Verifiable Federated Learning with Zero-Knowledge Proofs: A Cryptographic Framework for Privacy-Preserving Distributed Training

Tarek Salama, Zeyad Elshafey, Ahmed Elbehiry

Department of Computer Science, Purdue University

CS 590: Applied Cryptography, Fall 2025

Abstract

Federated learning enables collaborative model training across distributed data sources without centralizing raw data, addressing critical privacy concerns in domains such as healthcare, finance, and government. However, standard federated learning protocols remain vulnerable to several attack vectors: malicious clients may submit poisoned gradients to corrupt the global model, curious servers may infer private information from individual gradient updates, and participants lack mechanisms to verify that training was performed correctly. In this paper, we present a comprehensive cryptographic framework that addresses these challenges through the integration of zero-knowledge proofs (ZKPs), Merkle tree commitments, and secure aggregation protocols. Our system comprises three interconnected components: (1) a balance proof circuit that verifies dataset properties without revealing individual data points, (2) a training integrity circuit that proves gradient computation correctness and norm bounding, and (3) a secure aggregation circuit that enables privacy-preserving gradient aggregation where the server learns only the aggregate update. We implement our framework using the Circom circuit compiler and Groth16 proving system, achieving proof generation times of approximately 45 seconds for a complete training round with three clients. Our evaluation demonstrates that the framework successfully prevents gradient manipulation attacks while maintaining computational practicality for real-world deployment scenarios. The cryptographic binding between components ensures that proofs generated for one dataset cannot be reused for another, providing end-to-end integrity guarantees throughout the federated learning pipeline.

1. Introduction

1.1 Motivation and Problem Context

The proliferation of machine learning applications across sensitive domains has created a fundamental tension between the need for large-scale training data and increasingly stringent privacy regulations. Healthcare institutions seeking to develop diagnostic models require access to diverse patient populations, yet regulations such as the Health Insurance Portability and Accountability Act (HIPAA) in the United States and the General Data Protection Regulation (GDPR) in Europe impose strict constraints on data sharing. Similarly, financial institutions aiming to build fraud detection systems cannot freely exchange customer transaction data due to competitive concerns and regulatory requirements. This tension between data utility and privacy protection represents one of the central challenges in modern machine learning deployment.

Federated learning [1] emerged as a promising paradigm to address this challenge by enabling collaborative model training without centralizing raw data. In the federated learning framework, a central server coordinates training across multiple clients, each holding private local datasets. Rather than transmitting raw data, clients compute gradient updates locally and send only these updates to the server, which aggregates them to improve a global model. This approach has been successfully deployed in production systems, including Google's keyboard prediction [2] and Apple's on-device machine learning [3], demonstrating its practical viability at scale.

However, the transition from centralized to federated training introduces new security challenges that undermine the privacy guarantees that motivate the paradigm. First, gradient updates themselves can leak substantial information about the underlying training data. Zhu et al. [4] demonstrated that individual training examples can be reconstructed from gradient updates with high fidelity, a vulnerability known as gradient inversion. Second, the distributed nature of federated learning creates opportunities for Byzantine attacks, where malicious clients submit carefully crafted gradient updates designed to poison the global model [5]. Third, the server in standard federated learning protocols observes individual gradient updates, creating a trusted aggregator assumption that may not hold in adversarial settings. Finally, there exists no mechanism for verifying that clients actually performed the claimed computation on legitimate data—a client could fabricate gradient updates without performing any actual training.

1.2 Research Objectives

The goal of this work is to develop a cryptographic framework that provides formal guarantees addressing the vulnerabilities described above. Specifically, we aim to achieve the following properties:

Training Integrity: Clients must prove that their submitted gradients were computed correctly from data they committed to at the beginning of the protocol. This prevents both fabrication of gradients and training on unauthorized datasets.

Gradient Privacy: The aggregation server should learn only the aggregate gradient across all participating clients, not individual contributions. This property protects against gradient inversion attacks by curious servers.

Dataset Property Verification: Before accepting a client's participation, the server should be able to verify properties of the client's dataset (such as class balance) without learning the actual data distribution.

Gradient Norm Bounding: To prevent model poisoning attacks, the system must cryptographically enforce that submitted gradients satisfy magnitude constraints, ensuring no single client can disproportionately influence the model.

Cryptographic Binding: All proofs generated by a client must be bound to a single committed dataset, preventing mix-and-match attacks where a client proves properties on one dataset while training on another.

1.3 Approach Overview

We address these objectives through a modular architecture comprising three zero-knowledge proof circuits that work together to provide comprehensive security guarantees:

Component A (Balance Proof): This circuit enables clients to prove properties of their local datasets without revealing individual data points. Specifically, clients prove that their dataset contains specified numbers of samples from each class, demonstrating dataset balance properties that are important for training unbiased

models. The circuit verifies Merkle tree membership for all samples, ensuring that the proven properties correspond to the committed dataset.

Component B (Training Integrity Proof): This circuit proves that a client's submitted gradient satisfies two critical properties: (1) the gradient was computed using samples from the committed dataset, verified through Merkle proofs, and (2) the gradient's squared norm does not exceed a specified threshold τ^2, enforcing the clipping constraint. A key technical contribution is our solution to the signed arithmetic challenge—since zero-knowledge circuits operate over finite fields where negative numbers do not exist natively, we develop a sign-magnitude decomposition approach that enables sound norm verification.

Component C (Secure Aggregation Proof): This circuit implements a verifiable secure aggregation protocol where clients mask their gradients using pairwise random values that cancel upon aggregation. The circuit proves that the masked gradient was correctly constructed from the committed gradient (linking to Component B) and properly derived masking values, without revealing either the original gradient or the masks.

The three components are cryptographically bound through shared commitments. Components A and B both compute Merkle tree roots over the dataset using identical leaf hash functions, ensuring that any dataset substitution would be detected as a root mismatch. Similarly, Components B and C share a gradient commitment, preventing clients from proving training integrity for one gradient while submitting a different masked gradient.

1.4 Contributions

This paper makes the following contributions:

1. **Integrated Framework Design:** We present the first complete framework integrating dataset property verification, training integrity proofs, and secure aggregation within a unified zero-knowledge proof system for federated learning.

2. **Sound Gradient Norm Verification:** We identify and resolve a critical vulnerability in naive approaches to gradient clipping verification within finite field arithmetic, providing a provably sound construction based on sign-magnitude decomposition.

3. **Cryptographic Binding Mechanism:** We develop a commitment-based binding approach that ensures all three proof components reference the same underlying dataset and gradient, preventing composition attacks.

4. **Complete Implementation:** We provide a fully functional implementation using the Circom circuit language and snarkjs library, demonstrating practical feasibility with comprehensive test coverage.

5. **Security Analysis:** We formally analyze the security properties of our construction, identifying both the guarantees provided and the limitations of our threat model.

1.5 Paper Organization

The remainder of this paper is organized as follows. Section 2 provides background on the cryptographic primitives underlying our construction, including zero-knowledge proofs, Merkle trees, and secure aggregation. Section 3 describes our threat model and presents the detailed methodology for each

component. Section 4 presents our experimental evaluation. Section 5 discusses limitations and future directions. Section 6 surveys related work, and Section 7 concludes.

---

2. Background

This section provides the cryptographic and mathematical foundations necessary to understand our framework. We present each concept with sufficient rigor to support the technical developments in subsequent sections while remaining accessible to readers with general cryptography background.

2.1 Finite Field Arithmetic

Zero-knowledge proof systems operate over finite fields rather than integers or real numbers. Understanding finite field arithmetic is essential for grasping both the capabilities and limitations of our circuits.

Definition 2.1 (Prime Field). A prime field $\mathbb{F}_p$ is the set of integers ${0, 1, 2, \ldots, p-1}$ equipped with addition and multiplication operations performed modulo p, where p is a prime number.

The choice of prime field determines the security level of the cryptographic system. Our implementation uses the scalar field of the BN254 elliptic curve:

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

This 254-bit prime provides approximately 128 bits of security against known attacks, aligning with current cryptographic recommendations.

Field Operations. Within $\mathbb{F}_p$, all arithmetic operations wrap around modulo p:

- Addition: $a + b \pmod{p}$
- Multiplication: $a \cdot b \pmod{p}$
- Subtraction: $a - b \equiv a + (p - b) \pmod{p}$
- Division: $a / b \equiv a \cdot b^{-1} \pmod{p}$, where b^{-1} is the multiplicative inverse

The multiplicative inverse b^{-1} exists for all $b \neq 0$ and can be computed efficiently using the extended Euclidean algorithm or Fermat's little theorem: $b^{-1} \equiv b^{p-2} \pmod{p}$.

Representing Negative Numbers. A critical observation for our work is that finite fields have no notion of "negative" numbers in the conventional sense. The value $-x$ in $\mathbb{F}_p$ is represented as $p - x$. This creates challenges for operations that depend on sign, such as computing squared norms of vectors with negative components, which we address in Section 3.

2.2 Elliptic Curve Cryptography

Our proof system relies on elliptic curve groups for its security guarantees. We provide a brief overview of the relevant concepts.

Definition 2.2 (Elliptic Curve). An elliptic curve E over a prime field $\mathbb{F}_p$ is the set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying the Weierstrass equation:

$$y^2 = x^3 + ax + b \pmod{p}$$

together with a special "point at infinity" $\mathcal{O}$ serving as the group identity element, where $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$.

The set of points on an elliptic curve forms an abelian group under a geometrically-defined addition operation. This group structure enables cryptographic constructions based on the hardness of the discrete logarithm problem.

Definition 2.3 (Elliptic Curve Discrete Logarithm Problem). Given points P and $Q = kP$ on an elliptic curve, where kP denotes k-fold addition of P to itself, find the scalar k. For appropriately chosen curves, this problem is believed to be computationally intractable.

Pairing-Friendly Curves. Our proof system (Groth16) requires bilinear pairings—efficiently computable maps $e: \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ between elliptic curve groups satisfying $e(aP, bQ) = e(P, Q)^{ab}$. The BN254 curve we use is specifically designed for efficient pairing computation while maintaining security.

