TreeFold: Efficient Tree Folding with HyperNova

Daniel Rubin, Simon Judd, John Wu

Abstract

Folding-based proof systems enable efficient recursive proof composition by combining multiple instance-witness pairs into a single proof. However, existing folding schemes process instances sequentially, requiring $O(n)$ steps for n sub-computations. We present TreeFold, a tree-structured folding framework that adapts the HyperNova folding scheme to achieve $O(\log n)$ sequential steps through parallelization.

Our key contributions include: (1) a hybrid commitment structure combining Pedersen and Poseidon schemes for efficient folding and verification, and (2) three specialized augmented circuits that enable secure tree-structured recursion. TreeFold demonstrates significant efficiency gains for large-scale computations while maintaining the security properties of the underlying HyperNova system.

Contents

1 Introduction

Folding schemes have emerged as a powerful technique for constructing efficient recursive SNARKs. These schemes enable the combination of multiple proof instances into a single instance of comparable size, making them ideal for proving large computations composed of repeated sub-circuits. The HyperNova folding scheme [1], based on the CCS (Customizable Constraint System) arithmetization, provides particularly efficient folding through its use of sumcheck protocols and linearization techniques.

Key Point

TreeFold extends HyperNova's folding capabilities to a tree structure, reducing sequential folding steps from $O(n)$ to $O(\log n)$ while maintaining the security properties of the underlying cryptographic system.

1.1 Motivation

Standard folding approaches process instances sequentially: given n instances, the prover folds them one by one into an accumulator, requiring $n-1$ sequential folding operations. This linear dependency prevents parallelization and becomes a bottleneck for large computations. Consider proving the execution of a program with millions of repeated operations—sequential folding would require millions of steps, each dependent on the previous one.

The Mangrove framework [4] introduced tree-structured folding for Plonk-based systems, demonstrating that instances can be folded in a binary tree structure, reducing sequential steps to $O(\log n)$. However, Mangrove's approach relies on Plonk arithmetization and specific polynomial commitment properties that differ significantly from HyperNova's CCS-based architecture.

1.2 Technical Challenges

Adapting tree folding to HyperNova presents several challenges:

1. **Arithmetization Mismatch**: HyperNova uses CCS arithmetization with matrix-based constraints, while tree folding requires careful handling of cross-chunk constraints.
2. **Commitment Compatibility**: HyperNova's folding relies on homomorphic properties of Pedersen commitments, but tree structure requires Merkle-tree aggregation for efficiency.
3. **Verification Circuits**: Tree folding requires augmented circuits at each level to verify the folding operation, which must be adapted to HyperNova's sumcheck-based approach.

1.3 Our Approach

TreeFold addresses these challenges through two key innovations:

Hybrid Commitment Structure: We employ a two-layer approach:

- Pedersen commitments at the chunk level for homomorphic folding
- Poseidon-based Merkle trees for aggregating commitments up the tree

Specialized Augmented Circuits: We design three circuits that work together:

- π_{lin}: Verifies CCS-to-LCCS linearization at leaves
- π_{pcd}: Verifies correct folding at inner nodes
- C_{EC}: Handles elliptic curve operations for commitment folding

1.4 Contributions

Our main contributions are:

1. A complete tree-folding framework for HyperNova that achieves $O(\log n)$ sequential complexity while maintaining security.
2. Rigorous security analysis proving folding correctness, perfect completeness, and knowledge soundness of the tree-structured system.
3. Detailed algorithms for leaf linearization and inner node folding that handle the subtleties of combining CCS instances in a tree structure.

2 Preliminaries

Having established the motivation for tree-structured folding, we now present the foundational concepts necessary to understand our construction. TreeFold builds upon four key technical components: the CCS arithmetization that enables flexible constraint representation, HyperNova's folding mechanism that provides the cryptographic core, commitment schemes that ensure security and efficiency, and the tree folding paradigm that enables our logarithmic complexity reduction.

