes from 2^{10} to 2^{20}. The results are shown in Table 2. The table reports the total commit-and-open time, the PB time, and the corresponding PB-time share. Across all tested sizes, PB time accounts for between 83.49% and 88.12% of the standalone PCS runtime.

The average PB-time share across the table is 86.81%. At polynomial size 2^{15}, the PCS takes 123.715 ms in total, of which 109.024 ms is PB time (88.12% of the total time).

TABLE 2. **PB-CODE** PCS BENCHMARK ACROSS POLYNOMIAL SIZES. ALL TIMES ARE IN SECONDS.

Polynomial Size	Total Time	PB Time	PB Time Share
2^{10}	0.0042	0.0035	83.49%
2^{11}	0.0081	0.0070	86.28%
2^{12}	0.016	0.014	87.43%
2^{13}	0.031	0.027	87.92%
2^{14}	0.062	0.055	88.11%
2^{15}	0.12	0.11	88.12%
2^{16}	0.25	0.22	87.75%
2^{17}	0.50	0.43	87.34%
2^{18}	1.02	0.88	86.92%
2^{19}	2.08	1.80	86.43%
2^{20}	4.25	3.62	85.09%

End-to-end SNARK benchmark. We next benchmark the full SNARK compiled with HyperPlonk and the BaseFold PCS. We generate random circuits of sizes from 2^{10} to 2^{20} and benchmark the proof generation time. The results are shown in Table 3. Across all tested circuit sizes, PB time accounts for between 41.46% and 50.49% of the end-to-end prover runtime. At 2^{20} constraints, the full prover takes 287 s, of which 119 s is PB time. Therefore, at this parameter size, PB time accounts for 41.46% of the end-to-end prover runtime. This gives an empirical full-prover forced-work fraction of approximately $c_{\mathsf{SNARK}} \approx 0.4146$ at 2^{20} constraints. This supports that $\sigma(t) = c \cdot t$ can hold for a constant c in **PB-Code**.

TABLE 3. **PB-CODE** SNARK BENCHMARK ACROSS DIFFERENT CIRCUIT SIZES. ALL TIMES ARE IN SECONDS.

Circuit Size	Total Time	PB Time	PB Time Share
2^{10}	0.22	0.11	50.41%
2^{11}	0.43	0.22	50.49%
2^{12}	0.88	0.44	49.90%
2^{13}	1.79	0.88	49.09%
2^{14}	3.65	1.75	48.02%
2^{15}	7.46	3.49	46.84%
2^{16}	15.35	7.00	45.59%
2^{17}	31.68	14.13	44.60%
2^{18}	65.89	28.68	43.53%
2^{19}	135.11	57.68	42.69%
2^{20}	287.38	119.15	41.46%

7. Related Work

Hyperproofs and unstealability. Provenance binding is related to the unstealability property proposed in Hyperproofs [27], yet is distinct and strictly stronger as we argued in Section 1.1. Now we provide more details. In Hyperproofs, each prover has a watermarking secret key (WSK) α and the corresponding watermarking public key (WPK) g_2^{α}. A prover makes an opening proof π unstealable by watermarking it as $\pi' = \pi^{\alpha}$. The watermarked proof is verifiable with WPK, yet an attacker without α cannot re-watermark, or steal, π' to another public key. Provenance-binding is strictly stronger, as it also requires that a malicious prover cannot re-watermark its *own* proof with other identities chosen by the attacker without computing the proof from scratch. This is not the case with Hyperproofs, as an attacker can easily create multiple watermarked proofs with different α. Another way to look at the distinction is through the information given to the adversary: in unstealable proofs, the adversary is given just the watermarked proofs π', but in provenance-binding proofs, the adversary is the prover and thus is also given the prover's internal state, in this example, π.

Non-delegatable commitments. Fuchsbauer et al. [12] introduce non-delegatable commitments, which prevent a designated party from outsourcing cryptographic commitment generation without exposing its secret key. However, non-delegatable commitments concern possession of committed data, rather than the computation required for the commitments. Provenance-binding ZK-SNARKs instead require a lower bound on the online work after a fresh provenance string is sampled, even when the prover already knows the witness and retains its prior computation state.

Verifiable delay functions (VDFs). VDFs [7], [23], [29] are functions that require sequential work to evaluate but are efficiently verifiable. VDF cannot be sped up by preprocessing, a property shared by provenance-binding SNARKs at a high level, though they are fundamentally different schemes. *Watermarkable VDFs* [29] is a variant of VDF that is unstealable in the same sense as Hyperproofs, i.e., VDF computation is bound to an evaluator identity, for example, by hashing the watermark into the proof challenge.

Forced work mechanisms: VDFs and proofs of work. VDFs [7], [23], [29] and proofs of work [21] give generic ways to force fresh provenance-dependent effort. Given any SNARK, one could require the prover to evaluate a VDF or solve a hash puzzle on an input derived from the provenance string ρ. E.g., a naive construction is to require each SNARK proof to be accompanied by a VDF on ρ that requires T steps, then an attacker must spend at least another T steps to generate a proof for any $\rho' \neq \rho$. With proper parameterization, this would indeed be provenance-binding, but this approach adds artificial overhead. Our construction (Section 5) achieves provenance binding with minimal additional overhead.