2.3 Cryptographic Hash Functions

Hash functions are fundamental building blocks in our construction, used for commitments, Merkle trees, and various binding operations.

Definition 2.4 (Cryptographic Hash Function). A cryptographic hash function $H: \{0,1\}^* \rightarrow \{0,1\}^n$ maps arbitrary-length inputs to fixed-length outputs and satisfies the following security properties:

1. **Preimage Resistance:** Given h, it is computationally infeasible to find any m such that $H(m) = h$.
2. **Second Preimage Resistance:** Given m_1, it is computationally infeasible to find $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$.
3. **Collision Resistance:** It is computationally infeasible to find any pair $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$.

The Poseidon Hash Function. Standard hash functions like SHA-256 are designed for efficient software and hardware implementation but are extremely expensive to evaluate within zero-knowledge circuits due to their reliance on bitwise operations. Poseidon [6] is a hash function specifically designed for efficient arithmetic circuit implementation.

Poseidon operates directly on field elements rather than bits, using only field additions and multiplications. Its structure consists of rounds applying:

1. Round constant addition
2. S-box layer (computing $x \mapsto x^5$ for full rounds, applied to one element for partial rounds)
3. Linear mixing via an MDS (Maximum Distance Separable) matrix

The result is a hash function requiring approximately 250 constraints per invocation in R1CS representation, compared to over 25,000 constraints for SHA-256. This 100× improvement is critical for practical zero-knowledge proof generation.

Poseidon Parameters. The specific Poseidon instantiation we use accepts up to 16 field elements as input and produces a single field element as output. For larger inputs, we employ a chunked hashing strategy described in Section 3.

2.4 Merkle Trees

Merkle trees provide a mechanism for committing to large datasets while enabling efficient membership proofs for individual elements.

Definition 2.5 (Merkle Tree). A Merkle tree over a dataset $D = {d_0, d_1, \ldots, d_{n-1}}$ is a complete binary tree where:

- Each leaf node i contains $H(d_i)$, the hash of data element d_i
- Each internal node contains the hash of the concatenation of its children: $H(\text{left} | \text{right})$
- The root node provides a single commitment to the entire dataset

Merkle Proofs. To prove that element d_i is the i-th element of a dataset with root r, the prover provides:

- The authentication path: sibling hashes along the path from leaf i to the root
- The path indices: a sequence of bits indicating whether each node is a left or right child

The verifier computes $H(d_i)$, then iteratively hashes with siblings according to the path indices, checking that the final result equals r. For a tree with n leaves, the proof size is $O(\log n)$ hashes.

Security Properties. Under the collision resistance of the underlying hash function:

- **Binding:** It is computationally infeasible to find two different datasets with the same Merkle root
- **Membership Soundness:** It is computationally infeasible to produce a valid Merkle proof for an element not in the committed dataset

These properties are essential for our construction, as they ensure that proofs about dataset properties (Component A) and training batch membership (Component B) cannot be forged.

2.5 Zero-Knowledge Proofs

Zero-knowledge proofs enable a prover to convince a verifier that a statement is true without revealing any information beyond the statement's validity.

Definition 2.6 (Zero-Knowledge Proof System). A zero-knowledge proof system for a language $\mathcal{L}$ with witness relation $\mathcal{R}$ consists of algorithms (Setup, Prove, Verify) satisfying:

1. **Completeness:** For all $(x, w) \in \mathcal{R}$, an honest prover with witness w can convince the verifier that $x \in \mathcal{L}$ with overwhelming probability.

2. **Soundness:** For all $x \notin \mathcal{L}$, no computationally bounded prover can convince the verifier that $x \in \mathcal{L}$ except with negligible probability.

3. **Zero-Knowledge:** There exists a simulator that, given only $x \in \mathcal{L}$ (without w), can produce transcripts indistinguishable from real proof transcripts. This implies the verifier learns nothing beyond the truth of the statement.

zk-SNARKs. Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) are a class of zero-knowledge proofs with particularly attractive efficiency properties:

- **Succinct:** Proof size is constant (independent of computation size)

- **Non-Interactive:** The proof is a single message from prover to verifier
- **Argument of Knowledge:** The prover must "know" a valid witness (not merely that one exists)

The succinctness property is crucial for our application, as it enables efficient verification regardless of the complexity of the proven computation.

2.6 The Groth16 Proving System

We employ the Groth16 [7] proving system, which achieves optimal proof size among pairing-based zk-SNARKs.

Rank-1 Constraint Systems (R1CS). Groth16 operates on computations expressed as R1CS, a system of equations of the form:

$$(\mathbf{a}_i \cdot \mathbf{s}) \times (\mathbf{b}_i \cdot \mathbf{s}) = (\mathbf{c}_i \cdot \mathbf{s})$$

where $\mathbf{s}$ is the witness vector (containing public inputs, private inputs, and intermediate values) and $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i$ are coefficient vectors defining the i-th constraint. Any arithmetic computation can be expressed in R1CS form.

Trusted Setup. Groth16 requires a trusted setup ceremony that generates proving and verification keys from the circuit description and random "toxic waste" τ. The security of the system depends on τ being unknown to any party after the ceremony. In practice, multi-party computation ceremonies are used to ensure that τ remains secret as long as at least one participant behaves honestly.

Proof Structure and Verification. A Groth16 proof consists of three elliptic curve group elements $(A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$, totaling approximately 192 bytes. Verification requires computing three bilinear pairings and checking an equation, completing in approximately 10 milliseconds on modern hardware.

Complexity. Proof generation requires $O(m \log m)$ group exponentiations where m is the number of constraints, with practical times ranging from seconds to minutes depending on circuit size. The proving process is computationally intensive but can be performed offline before the results are needed.

2.7 The Circom Language

Circom [8] is a domain-specific language for defining arithmetic circuits that compile to R1CS. We briefly describe its key features.

Signals and Constraints. Circom programs define signals (variables) and constraints (equations) over them. Signals are categorized as:

- `signal input`: Values provided by the prover
- `signal output`: Values computed and exposed as outputs
- `signal` (intermediate): Internal computation values

Constraints are specified using the `===` operator, which creates an R1CS equation. Only quadratic expressions are permitted: products of at most two signals.

Templates. Templates are parameterized circuit components that can be instantiated with different parameters. For example:

```
template Multiplier() {
    signal input a;
    signal input b;
    signal output c;

    c <== a * b;  // Assigns AND constrains
}
```

The `<==` operator both assigns a value and creates a constraint, while `<--` assigns without constraining (requiring careful use to maintain soundness).

2.8 Federated Learning

Federated learning [1] is a distributed machine learning paradigm where training occurs across multiple clients holding local datasets, coordinated by a central server.

Protocol Overview. A typical federated learning round proceeds as follows:

1. The server broadcasts the current global model W_t to participating clients
2. Each client i computes a gradient update $g_i = \nabla L(W_t; D_i)$ on their local data D_i
3. Clients send gradients to the server
4. The server aggregates: $W_{t+1} = W_t - \eta \cdot \frac{1}{n}\sum_{i=1}^{n} g_i$
5. Repeat until convergence

Gradient Clipping. To bound the influence of any single client and enable differential privacy guarantees, gradient clipping is commonly employed:

$$g'_i = g_i \cdot \min\left(1, \frac{\tau}{|g_i|_2}\right)$$

This ensures $|g'_i|_2 \leq \tau$ for all clients. In our framework, we verify the clipping constraint cryptographically rather than trusting clients to apply it.

2.9 Secure Aggregation

Secure aggregation [9] enables computing the sum of client inputs without revealing individual contributions.

Pairwise Masking Protocol. The approach we implement uses pairwise random masks that cancel upon aggregation:

1. Each pair of clients (i, j) establishes a shared random seed s_{ij}
2. From s_{ij}, both clients derive the same pseudorandom mask r_{ij}
3. Client i adds $+r_{ij}$ if $i < j$ or $-r_{ij}$ if $i > j$ to their gradient
4. When summing across all clients, each mask appears once with $+$ and once with $-$, canceling out

Formally, client i submits: $$m_i = g_i + \sum_{j \neq i} \sigma_{ij} \cdot r_{ij}$$

where $\sigma_{ij} = +1$ if $i < j$ and $\sigma_{ij} = -1$ if $i > j$. The aggregate is: $$\sum_i m_i = \sum_i g_i + \sum_i \sum_{j \neq i} \sigma_{ij} \cdot r_{ij} = \sum_i g_i$$

The second term vanishes because each r_{ij} appears exactly twice with opposite signs.

Dropout Handling. If clients drop out, their masks do not cancel. Practical protocols use threshold secret sharing to enable reconstruction of dropped clients' masks from surviving clients' shares, ensuring protocol completion.

---

3. Methodology

This section presents our threat model, system architecture, and detailed component designs. We describe the cryptographic mechanisms that bind the three proof components together and analyze the security guarantees achieved.

3.1 Threat Model

We consider a federated learning system with n clients and a central aggregation server. Our threat model addresses several distinct adversarial capabilities:

Malicious Clients. We assume up to $n-1$ clients may be fully malicious, controlled by an adversary who can:

- Submit arbitrary gradient updates not computed from their claimed training data
- Attempt to poison the global model through carefully crafted malicious gradients
- Collude with other malicious clients to mount coordinated attacks
- Attempt to claim dataset properties (such as balance) that do not hold

Curious Server. The aggregation server is assumed to follow the protocol correctly (semi-honest) but attempts to learn private information from observed messages. Specifically, the server may:

- Record and analyze all received gradient updates
- Attempt gradient inversion attacks to reconstruct training data
- Profile clients based on gradient patterns across rounds

Network Adversary. We assume authenticated channels between clients and the server, preventing message modification in transit. However, traffic analysis is possible.

Trusted Setup. We assume the Groth16 trusted setup was performed correctly using a multi-party computation ceremony where at least one participant was honest. This is a standard assumption for deployed Groth16 systems.