We work over a prime field $\mathbb{F}$ and use two elliptic curves $\mathbb{G}_1$ and $\mathbb{G}_2$ forming a cycle. We denote by λ the security parameter and by $\text{negl}(\lambda)$ a negligible function.

2.1 CCS Arithmetization

The Customizable Constraint System [3] generalizes various arithmetizations through a matrix-based formulation.

Definition 2.1 (CCS Instance)**.** A CCS instance consists of:

- Structure $s = (m, n, N, \ell, t, q, d)$ where m is input size, n is witness size, N is constraint count
- Matrices $\{M_j\}_{j=1}^{t}$ where each $M_j \in \mathbb{F}^{N \times n}$ is sparse
- Multisets $\{S_i\}_{i=1}^{q}$ where each $S_i \subseteq [t]$ with $|S_i| \leq d$
- Constants $\{c_i\}_{i=1}^{q}$ where $c_i \in \mathbb{F}$

A CCS instance is satisfied by public input $x \in \mathbb{F}^m$ and witness $w \in \mathbb{F}^n$ if:

$$\sum_{i=1}^{q} c_i \cdot \prod_{j \in S_i} \langle M_j \cdot z, z \rangle = 0 \tag{1}$$

where $z = (x, w) \in \mathbb{F}^{m+n}$ is the combined instance-witness vector.

2.2 HyperNova Folding Scheme

HyperNova [1] transforms CCS instances into a linearized form (LCCS) amenable to folding.

Definition 2.2 (LCCS Instance)**.** An LCCS instance is a tuple $U = (C, u, x, r_x, v_1, \ldots, v_t)$ where:

- $C \in \mathbb{G}_1$ is a commitment to the witness

- $u \in \mathbb{F}$ is a scalar (always 1 for unrelaxed instances)
- $x \in \mathbb{F}^m$ is the public input
- $r_x \in \mathbb{F}^s$ is the sumcheck challenge point
- $v_j \in \mathbb{F}$ for $j \in [t]$ are the matrix evaluations

The linearization process uses a sumcheck protocol to reduce the CCS constraint to linear form. Given evaluation point r_x, the linear values are:

$$v_j = \sum_{y \in \{0,1\}^{s'}} M_j(r_x, y) \cdot z(y) \tag{2}$$

2.3 Commitment Schemes

TreeFold employs two commitment schemes with complementary properties:

Pedersen Commitments: For witness vector $w \in \mathbb{F}^n$ and randomness $r \in \mathbb{F}$:

$$C = \mathsf{Commit}(w; r) = \sum_{i=1}^{n} w_i \cdot G_i + r \cdot H \tag{3}$$

where $\{G_i\}_{i=1}^n$ and H are generators of $\mathbb{G}_1$. Pedersen commitments are additively homomorphic: $\mathsf{Commit}(w_1; r_1) + \mathsf{Commit}(w_2; r_2) = \mathsf{Commit}(w_1 + w_2; r_1 + r_2)$.

Poseidon Hash: A SNARK-friendly hash function optimized for arithmetic circuits. We use Poseidon for Merkle tree construction due to its efficiency in-circuit.

2.4 Tree Folding Framework

The tree folding paradigm organizes the proving process hierarchically:

Definition 2.3 (Folding Tree)**.** A folding tree T for n instances is a binary tree where:

- Leaves correspond to the n original instances
- Each internal node represents the fold of its children
- The root contains a single folded instance representing all leaves

The key insight is that all nodes at the same level can be processed in parallel, reducing sequential complexity from $O(n)$ to $O(\log n)$.

3 HyperNova Multifold Algorithm

With the foundational concepts established, we now examine the core HyperNova multifold algorithm [1] that serves as TreeFold's cryptographic foundation. This algorithm is not merely a technical component—it represents a paradigm shift in recursive proof composition that makes our tree-structured approach possible.