Preprocessing primitives. Our preprocessing attacks are related to performance optimization techniques based on preprocessing. Kedlaya and Umans give data structures for polylogarithmic polynomial evaluation with extensive preprocessing [17] (as opposed to quasi-linear), although their construction is concretely inefficient. Regarding group operations, multi-scalar exponentiations (MSMs) have a long line of optimized algorithms, including Pippenger's bucket

method [24] and follow-ups [20]. In addition, fixed-base exponentiation methods with precomputation trade memory for faster online exponentiation when the bases are known in advance [8], [18]. Our preprocessing attacks of KZG and PLONK make use of the ideas from these works.

Polynomial commitments with succinct opening time. A related line of work designs polynomial commitments with fast online openings after preprocessing. Ateniese et al. [5] construct a succinct polynomial commitment whose opening algorithm runs in time polylogarithmic in the polynomial degree. Campanelli et al. [9] present a polynomial commitment with theoretically fast openings, which is constructed from polynomial preprocessing protocols and vector commitments. These works present new protocols with asymptotically efficient online proofs, but our preprocessing attack aims to speed up existing schemes that are not designed to have preprocessing, and be concretely fast.

Prover markets. Our work is directly applicable to proof markets. A line of work studies markets for outsourced proof generation. Prooφ gives, to our knowledge, the first formal model and construction of prover markets [28]. The Succinct Network proposes a decentralized prover market in which provers compete to generate proofs [25]. More broadly, recent work on procurement mechanisms, such as Beyond Winner-Take-All Procurement Auctions [14], argues that prover markets may benefit from mechanisms that allocate work to multiple provers. These works motivate our setting: in prover markets, a proof should be tied to its provenance information.

SNARKs. ZK-SNARKs are succinct proof systems that allow a verifier to check the correctness of a computation with a short proof and efficient verification. Modern SNARKs are often built from polynomial commitments and polynomial IOPs [10], [11], [13], [16], [26], [30]. Our work studies provenance binding as an additional security dimension for such proof systems, motivated by outsourced proving and prover-market settings.

8. Conclusions and Future Directions

We have initiated the study of provenance-binding ZK-SNARKs, formalizing the requirement that a proof for a fresh provenance string should require substantial fresh online work even after provenance-independent preprocessing. We have shown that simply hashing provenance information into Fiat-Shamir is not sufficient for KZG-PLONK: an adversary can do preprocessing and later output an accepting proof with $o(t_{\mathsf{hon}})$ online work, where t_{hon} is the honest prover time. We have also given a low-overhead provenance-binding SNARK construction from code-based PCS.

Our preprocessing attack for the full KZG-PLONK protocol is currently asymptotic. A natural next step is to develop more efficient attacks and to benchmark them. Another important direction is to design low-overhead provenance-binding SNARKs for KZG-PLONK and other SNARK protocols.

References

[1] Fermah docs. https://docs.fermah.xyz/network.

[2] Lagrange ZK Prover Network. https://lagrange.dev/zk-prover-network.

[3] Marlin Kalypso. https://docs.kalypso.org/learn/.

[4] RiscZero Boundless. https://dev.risczero.com/api/generating-proofs/remote-proving.

[5] Giuseppe Ateniese, Foteini Baldimtsi, Matteo Campanelli, Danilo Francati, and Ioanna Karantaidou. Advancing scalability in decentralized storage: A novel approach to proof-of-replication via polynomial evaluation. In *Annual International Cryptology Conference*, pages 3–39. Springer, 2024.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*, pages 14–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[7] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[8] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 200–207. Springer, 1992.

[9] Matteo Campanelli, Mario Carrillo, Ignacio Cascudo, Dario Fiore, Danilo Francati, and Rosario Gennaro. On the power of polynomial preprocessing: Proving computations in sublinear time, and more. *Cryptology ePrint Archive*, 2025.

[10] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023.

[11] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 738–768. Springer, 2020.

[12] Georg Fuchsbauer, Pranav Garimidi, Guru-Vamsi Policharla, Max Resnick, and Ertem Nusret Tas. Non-delegatable commitments. *Cryptology ePrint Archive*, 2025.

[13] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[14] Pranav Garimidi, Michael Neuder, and Tim Roughgarden. Beyond winner-take-all procurement auctions. *arXiv preprint arXiv:2603.27779*, 2026.

[15] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[16] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*, pages 177–194. Springer, 2010.

[17] Kiran S Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011.

[18] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Annual International Cryptology Conference*, pages 95–107. Springer, 1994.

[19] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[20] Bodo Möller. Algorithms for multi-exponentiation. In *International workshop on selected areas in cryptography*, pages 165–180. Springer, 2001.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[22] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography Conference*, pages 222–242. Springer, 2013.

[23] Krzysztof Pietrzak. Simple verifiable delay functions. In *10th innovations in theoretical computer science conference (itcs 2019)*, pages 60–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

[24] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263. IEEE Computer Society, 1976.

[25] Uma Roy, John Guibas, Kshitij Kulkarni, Mallesh Pai, and Dan Robinson. Succinct network: Prove the world's software. 2024.

[26] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

[27] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, 2022.

[28] Wenhao Wang, Lulu Zhou, Aviv Yaish, Fan Zhang, Ben Fisch, and Benjamin Livshits. Prooφ: A zkp market mechanism. In *International Conference on Financial Cryptography and Data Security*, pages 180–196. Springer, 2025.

[29] Benjamin Wesolowski. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 379–407. Springer, 2019.

[30] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: efficient field-agnostic polynomial commitment schemes from foldable codes. In *Annual International Cryptology Conference*, pages 138–169. Springer, 2024.