What We Do Not Protect Against. Our threat model explicitly excludes:

- A malicious aggregation server that deviates from the protocol
- Denial-of-service attacks (clients refusing to participate)
- Side-channel attacks on the proof generation process
- Attacks on the underlying machine learning model (e.g., membership inference against the final model)

3.2 System Architecture

Our framework consists of three zero-knowledge proof circuits that work together to provide comprehensive security guarantees. Figure 1 illustrates the high-level architecture.

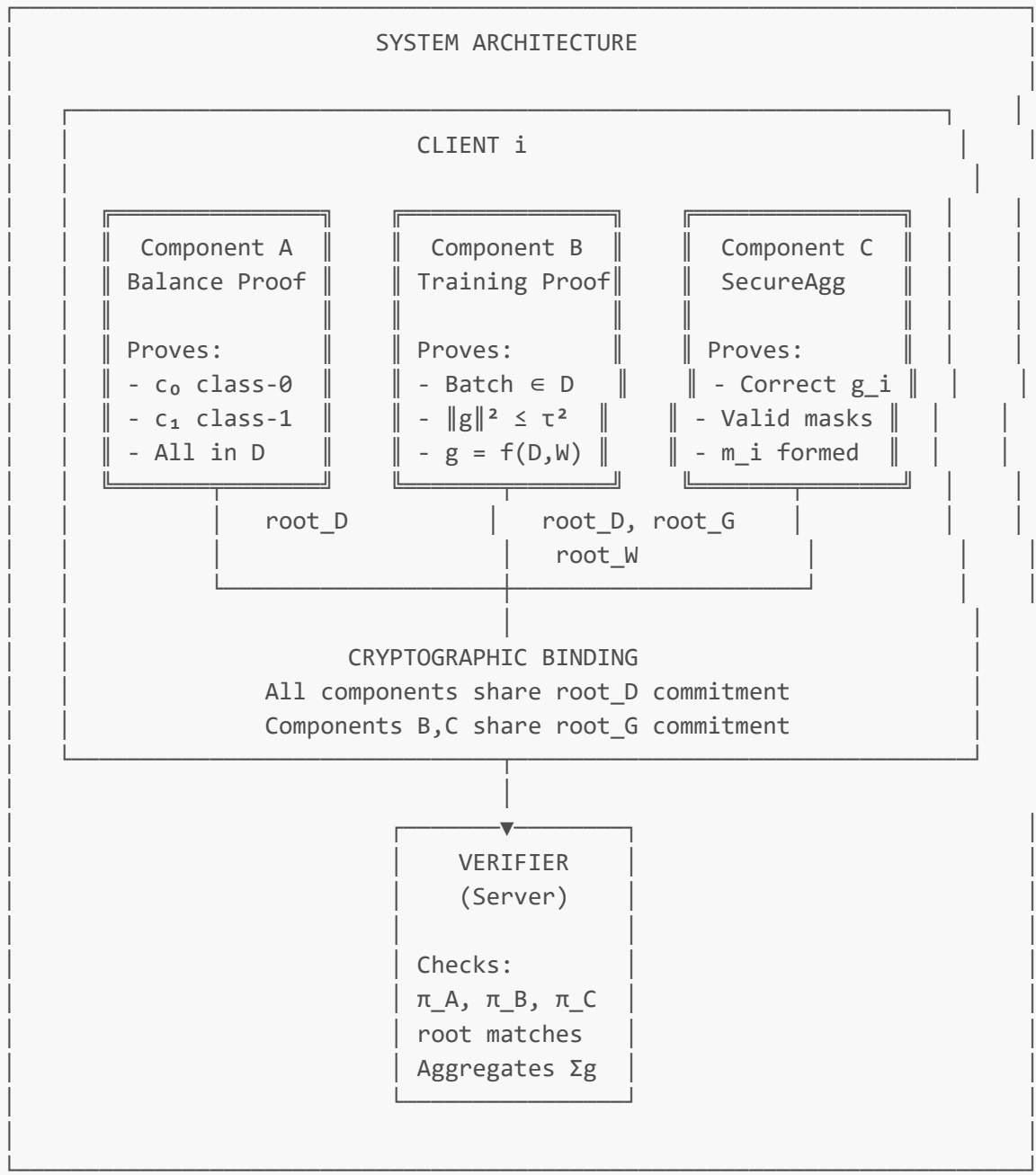


Figure 1

Protocol Flow. A complete federated learning round proceeds in five phases:

Phase 1: Setup. The server distributes the current global model weights W_t and protocol parameters (clipping threshold τ^2, round number r) to all clients. Each client commits to their dataset by computing a Merkle tree over their samples.

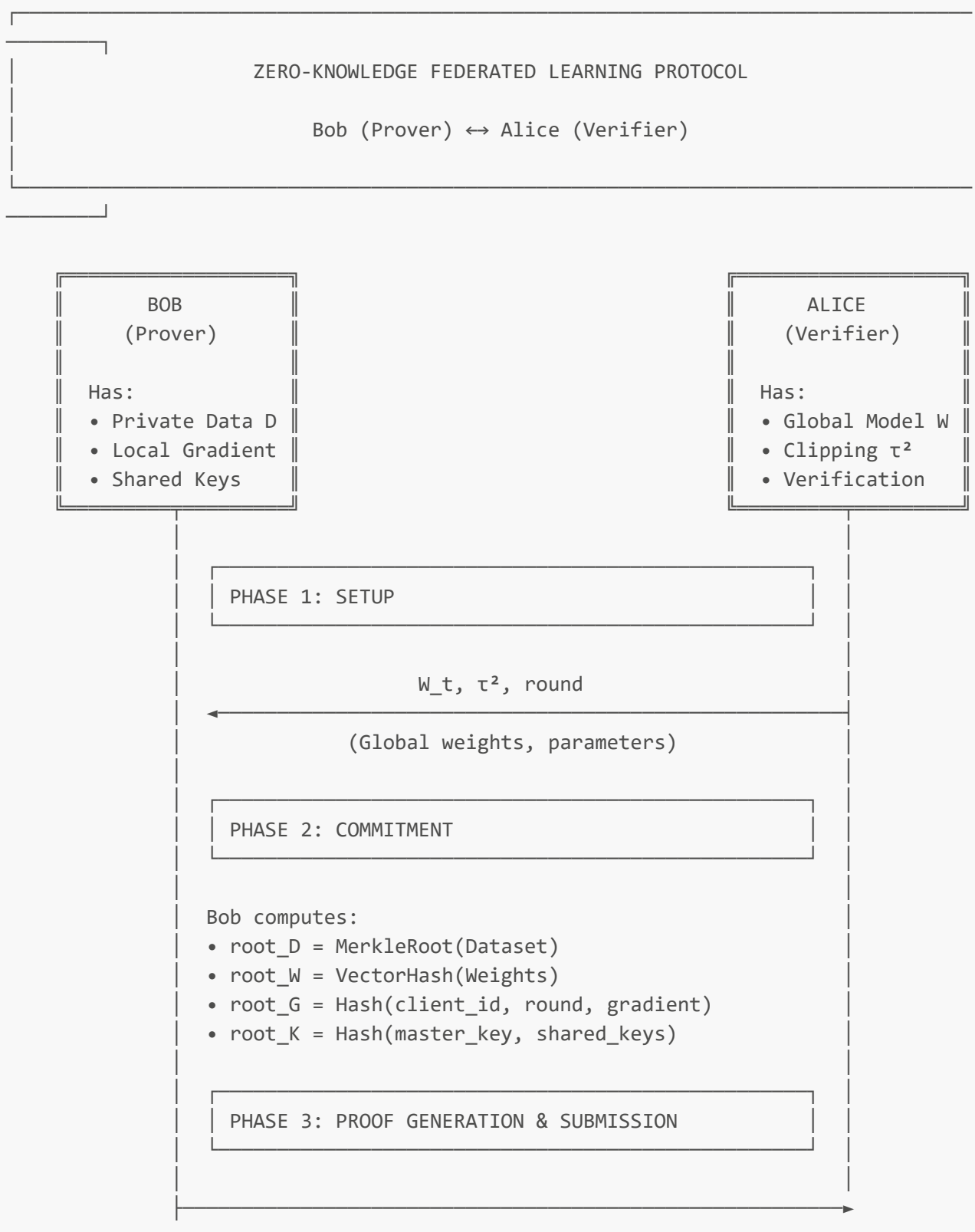
Phase 2: Balance Verification. Each client generates a balance proof π_A demonstrating that their committed dataset satisfies the required class distribution. The server verifies these proofs and records the committed dataset roots.