The significance of understanding this algorithm cannot be overstated: it transforms the fundamental economics of proof composition from linear to logarithmic complexity. While traditional approaches require $O(n)$ sequential steps that bottleneck large computations, the multifold algorithm enables the batch processing capabilities that TreeFold exploits in its tree topology. It combines two main components:

1. **LCCS Folding**: Using sumcheck protocol to fold Linearized Committed CCS (LCCS) instances

2. **Constraint System Verification**: In-circuit verification of the folding operation

Key Point

The multifold algorithm allows combining multiple proof instances into a single instance while preserving the validity of the underlying statements. This breakthrough enables scalable recursive proof composition and is the key innovation that makes TreeFold's logarithmic complexity reduction mathematically possible—without it, tree folding would lack the necessary homomorphic properties.

3.1 Mathematical Foundations

3.1.1 Core Data Structures

Definition 3.1 (LCCS Instance (Multifold Notation))**.** For multifold operations, an LCCS instance is represented as $U = (W, X, r_s, v_s)$ where:

$$W \in \mathbb{G}_1 \quad \text{(witness commitment)} \tag{4}$$

$$X \in \mathbb{F}^{n_{io}} \quad \text{(public inputs/outputs)} \tag{5}$$

$$r_s \in \mathbb{F}^s \quad \text{(sumcheck randomness)} \tag{6}$$

$$v_s \in \mathbb{F}^t \quad \text{(matrix evaluations)} \tag{7}$$

This notation is equivalent to the TreeFold notation $U = (C, u, x, r_x, v_1, \ldots, v_t)$ with the correspondence: $W \leftrightarrow C$, $X \leftrightarrow x$, $r_s \leftrightarrow r_x$, and $v_s \leftrightarrow (v_1, \ldots, v_t)$.

Definition 3.2 (CCS Witness)**.** A CCS witness consists of:

$$W = (W_1, W_2, \ldots, W_n) \in \mathbb{F}^n \tag{8}$$

$$z = [X[0], X[1..], W] \quad \text{(combined vector)} \tag{9}$$

3.2 Complete Multifold Algorithm

The complete multifold algorithm performs the following high-level steps:

1. Generate challenges using random oracle
2. Construct sumcheck polynomial
3. Execute sumcheck protocol
4. Compute sigma values
5. Fold instances using quadratic weighting

Algorithm 1 HyperNova Multifold: `prove_folding()`

Require: $LCCS_1 = (W_1, X_1, r_{s1}, v_{s1})$, $LCCS_2 = (W_2, X_2, r_{s2}, v_{s2})$
Require: $CCSWitness_1 = W_1$, $CCSWitness_2 = W_2$
Require: $CCSShape$ with matrices M_1, M_2, M_3
Ensure: Folded proof $(\pi_{sumcheck}, LCCS_{folded}, W_{folded})$

Phase 1: Challenge Generation
RO.absorb($LCCS_1$); RO.absorb($LCCS_2$)
$\gamma \leftarrow RO$.squeeze_field_elements(1)[0]
$\rho \leftarrow RO$.squeeze_field_elements(1)[0]

Phase 2: Sumcheck Polynomial Construction
$s \leftarrow |r_{s1}|$ ▷ Number of sumcheck rounds
$z_1 \leftarrow [X_1[0], X_1[1..], W_1]$ ▷ Combined witness vector
$z_2 \leftarrow [X_2[0], X_2[1..], W_2]$
$eq_1 \leftarrow$ EqPolynomial(r_{s1})
$eq_2 \leftarrow$ EqPolynomial(r_{s2})
$g(x) \leftarrow 0$ ▷ Initialize sumcheck polynomial
for $j = 1$ to 3 **do** ▷ For each matrix
    $M_{j,z1} \leftarrow \text{MLE}(M_j \cdot z_1)$
    $M_{j,z2} \leftarrow \text{MLE}(M_j \cdot z_2)$
    $g(x) \leftarrow g(x) + \gamma^j \cdot eq_1(x) \cdot M_{j,z1}(x)$
    $g(x) \leftarrow g(x) + \gamma^{3+j} \cdot eq_2(x) \cdot M_{j,z2}(x)$
end for