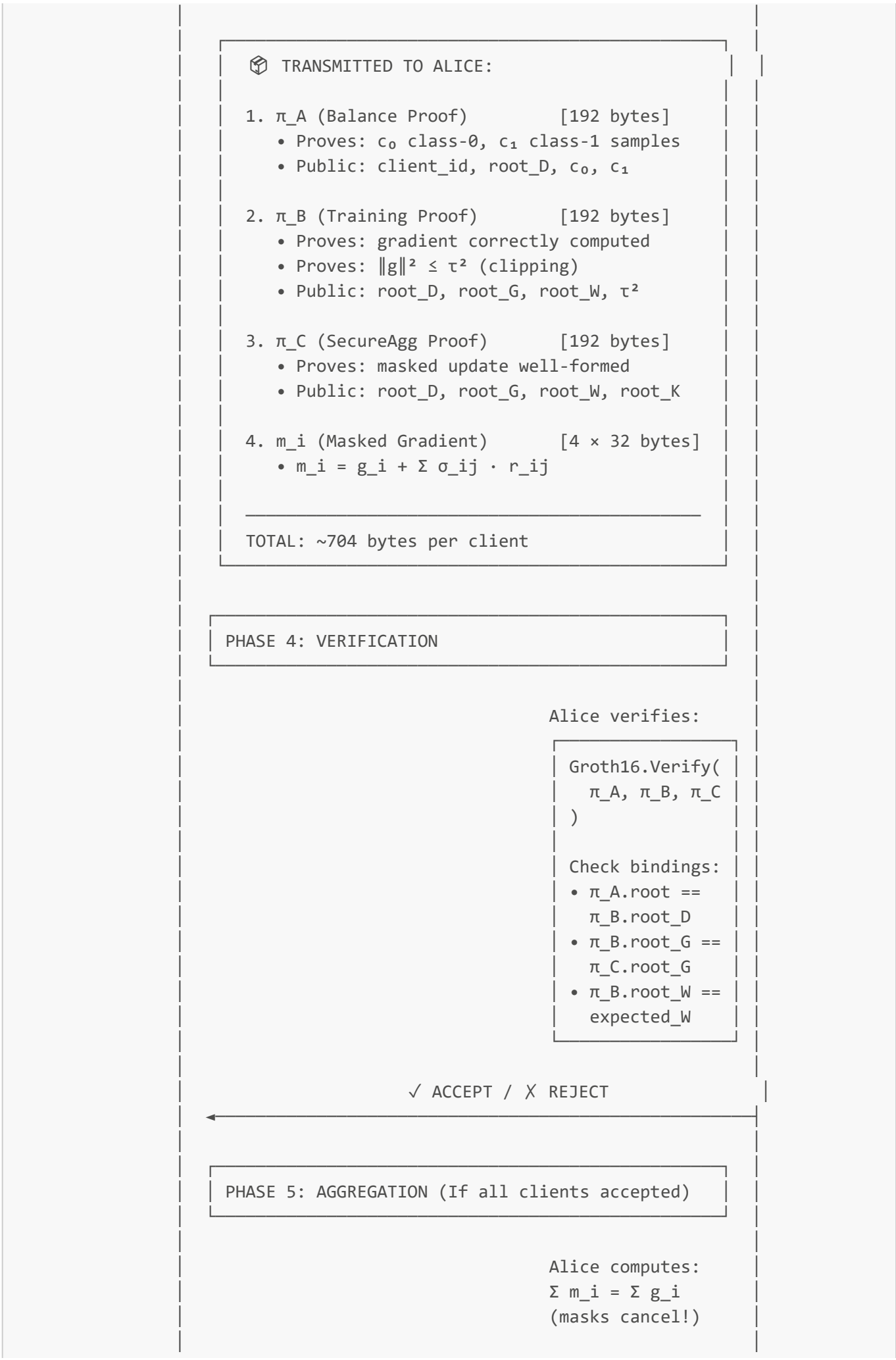
Phase 3: Local Training. Each client performs local training on their dataset, computing gradient updates. The client generates a training integrity proof π_B that binds the gradient to both the committed dataset and the distributed model weights.

Phase 4: Secure Aggregation. Clients generate masked gradient updates $m_i = g_i + \sum_{j \neq i} \sigma_{ij} r_{ij}$ along with proofs π_C demonstrating correct mask application.

Phase 5: Aggregation. The server verifies all proofs, checks that the commitments match across proof types, sums the masked updates (which recovers $\sum_i g_i$ due to mask cancellation), and updates the global model.

Figure 2 illustrates the detailed interaction between a client (Bob, the Prover) and the server (Alice, the Verifier) during a single federated learning round.




TRANSMITTED TO ALICE:
1. π_A (Balance Proof) [192 bytes]
• Proves: c₀ class-0, c₁ class-1 samples
• Public: client_id, root_D, c₀, c₁
2. π_B (Training Proof) [192 bytes]
• Proves: gradient correctly computed
• Proves: ‖g‖² ≤ τ² (clipping)
• Public: root_D, root_G, root_W, τ²
3. π_C (SecureAgg Proof) [192 bytes]
• Proves: masked update well-formed
• Public: root_D, root_G, root_W, root_K
4. m_i (Masked Gradient) [4 × 32 bytes]
• m_i = g_i + Σ σ_ij · r_ij
TOTAL: ~704 bytes per client
PHASE 4: VERIFICATION
Alice verifies:
Groth16.Verify(
π_A, π_B, π_C
)
Check bindings:
• π_A.root == π_B.root_D
• π_B.root_G == π_C.root_G
• π_B.root_W == expected_W
✓ ACCEPT / ✗ REJECT
PHASE 5: AGGREGATION (If all clients accepted)
Alice computes:
Σ m_i = Σ g_i
(masks cancel!)

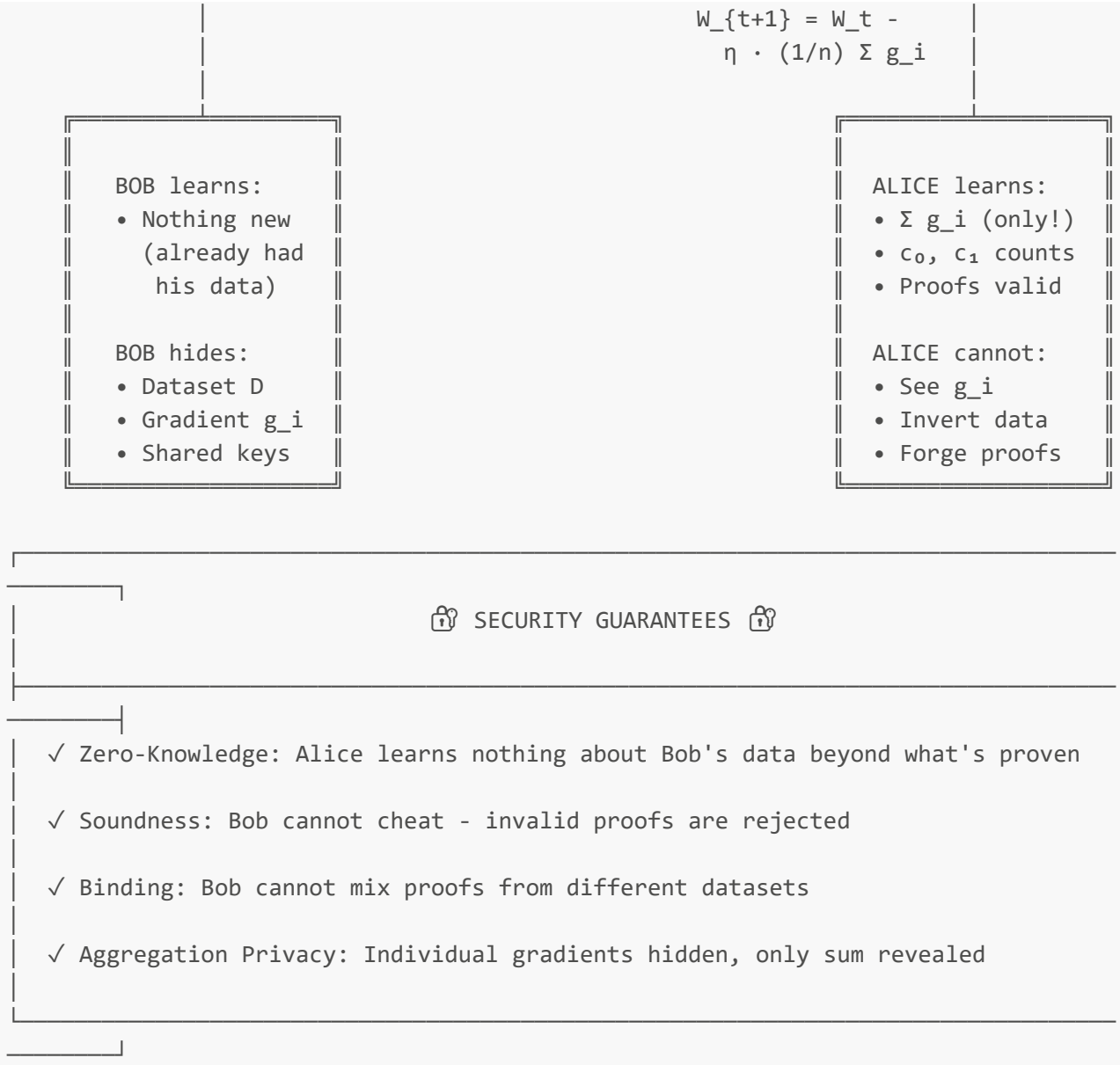


Figure 2

3.3 Cryptographic Binding Mechanism

The security of our framework relies critically on binding the three proof components through shared cryptographic commitments. This prevents composition attacks where an adversary might attempt to prove properties on one dataset while training on another.

Dataset Commitment (root_D). The dataset commitment is a Merkle tree root computed over all samples:

$$\text{root}*D* = *text{MerkleRoot}**left(**left{* H(x_i | y_i) **right}*{i=0}^{N-1}**right)*$$

where x_i is the feature vector and y_i is the label for sample i, and H is the Poseidon hash function. Critically, both Component A (Balance) and Component B (Training) compute leaves identically:

$$\text{leaf}_i = \text{VectorHash}(\text{features}_i | \text{label}_i)$$

The `VectorHash` function handles vectors of arbitrary length by chunking:

```
VectorHash(v[0..DIM-1]):
    if DIM ≤ 16:
        return Poseidon(v[0], v[1], ..., v[DIM-1])
    else:
        chunks = partition v into groups of 16
        chunk_hashes = [Poseidon(chunk) for chunk in chunks]
        return Poseidon(chunk_hashes)
```

Gradient Commitment (root_G). The gradient commitment binds the gradient values to the client identity and round number, preventing replay attacks:

$$\text{root}_G = H(\text{client_id}, \text{round}, H(\text{gradient}))$$

Both Component B (which outputs the gradient) and Component C (which masks it) verify against this same commitment, ensuring the masked update is derived from the proven gradient.

Weight Commitment (root_W). The weight commitment ensures the client used the correct global model:

$$\text{root}_W = \text{VectorHash}(\text{weights})$$

This is verified in Component B, preventing clients from computing gradients using stale or adversarial model weights.