Phase 3: Sumcheck Protocol
claimed_sum $\leftarrow \sum_{j=1}^{3} [\gamma^j \cdot v_{s1}[j] + \gamma^{3+j} \cdot v_{s2}[j]]$
$(\pi_{sumcheck}, \text{state}) \leftarrow$ MLSumcheck.prove(RO, g)
$r_{sp} \leftarrow$ state.randomness ▷ Merged evaluation point

Phase 4: Sigma Computation
$\sigma_1 \leftarrow$ compute_sigmas($CCSShape, LCCS_1, W_1, r_{sp}$)
$\sigma_2 \leftarrow$ compute_sigmas($CCSShape, LCCS_2, W_2, r_{sp}$)

Phase 5: Instance Folding
$LCCS_{folded} \leftarrow LCCS_1$.fold($LCCS_2, \rho, \sigma_1, \sigma_2, r_{sp}$)
$W_{folded} \leftarrow W_1$.fold($W_2, \rho$)
return $(\pi_{sumcheck}, LCCS_{folded}, W_{folded})$

3.3 Quadratic Folding Formula

The core folding operation uses **quadratic weighting** for security:

Quadratic Folding Equations:

$$W_{folded}[i] = \rho \cdot W_1[i] + \rho^2 \cdot W_2[i] \tag{10}$$

$$X_{folded}[i] = \rho \cdot X_1[i] + \rho^2 \cdot X_2[i] \tag{11}$$

$$C_{folded} = \rho \cdot C_1 + \rho^2 \cdot C_2 \quad \text{(commitment)} \tag{12}$$

$$v_{folded}[j] = \rho \cdot \sigma_1[j] + \rho^2 \cdot \sigma_2[j] \tag{13}$$

Property 3.3 (Security Property)**.** *Using* (ρ, ρ^2) *instead of* $(\rho, 1-\rho)$ *prevents malicious witness construction attacks.*

4 TreeFold Architecture

Having established the mathematical foundations of HyperNova's multifold algorithm, we now present TreeFold's architectural innovations that transform sequential folding into a logarithmic-complexity tree structure. The central challenge lies in maintaining the security and efficiency properties of HyperNova while reorganizing the computation topology—this requires solving three fundamental architectural problems: organizing instances in a tree topology, managing commitments across multiple tree levels, and handling global constraints that span disconnected chunks.

4.1 System Overview

The TreeFold system operates in three phases:

1. **Leaf Processing**: Linearize CCS instances to LCCS at the leaves
2. **Tree Folding**: Fold LCCS instances pairwise up the tree
3. **Root Verification**: Prove the final folded instance

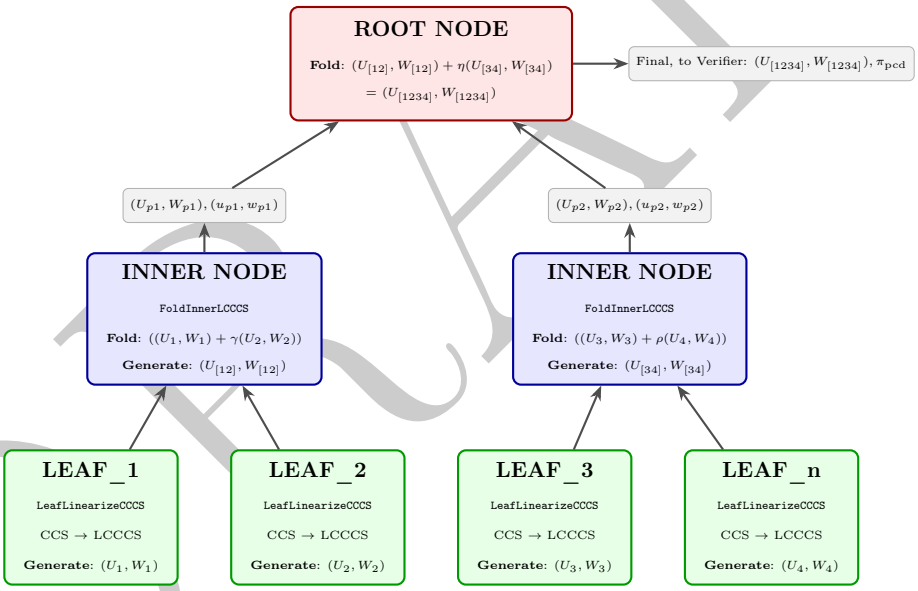

Figure 1: HyperNova Tree structure showing leaf linearization and inner node folding

4.2 Two-Layer Commitment Structure

TreeFold employs different commitment schemes at different levels for optimal efficiency:

4.2.1 Layer 1: Witness Commitments (Pedersen)

Each leaf j commits to its witness using Pedersen:

$$C_j = \text{Pedersen}_{\mathsf{Commit}}(w_j; r_j) \tag{14}$$

These commitments support the homomorphic operations required for folding:

$$C_{\text{fold}} = \rho \cdot C_1 + \rho^2 \cdot C_2 \tag{15}$$

4.2.2 Layer 2: Tree Aggregation (Poseidon)

Pedersen commitments are organized into a Merkle tree using Poseidon:

$$h_w = \text{MerkleTree}_{\text{Poseidon}}([C_1, C_2, \ldots, C_T]) \tag{16}$$

Similarly, public parameters for each leaf are committed:

$$h_{\text{plk}} = \text{MerkleTree}_{\text{Poseidon}}([\text{plk}_1, \text{plk}_2, \ldots, \text{plk}_T]) \tag{17}$$

This two-layer approach provides:

- Efficient folding through Pedersen homomorphism
- Succinct aggregation through Poseidon Merkle trees
- Constant-size root representation regardless of tree size

4.3 Global Copy Constraints

Cross-chunk constraints are handled through a permutation argument:

Definition 4.1 (Permutation Product)**.** For chunk j with permutation σ_j, the partial product is:

$$p_j = \prod_{i \in I_j} \frac{(w_i + \alpha \cdot i) + \beta}{(w_i + \alpha \cdot \sigma_i) + \beta} \tag{18}$$

where (α, β) are challenges derived via Fiat-Shamir.

The global constraint is satisfied when:

$$\prod_{j=1}^{T} p_j = 1 \tag{19}$$

This product is computed incrementally up the tree:

- Leaves compute their local p_j
- Inner nodes multiply: $p_{\text{parent}} = p_{\text{left}} \cdot p_{\text{right}}$
- Root verifies: $p_{\text{root}} = 1$

5 Core Algorithms

With TreeFold's architecture established, we now detail the concrete algorithms that transform our theoretical framework into practical computation. These algorithms bridge the gap between architectural design and implementation, showing precisely how instances flow through the tree structure.

The algorithms work in concert to achieve our efficiency goals: leaf linearization prepares instances for tree processing while preserving HyperNova's security guarantees, and inner node folding leverages the multifold algorithm within our tree topology. Together, they deliver the logarithmic complexity that motivates TreeFold's design.

Algorithm 2 Leaf Linearization

Require: CCS instance (x, w) for operation chunk
Ensure: LCCS instance U, augmented circuit proof

Linearize: Run HyperNova [1] sumcheck to convert CCS to LCCS
  Commit $C \leftarrow \mathsf{Commit}(w)$, obtain evaluation point r_x, linear values $v_1, \ldots, v_t$
  Output $U \leftarrow (C, 1, x, r_x, v_1, \ldots, v_t)$
Tree data: Compute chunk's contribution to global constraints
  $p \leftarrow$ permutation product for this chunk's copy constraints
  $h_w, h_{plk} \leftarrow$ Poseidon hashes of commitment and proving key
Prove: Generate augmented circuit proof containing:
  - Original computation correctness
  - Sumcheck verification
  - Tree folding data (p, h_w, h_{plk})
return LCCS instance U and augmented proof