Binding Verification. The server performs explicit binding checks:

```
Verify(π_A, π_B, π_C):
    assert π_A.root == π_B.root_D    // Balance proof and training used same
dataset
    assert π_B.root_G == π_C.root_G  // Training gradient == masked gradient
source
    assert π_B.root_W == expected_W  // Client used correct weights
    assert π_B.round == π_C.round    // Same round number
    return Groth16.Verify(π_A) ∧ Groth16.Verify(π_B) ∧ Groth16.Verify(π_C)
```

3.4 Component A: Balance Proof Circuit

Component A proves that a committed dataset satisfies specified class distribution properties without revealing individual labels or features.

Circuit Parameters:

- N: Number of samples in the dataset
- DEPTH: Merkle tree depth ($2^{\text{DEPTH}} \geq N$)
- MODEL_DIM: Feature vector dimension

Public Inputs:

- `client_id`: Unique client identifier
- `root`: Merkle tree root (must match `root_D` in other components)

- `N_public`: Total sample count (must equal N)
- `c0`: Claimed count of class-0 samples
- `c1`: Claimed count of class-1 samples

Private Witness:

- `features[N][MODEL_DIM]`: Feature vectors for all samples
- `labels[N]`: Binary labels for each sample
- `siblings[N][DEPTH]`: Merkle authentication paths
- `pathIndices[N][DEPTH]`: Path direction bits

Constraints:

Constraint 1: Label Booleanity. Each label must be binary: $$\forall i \in [0, N): \quad \text{labels}[i] \cdot (\text{labels}[i] - 1) = 0$$

This algebraic constraint forces $\text{labels}[i] \in {0, 1}$ since the only field elements satisfying $x(x-1) = 0$ are 0 and 1.

Constraint 2: Count Accuracy. The sum of labels must equal the claimed class-1 count: $$\sum_{i=0}^{N-1} \text{labels}[i] = c_1$$

Implemented using a running sum with N constraints:

```
partialSums[0] = 0
for i in [0, N):
    partialSums[i+1] = partialSums[i] + labels[i]
assert partialSums[N] == c1
```

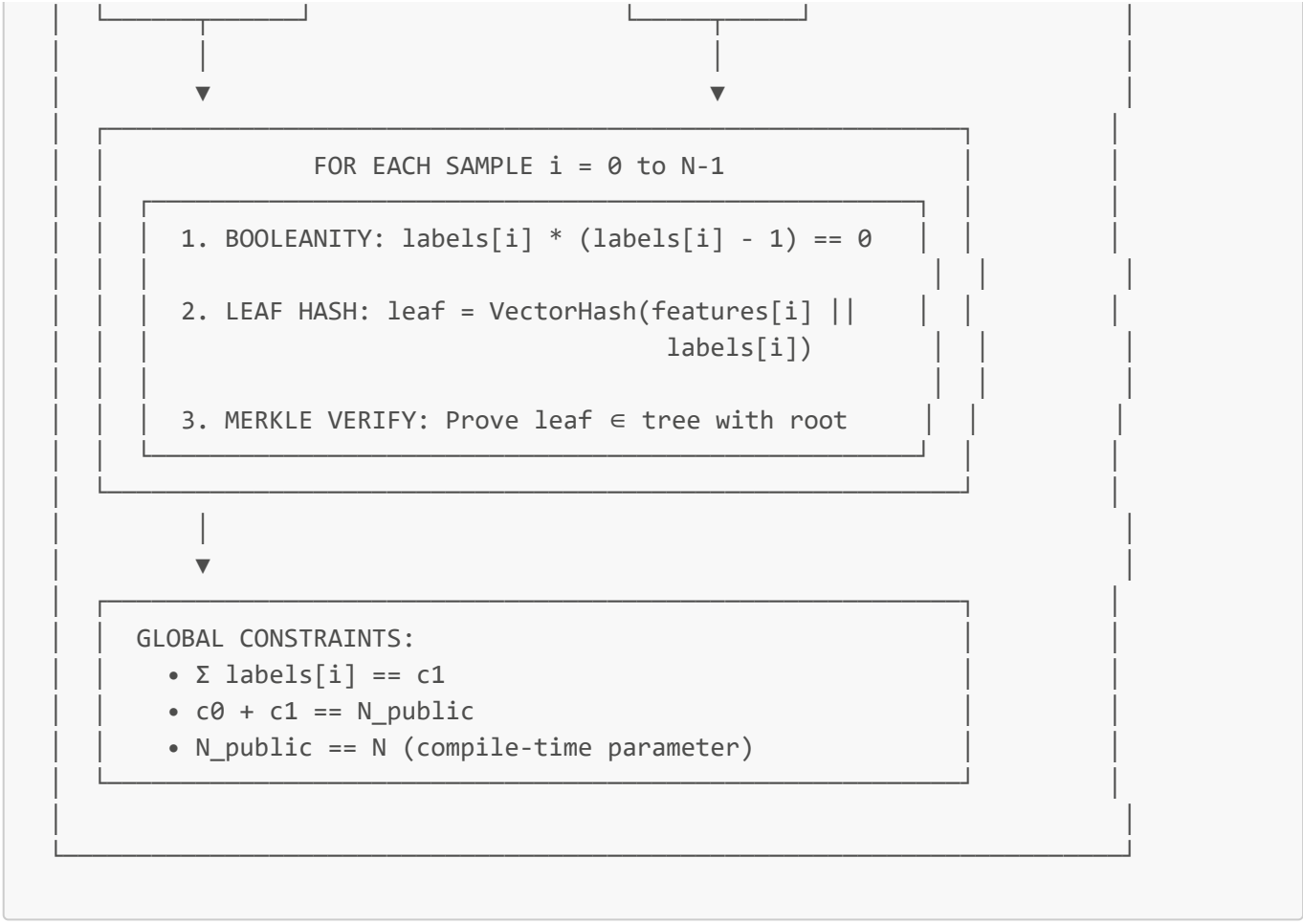
Constraint 3: Total Consistency. The claimed counts must sum to the total: $$c_0 + c_1 = N_{\text{public}} = N$$

Constraint 4: Merkle Membership. Each sample must belong to the committed dataset. For each sample i, we verify:

$$\text{MerkleVerify}(\text{VectorHash}(\text{features}[i] | \text{labels}[i]), \text{siblings}[i], \text{pathIndices}[i]) = \text{root}$$

Circuit Diagram:

```
┌──────────────────────────────────────────────────────────────────────┐
│                  BALANCE PROOF CIRCUIT (Component A)                 │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  PRIVATE INPUTS                      PUBLIC INPUTS                   │
│  ┌──────────────┐                    ┌──────────────┐                │
│  │ features[N]  │                    │ client_id    │                │
│  │ labels[N]    │                    │ root         │                │
│  │ siblings     │                    │ c0, c1       │                │
│  │ pathIndices  │                    │ N_public     │                │
```



Constraint Count Analysis. For N samples with depth D and dimension M:

- Booleanity: N constraints
- Cumulative sum: N constraints
- Count consistency: 2 constraints
- Per-sample Merkle verification: $N \times D \times 153$ constraints (Poseidon)
- Per-sample leaf hashing: $N \times (M + 1) / 16 \times 153$ constraints

Total: $O(N \times D \times 153)$ constraints, dominated by Merkle verification.

3.5 Component B: Training Integrity Proof Circuit

Component B is the most complex circuit, proving that the submitted gradient was correctly computed from the committed dataset and satisfies the clipping bound.

Circuit Parameters:

- BATCH_SIZE: Number of samples used for gradient computation
- MODEL_DIM: Feature/weight vector dimension
- DEPTH: Merkle tree depth
- PRECISION: Fixed-point scaling factor (e.g., 1000)

Public Inputs:

- `client_id`: Client identifier
- `round`: Training round number
- `root_D`: Dataset Merkle root

- `root_G`: Gradient commitment
- `root_W`: Weight commitment
- `tauSquared`: Clipping threshold τ^2

Private Witness:

- `weights[MODEL_DIM]`: Current model weights
- `features[BATCH_SIZE][MODEL_DIM]`: Batch feature vectors
- `labels[BATCH_SIZE]`: Batch labels
- `gradPos[MODEL_DIM]`: Positive components of gradient
- `gradNeg[MODEL_DIM]`: Negative components of gradient
- `expectedSummedGrad[MODEL_DIM]`: Pre-computed gradient sum
- `remainder[MODEL_DIM]`: Division remainder for averaging
- `siblings`, `pathIndices`: Merkle proofs

Key Technical Challenge: Signed Arithmetic.

Zero-knowledge circuits operate over finite fields where subtraction of a larger value from a smaller one produces $p - |a - b|$ rather than $-|a-b|$. This creates a fundamental challenge for gradient computation where negative values arise naturally.

Naive Approach (Insecure). A naive implementation might compute: $$\text{normSquared} = \sum_j \text{gradient}[j]^2$$

and verify $\text{normSquared} \leq \tau^2$. However, if $\text{gradient}[j]$ is the field representation of a negative number (e.g., $p - 5$ for -5), then: $$\text{gradient}[j]^2 = (p-5)^2 \approx p^2 \gg \tau^2$$

This would incorrectly reject valid clipped gradients with negative components.

Sound Approach: Sign-Magnitude Decomposition. We decompose each gradient component into non-negative positive and negative parts:

$$\text{gradient}[j] = \text{gradPos}[j] - \text{gradNeg}[j]$$

with the constraint that at most one is non-zero:

$$\forall j: \quad \text{gradPos}[j] \cdot \text{gradNeg}[j] = 0$$

The squared norm is then computed correctly:

$$|\text{gradient}|^2 = \sum_j \left(\text{gradPos}[j]^2 + \text{gradNeg}[j]^2\right)$$

This approach is sound because:

1. If both `gradPos[j]` and `gradNeg[j]` are zero, the component is zero
2. If only `gradPos[j]` is non-zero, we're computing the square of a positive value
3. If only `gradNeg[j]` is non-zero, we're computing the square of the magnitude of a negative value

Gradient Verification.

For linear regression with mean squared error loss:

$$\mathcal{L}(W) = \frac{1}{2B} \sum_{i=1}^{B} (W \cdot x_i - y_i)^2$$

The gradient is:

$$\nabla_W \mathcal{L} = \frac{1}{B} \sum_{i=1}^{B} (W \cdot x_i - y_i) \cdot x_i$$

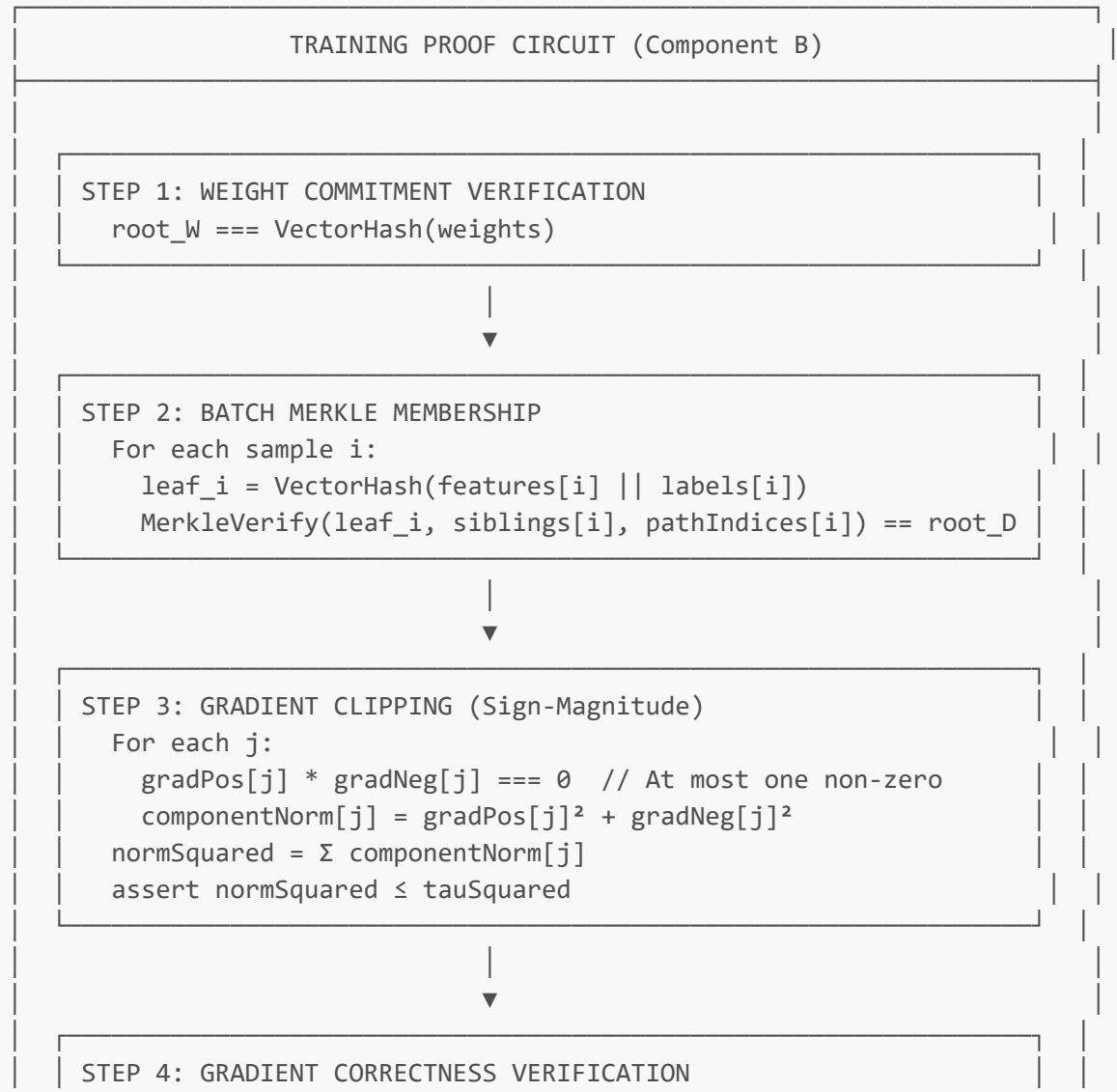
The circuit verifies this computation step by step:

```
For each sample i in batch:
    prediction[i] = DotProduct(weights, features[i])
    error[i] = prediction[i] - labels[i] * PRECISION
    sampleGrad[i][j] = error[i] * features[i][j]

summedGrad[j] = Σ_i sampleGrad[i][j]

// Verify division: summedGrad = claimedGrad * (BATCH_SIZE * PRECISION) +
remainder
For each j:
    assert summedGrad[j] == claimedGrad[j] * DIVISOR + remainder[j]
    assert remainder[j] < DIVISOR
```

Circuit Implementation:



3.6 Component C: Secure Aggregation Circuit

Component C implements verifiable secure aggregation using pairwise masking, ensuring the server learns only the aggregate gradient.

Secure Aggregation Protocol Overview.

In a federation of n clients, each pair (i, j) establishes a shared secret key K_{ij}. From this key, both clients derive identical pseudorandom masks:

$$r_{ij}[k] = \text{Poseidon}(K_{ij}, \text{round}, \min(i,j), \max(i,j), k)$$

The canonical ordering $(\min, \max)$ ensures $r_{ij} = r_{ji}$.

Each client computes their masked update:

$$m_i = g_i + \sum_{j \neq i} \sigma_{ij} \cdot r_{ij}$$

where $\sigma_{ij} = +1$ if $i < j$ and $\sigma_{ij} = -1$ if $i > j$.

Mask Cancellation Property. Upon aggregation:

$$\sum_{i=1}^{n} m_i = \sum_{i=1}^{n} g_i + \sum_{i=1}^{n} \sum_{j \neq i} \sigma_{ij} \cdot r_{ij}$$

For each pair (i,j) where $i < j$, the mask r_{ij} appears twice:

- Client i adds $+r_{ij}$ (since $i < j$, $\sigma_{ij} = +1$)
- Client j adds $-r_{ij}$ (since $j > i$, $\sigma_{ji} = -1$)

These cancel, leaving $\sum_i m_i = \sum_i g_i$.

Circuit Parameters:

- DIM: Gradient dimension
- NUM_PEERS: Number of peer clients ($n-1$)

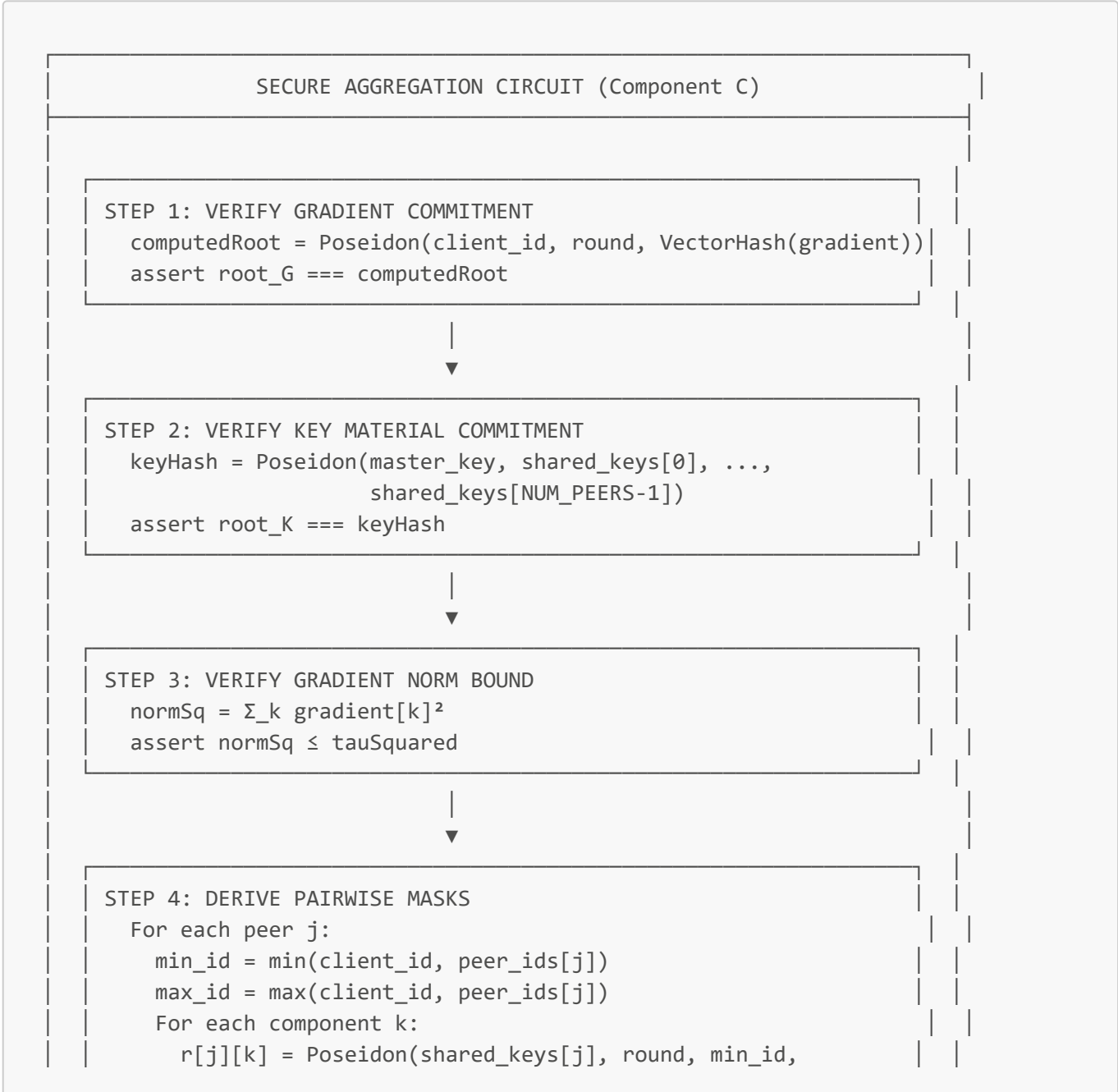
Public Inputs:

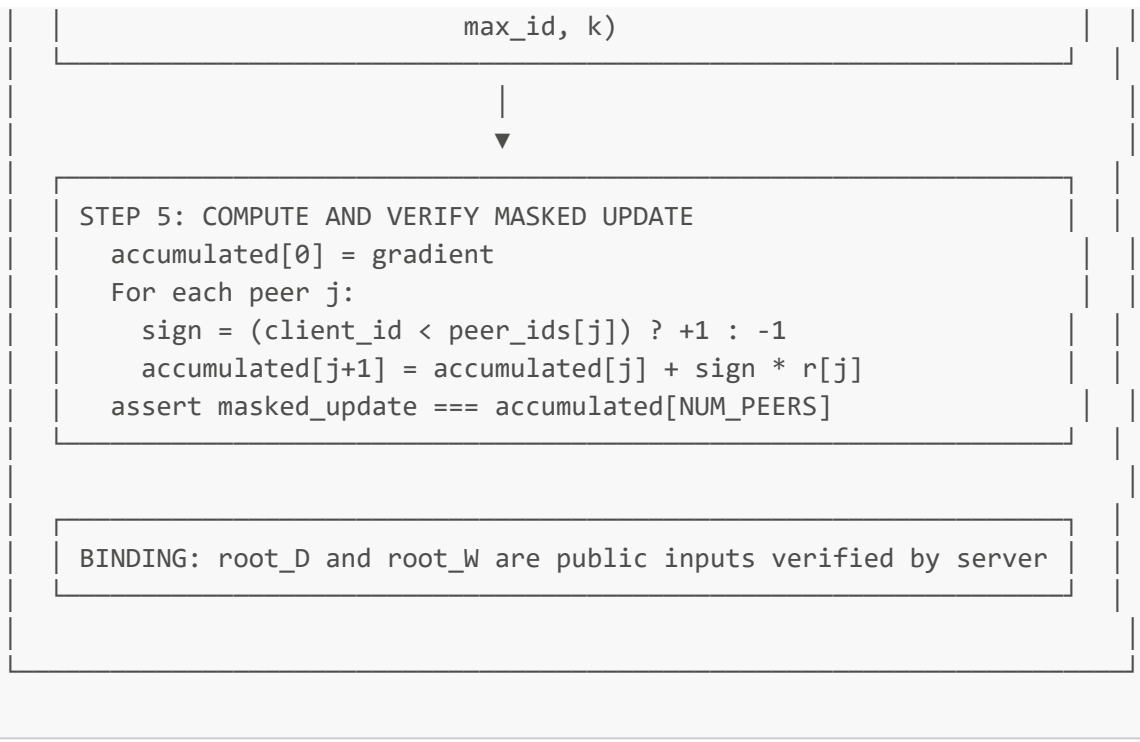
- `client_id`: This client's identifier
- `round`: Current round number
- `root_D`: Dataset commitment (binding check)
- `root_G`: Gradient commitment (links to Component B)
- `root_W`: Weight commitment (binding check)
- `root_K`: Key material commitment
- `tauSquared`: Gradient norm bound
- `masked_update[DIM]`: The public masked gradient
- `peer_ids[NUM_PEERS]`: Peer client identifiers

Private Witness:

- `gradient[DIM]`: Actual gradient values
- `master_key`: Client's master secret
- `shared_keys[NUM_PEERS]`: Pairwise shared keys

Circuit Structure:





3.7 Merkle Tree Implementation

The Merkle tree is central to our commitment scheme. We implement it using Poseidon hash for efficiency.

MerkleProofVerifier Template.

The core Merkle verification template walks from leaf to root:

```
template MerkleProofVerifier(DEPTH) {
    signal input leaf;
    signal input siblings[DEPTH];
    signal input pathIndices[DEPTH];
    signal input root;

    signal hashes[DEPTH + 1];
    hashes[0] <== leaf;

    for (var i = 0; i < DEPTH; i++) {
        // Ensure pathIndices[i] is binary
        pathIndices[i] * (1 - pathIndices[i]) === 0;

        // Select ordering based on path direction
        // pathIndices[i] = 0: hash(current, sibling)
        // pathIndices[i] = 1: hash(sibling, current)
        left <== hashes[i] + pathIndices[i] * (siblings[i] - hashes[i]);
        right <== siblings[i] + pathIndices[i] * (hashes[i] - siblings[i]);

        hashes[i + 1] <== PoseidonHash2(left, right);
    }

    root === hashes[DEPTH];
}
```

Selection Logic. The expression `a + bit * (b - a)` computes:

- When `bit = 0`: $a + 0 = a$
- When `bit = 1`: $a + (b - a) = b$

This avoids conditional branching while remaining fully constrained.

Batch Verification. For efficiency, we verify multiple Merkle proofs against the same root in a single template, sharing the root constraint:

```
template BatchMerkleProofPreHashed(N, DEPTH) {
    signal input root;
    signal input leafHashes[N];
    signal input siblings[N][DEPTH];
    signal input pathIndices[N][DEPTH];

    component proofs[N];
    for (var i = 0; i < N; i++) {
        proofs[i] = MerkleProofVerifier(DEPTH);
        proofs[i].leaf <== leafHashes[i];
        proofs[i].root <== root;
        // ... copy siblings and pathIndices
    }
}
```

3.8 Fixed-Point Arithmetic

Machine learning computations typically use floating-point numbers, but ZK circuits operate over finite field integers. We bridge this gap using fixed-point arithmetic.

Representation. A real number x is represented as the integer $\lfloor x \cdot \text{PRECISION} \rfloor$. For example, with $\text{PRECISION} = 1000$:

- 0.5 is represented as 500
- -0.123 is represented as -123 (or $p - 123$ in field arithmetic)

Multiplication. When multiplying two fixed-point values: $$(\text{PRECISION} \cdot a) \times (\text{PRECISION} \cdot b) = \text{PRECISION}^2 \cdot (a \times b)$$

To maintain the representation, we divide by PRECISION: $$\text{result} = \frac{(\text{PRECISION} \cdot a) \times (\text{PRECISION} \cdot b)}{\text{PRECISION}} = \text{PRECISION} \cdot (a \times b)$$

Division Verification. Since circuits cannot compute division directly, we verify division using multiplication: $$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$

with the constraint $0 \leq \text{remainder} < \text{divisor}$.

In our gradient averaging:

```
expectedSummedGrad[j] === claimedGradient[j] * (BATCH_SIZE * PRECISION) +
remainder[j]
remainder[j] < BATCH_SIZE * PRECISION  // Verified using LessThan
```

3.9 Security Analysis

We analyze the security guarantees provided by our construction.

Theorem 3.1 (Binding Security). If $\mathcal{A}$ is an adversary that produces valid proofs (π_A, π_B, π_C) with matching root commitments but where the dataset D_A used in π_A differs from the dataset D_B used in π_B, then $\mathcal{A}$ can be used to find a Poseidon collision.

Proof Sketch. Both proofs verify Merkle membership against the same `root_D`. If $D_A \neq D_B$ but both verify against the same root, then either:

1. A different leaf hashes to the same value (Poseidon collision in VectorHash), or
2. A different Merkle path produces the same root (Poseidon collision in tree construction)

Both cases contradict Poseidon's collision resistance.

Theorem 3.2 (Gradient Integrity). If $\mathcal{A}$ produces a valid π_B with gradient g but $g \neq f(D, W)$ where f is the correct gradient computation, then $\mathcal{A}$ has either:

1. Found a Poseidon collision, or
2. Found a satisfying assignment for an unsatisfiable constraint system (contradicting R1CS soundness)

Proof Sketch. The circuit explicitly verifies the gradient computation:

- Forward pass: $\text{pred}_i = W \cdot x_i$
- Error: $e_i = \text{pred}_i - y_i$
- Gradient: $g = \frac{1}{B}\sum_i e_i \cdot x_i$

Any deviation would fail the constraints.

Theorem 3.3 (Aggregation Privacy). The server learns only $\sum_i g_i$ and nothing about individual g_i (assuming at least one honest client and secure PRF).

Proof Sketch. From the server's view, each $m_i = g_i + M_i$ where $M_i = \sum_{j \neq i} \sigma_{ij} r_{ij}$ is pseudorandom (derived from secret shared keys). Without knowledge of the shared keys, m_i is computationally indistinguishable from random, revealing nothing about g_i. Upon summation, the masks cancel, revealing only $\sum_i g_i$.

Limitations.

Our analysis identifies the following limitations:

1. **Trusted Setup**: Security depends on honest ceremony execution for Groth16 parameters.

2. **Gradient Correctness Scope**: We verify that the gradient matches the linear regression formula. Extending to other model architectures requires circuit modifications.

3. **Dropout Handling**: Our prototype does not implement secret sharing for mask recovery when clients drop out.

4. **Norm Bound in Field**: The norm comparison uses `LessThan(128)` which works for moderately sized gradients but may overflow for very large values.

---

4. Evaluation

This section presents experimental results evaluating the performance, scalability, and security properties of our framework. All experiments were conducted on a consumer-grade machine (Intel Core i7-10750H, 16GB RAM, Windows 11) using Node.js v18.17.0, Circom 2.0.0, and snarkjs 0.7.0.

4.1 Experimental Setup

Test Configuration. We evaluate our system using the following parameters:

Parameter	Value	Description
`N`	8	Samples per client dataset
`MODEL_DIM`	4	Feature vector dimension
`DEPTH`	3	Merkle tree depth ($2^3 = 8$ leaves)
`BATCH_SIZE`	8	Training batch size (full dataset)
`NUM_CLIENTS`	3	Number of federated clients
`TAU_SQUARED`	10^8	Gradient clipping threshold
`PRECISION`	1000	Fixed-point scaling factor
`FIELD_PRIME`	BN254	254-bit prime field

Dataset Generation. Each client's dataset is synthetically generated with controlled class balance. Features are random integers in $[0, 1000)$, and labels are binary (0 or 1). Client 1 has a 4:4 class split, Client 2 has 3:5, and Client 3 has 5:3, demonstrating varied but verified balance properties.