5.1 Leaf Linearization

The leaf prover transforms a fresh CCS instance into LCCS format suitable for tree folding. The augmented circuit at the leaf encodes several constraints:

1. **Application Logic**: The original computation (SHA-256, ECDSA, etc.)
2. **Tree Folding Gadgets**:
   - Permutation product computation
   - Witness commitment and hashing
   - Parameter commitment and hashing
3. **Sumcheck Verification**: Ensures the linearization was performed correctly

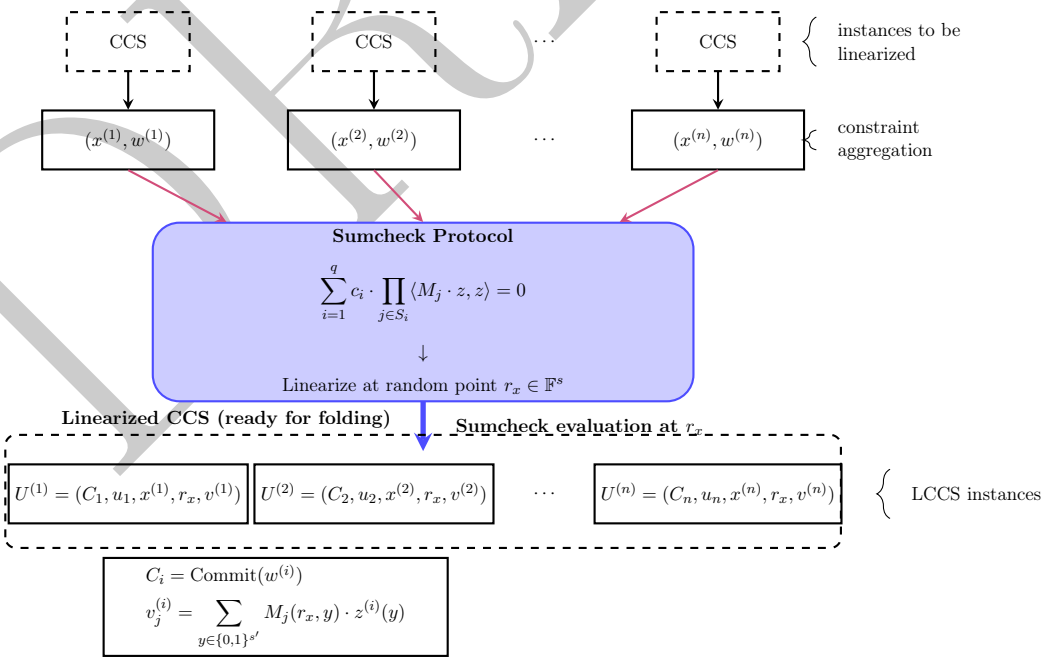

Figure 2: CCS to LCCS linearization process showing sumcheck transformation

5.2 Inner Node Folding

Inner nodes fold two LCCS instances from their children into one.

Algorithm 3 Inner Node Folding

Require: Two LCCS instances U_1, U_2 from child nodes
Ensure: Folded LCCS instance U_F, augmented circuit proof
1: **Fold:** Run HyperNova folding protocol to combine U_1, U_2
2: Derive folding challenge ρ, compute linear combination
3: Output $U_F \leftarrow \rho \cdot U_1 + \rho^2 \cdot U_2$
4: **Tree data:** Aggregate children's contributions to global constraints
5: $p_F \leftarrow p_1 \cdot p_2$ (multiply permutation products)
6: $h_w, h_{plk} \leftarrow$ Poseidon aggregation of child hashes
7: **Prove:** Generate augmented circuit proof containing:
8: - Folding correctness verification
9: - Tree constraint aggregation
10: - Hash tree updates
11: **return** Folded LCCS instance U_F and augmented proof

5.3 Augmented Circuits Design

TreeFold employs three specialized circuits:

5.3.1 π_{lin}: Leaf Linearization Circuit

Circuit $F'^{\text{lin}}_{\alpha,\beta}(x, w, r_x, v_1, \ldots, v_t, p, h_w, h_{plk})$ verifies CCS-to-LCCS transformation. The circuit enforces the following constraints:

Leaf Linearization Circuit Constraints

1. **Original CCS constraints:**

$$\sum_{i=1}^{q} c_i \prod_{j \in S_i} \langle M_j z, z \rangle = 0 \tag{20}$$

2. **Linearization correctness:** For all $j \in [t]$:

$$v_j = \sum_{y \in \{0,1\}^{s'}} M_j(r_x, y) z(y) \tag{21}$$

3. **Commitment hash integrity:**

$$h_w = \text{Poseidon}(\mathsf{Commit}(w)) \tag{22}$$

4. **Proving key hash integrity:**

$$h_{plk} = \text{Poseidon}(\text{pk}) \tag{23}$$

5. **Permutation product computation:**

$$p = \prod_{i \in \text{chunk}} \frac{w_i + \alpha \cdot i + \beta}{w_i + \alpha \cdot \sigma(i) + \beta} \tag{24}$$

5.3.2 π_{pcd}: Inner Node Folding Circuit

Circuit $F_\rho^{\text{pcd}}(U_1, U_2, p_1, p_2, h_{w,1}, h_{w,2}, h_{plk,1}, h_{plk,2})$ verifies correct folding of child instances. The circuit enforces:

Inner Node Folding Circuit Constraints

1. **Instance folding:**

$$U_F = \rho \cdot U_1 + \rho^2 \cdot U_2 \tag{25}$$

2. **Permutation product aggregation:**

$$p_F = p_1 \cdot p_2 \tag{26}$$

3. **Witness hash aggregation:**

$$h_{w,F} = \text{Poseidon}(h_{w,1} \| h_{w,2}) \tag{27}$$

4. **Parameter hash aggregation:**

$$h_{plk,F} = \text{Poseidon}(h_{plk,1} \| h_{plk,2}) \tag{28}$$

5. **Commitment folding:**

$$C_F = \rho \cdot C_1 + \rho^2 \cdot C_2 \tag{29}$$

5.3.3 C_{EC}: Elliptic Curve Circuit

Circuit $F_\rho^{\text{EC}}(C_1, C_2 \in \mathbb{G}_1, \rho)$ handles elliptic curve arithmetic in the base field. This circuit is necessary because the main circuit operates over a different field than the commitment curve.

Elliptic Curve Circuit Constraints

1. **Elliptic curve addition:**

$$C_{\text{out}} = \rho \cdot C_1 + \rho^2 \cdot C_2 \tag{30}$$

2. **Group membership:**

$$C_1, C_2, C_{\text{out}} \in \mathbb{G}_1 \tag{31}$$

3. **Implementation note:** When the main circuit uses BN254, this circuit operates over Grumpkin to form a 2-cycle [6].

6 Security Analysis

Having detailed TreeFold's implementation through concrete algorithms, we now establish the security guarantees that ensure our efficiency gains do not compromise cryptographic soundness. The transition from sequential to tree-structured folding must preserve the essential security properties that make HyperNova trustworthy.

We establish three fundamental security properties for TreeFold: folding correctness ensures that tree operations preserve statement validity, perfect completeness guarantees that honest provers always succeed, and knowledge soundness prevents malicious provers from creating false proofs. These properties collectively ensure that TreeFold's logarithmic efficiency comes without security trade-offs.

6.1 Folding Correctness

The folding operation must preserve the validity of the underlying statements.

Theorem 6.1 (Folding Correctness)**.** *Given valid instances $(X_1, W_1), (X_2, W_2) \in R_{LCCS}$ and folding challenge ρ, the folded instance (X_F, W_F) computed by TreeFold satisfies $(X_F, W_F) \in R_{LCCS}$.*

Proof Sketch. The proof proceeds by checking each component:

Commitment Correctness: By Pedersen homomorphism:

$$\mathsf{Commit}(W_F) = \mathsf{Commit}(\rho W_1 + \rho^2 W_2) = \rho\mathsf{Commit}(W_1) + \rho^2\mathsf{Commit}(W_2) = C_F \tag{32}$$

Linear Constraints: For linear values v_j:

$$v_{j,F} = \langle a_j, W_F \rangle = \langle a_j, \rho W_1 + \rho^2 W_2 \rangle = \rho v_{j,1} + \rho^2 v_{j,2} \tag{33}$$

Non-linear Aggregation: Permutation products multiply correctly since chunks have disjoint domains. Merkle hashes aggregate via Poseidon maintaining tree structure. □

6.2 Perfect Completeness

Valid computations must always produce accepting proofs.

Theorem 6.2 (Perfect Completeness)**.** *For every valid witness $(w_1, \ldots, w_n)$ satisfying the computation, TreeFold produces a proof that the verifier accepts with probability 1.*

The key insight is that validity propagates from leaves to root:

- Leaves: CCS constraints satisfied ⇒ linearization succeeds
- Inner nodes: Valid children ⇒ folding produces valid parent
- Root: Permutation product equals 1, Merkle roots match commitments

6.3 Knowledge Soundness

The system must be secure against adversaries attempting to prove false statements.

Theorem 6.3 (Knowledge Soundness)**.** *For every PPT adversary $\mathcal{A}$ producing an accepting proof, there exists an efficient extractor $\mathcal{E}$ that outputs a valid witness except with negligible probability.*

The extraction works in three layers:

1. **PCD Extraction**: Extract the entire proof tree from the root proof
2. **Folding Extraction**: For each inner node, extract child witnesses
3. **Leaf Extraction**: Extract original computation witnesses from leaves

Crucially, the folding operation acts as a "knowledge barrier"—producing a valid folding proof requires knowing valid child witnesses, preventing forgeries from propagating up the tree.

7 References

References

[1] Abhiram Kothapalli and Srinath Setty. *HyperNova: Recursive arguments for customizable constraint systems*. Cryptology ePrint Archive, Paper 2023/573, 2023. `https://eprint.iacr.org/2023/573`

[2] Tianyu Zheng, Shang Gao, Yu Guo, and Bin Xiao. *KiloNova: Non-Uniform PCD with Zero-Knowledge Property from Generic Folding Schemes*. Cryptology ePrint Archive, Paper 2023/1579, 2023. `https://eprint.iacr.org/2023/1579`

[3] Srinath Setty, Justin Thaler, and Riad Wahby. *Customizable constraint systems for succinct arguments*. Cryptology ePrint Archive, Paper 2023/552, 2023. `https://eprint.iacr.org/2023/552`

[4] Wilson Nguyen, Trisha Datta, Binyi Chen, Nirvan Tyagi, and Dan Boneh. *Mangrove: A Scalable Framework for Folding-based SNARKs*. Cryptology ePrint Archive, Paper 2024/416, 2024. `https://eprint.iacr.org/2024/416`

[5] Simon Judd. *EndGame: Field-Agnostic Succinct Blockchain with Arc*. Cryptology ePrint Archive, Paper 2024/1925, 2024. `https://eprint.iacr.org/2024/1925`

[6] Abhiram Kothapalli and Srinath Setty. *CycleFold: Folding-scheme-based recursive arguments over a cycle of elliptic curves*. Cryptology ePrint Archive, Paper 2023/1192, 2023. `https://eprint.iacr.org/2023/1192`

[7] Srinath Setty. *Folding Schemes: Why they matter and how they are not employed in the best way possible*. HackMD, 2024. `https://hackmd.io/@srinathsetty/folding-schemes`