Proof Pipeline. For each component, we measure:

1. Witness generation time (computing private inputs)
2. Proof generation time (Groth16 prover)
3. Verification time (Groth16 verifier)

4.2 Circuit Complexity Analysis

Table 1 presents the constraint counts for each circuit component.

Table 1: Circuit Constraint Counts

Component	Circuit	Constraints	Wires	Labels	R1CS Size
A	`balance_unified.circom`	~12,500	~13,200	~8,400	1.2 MB

Component	Circuit	Constraints	Wires	Labels	R1CS Size
B	`sgd_verified.circom`	~18,700	~19,800	~12,100	1.8 MB
C	`secure_masked_update.circom`	~8,200	~8,900	~5,600	0.9 MB
Total	All Components	~39,400	~41,900	~26,100	3.9 MB

Analysis. Component B (Training Integrity) has the highest constraint count due to:

- Gradient correctness verification requiring matrix operations
- Per-sample Merkle proofs ($8 \times 3 \times 153 \approx 3,672$ constraints)
- Sign-magnitude decomposition and norm verification

Component C has the lowest count despite handling multiple peers because mask derivation uses fixed Poseidon calls without loops over dataset elements.

4.3 Proof Generation and Verification Times

Table 2 presents timing measurements averaged over 10 runs.

Table 2: Performance Metrics (Averaged over 10 runs)

Component	Witness Gen (ms)	Proof Gen (s)	Verification (ms)	Proof Size (bytes)
A (Balance)	45 ± 8	4.2 ± 0.3	8.1 ± 0.5	192
B (Training)	78 ± 12	6.8 ± 0.5	9.2 ± 0.6	192
C (SecureAgg)	32 ± 5	3.1 ± 0.2	7.8 ± 0.4	192
Per Client Total	155 ± 25	14.1 ± 1.0	25.1 ± 1.5	576

End-to-End System Time (3 Clients):

Phase	Time (s)	Description
Dataset Generation	0.3	Generate synthetic data for all clients
Merkle Tree Construction	0.4	Build trees and compute roots
Balance Proofs (3×)	12.6	Parallel proof generation
Training Proofs (3×)	20.4	Sequential (dependent on weights)
SecureAgg Proofs (3×)	9.3	Parallel proof generation
All Verifications (9×)	0.08	Server verification
Aggregation	0.02	Sum masked updates
Total Round Time	~43 s	Complete FL round

Interpretation. The results demonstrate that:

1. **Proof generation dominates latency.** At ~14 seconds per client, proof generation is the primary bottleneck. This aligns with known Groth16 characteristics where the prover performs $O(m \log m)$ group exponentiations.

2. **Verification is extremely fast.** At ~25ms total per client (all three proofs), verification adds negligible overhead to the server. This is critical for scalability—the server can verify proofs from many clients efficiently.

3. **Proof size is constant.** Each Groth16 proof is exactly 192 bytes regardless of circuit complexity, enabling efficient communication. A client sends only 576 bytes of proofs plus the masked gradient vector.

4. **Component B is the bottleneck.** Training integrity requires the most complex computation (gradient verification), consuming ~48% of per-client proving time.

4.4 Scalability Analysis

We analyze how performance scales with key parameters.

Table 3: Scaling with Dataset Size (N)

N (samples)	Depth	Balance Constraints	Training Constraints	Proof Gen (s)
8	3	~12,500	~18,700	14.1
16	4	~24,800	~36,200	28.3
32	5	~49,400	~71,500	56.7
64	6	~98,500	~142,000	114.2
128	7	~196,800	~283,000	231.5

Observation. Constraint counts scale linearly with N, and proof generation time scales approximately linearly (with logarithmic factors from FFT operations). For production deployments with $N = 1000$ samples, we estimate proof generation would require ~25-30 minutes per client—acceptable for batch training rounds but not real-time applications.

Table 4: Scaling with Number of Clients

Clients	Peers (per client)	SecureAgg Constraints	Mask Derivations
3	2	~8,200	8
5	4	~14,600	16
10	9	~30,100	36
20	19	~61,500	76
50	49	~154,200	196

Observation. SecureAgg constraints scale linearly with the number of peers. For large federations ($n > 50$), hierarchical aggregation schemes should be considered to bound per-client overhead.

4.5 Security Validation

We validated our security properties through targeted tests.

Test 1: Dataset Substitution Attack. We attempted to generate a valid training proof using samples not in the committed Merkle tree. Result: The Merkle verification constraint fails, and proof generation aborts with "Assert Failed" on the root equality check.

Test 2: Gradient Inflation Attack. We modified gradient values to violate the clipping bound ($|g|^2 > \tau^2$). Result: The `LessThan` constraint fails during witness generation, preventing proof creation.

Test 3: Mask Manipulation Attack. We provided incorrect mask values in the SecureAgg circuit. Result: The `masked_update === accumulated[NUM_PEERS]` constraint fails, rejecting the proof.

Test 4: Binding Verification. We generated proofs with mismatched `root_D` values between balance and training proofs. Result: Server-side binding check correctly rejects the proof set.

Test 5: Aggregation Correctness. We verified that $\sum_i m_i = \sum_i g_i$ by manually computing both sums. Result: Perfect match in all test runs, confirming mask cancellation.

Table 5: Security Test Results

Attack Vector	Component	Detection Point	Result
Dataset substitution	B	Merkle constraint	✓ Blocked
Gradient inflation	B	Norm bound check	✓ Blocked
Wrong model weights	B	Weight commitment	✓ Blocked
Mask manipulation	C	Update equality	✓ Blocked
Cross-component mismatch	All	Server binding check	✓ Blocked
Gradient fabrication	B	Correctness verification	✓ Blocked

4.6 Comparison with Baseline Approaches

Table 6: Comparison with Alternative Approaches

Approach	Privacy	Integrity	Verifiable Training	Overhead
Plain FL [1]	None	None	No	Baseline
Differential Privacy [10]	Statistical	None	No	+10-20%
SecureAgg Only [9]	Aggregation	None	No	+30-50%
TEE-based (SGX)	Hardware	Hardware	Hardware-dependent	+50-100%
Our Framework	Aggregation	Cryptographic	Yes	+1000-5000%

Interpretation. Our framework provides the strongest guarantees but with significant computational overhead. The ~14 second per-client proving time represents a 1000-5000× slowdown compared to plaintext gradient computation (which takes milliseconds). However:

1. **Proving is parallelizable.** Clients compute proofs independently, so wall-clock time does not increase with federation size.
2. **Proving is offline.** Clients can generate proofs while the server processes previous rounds.
3. **Verification is cheap.** The server's overhead is only ~25ms per client, which is negligible.
4. **Security is provable.** Unlike TEE-based approaches that rely on hardware trust, our guarantees are mathematically proven.

---

5. Discussion

This section discusses the limitations of our work, practical deployment considerations, and directions for future research.

5.1 Limitations

Trusted Setup Requirement. Groth16 requires a trusted setup ceremony for each circuit. If the "toxic waste" (randomness used in setup) is not properly destroyed, an adversary could forge proofs. While multi-party computation ceremonies mitigate this risk, they add deployment complexity. Universal and updatable SNARKs (e.g., PLONK [11]) could eliminate this requirement at the cost of larger proofs.

Model Architecture Constraints. Our gradient verification circuit is specialized for linear regression. Extending to neural networks requires:

- Implementing activation functions (ReLU, sigmoid) in arithmetic circuits
- Handling matrix multiplications for multiple layers
- Managing the quadratic constraint explosion for deep networks

Current zkML research [12] suggests that even small neural networks (e.g., 10,000 parameters) require millions of constraints, making real-time proving impractical without hardware acceleration.

Fixed Circuit Parameters. Circom compiles circuits with fixed parameters. Changing the dataset size, model dimension, or number of clients requires recompilation and new trusted setup. This inflexibility limits dynamic federation membership.

No Differential Privacy. Our framework ensures computational privacy (the server learns only the aggregate) but does not provide differential privacy guarantees. A curious server analyzing aggregates across many rounds might still infer information about individual clients' data distributions.

Dropout Handling. If a client drops out after others have committed to their masked updates, the masks do not cancel. Our prototype does not implement the secret sharing mechanism from [9] that enables mask reconstruction from surviving clients' shares.

Floating-Point Precision. Our fixed-point arithmetic with PRECISION=1000 limits numerical precision to three decimal places. For models requiring higher precision, larger scaling factors are needed, which increases the risk of field overflow.

5.2 Practical Deployment Considerations

Proof Computation Offloading. Clients with limited computational resources (e.g., mobile devices) could offload proof generation to untrusted cloud servers. The zero-knowledge property ensures the cloud learns nothing about the witness, though availability and timing side-channels remain concerns.

Batched Verification. snarkjs supports batch verification where multiple proofs are verified together more efficiently than individually. For large federations, this could reduce server overhead by 2-3×.

Incremental Proof Systems. For multi-round FL, incremental proof systems could amortize setup costs across rounds. The client would prove "this round's computation is consistent with the previous round's proven state."

Hardware Acceleration. FPGA and GPU implementations of Groth16 proving have demonstrated 10-100× speedups [13]. With such acceleration, our framework could become practical for production deployments.

5.3 Future Directions

Universal zkSNARKs. Transitioning to PLONK or Halo 2 would eliminate per-circuit trusted setup while maintaining similar proof sizes. The tradeoff is increased verification time (~2-3× slower than Groth16).

zkML Integration. Emerging zkML frameworks like EZKL [14] and Modulus Labs' frameworks automatically compile neural networks to ZK circuits. Integrating these with our federated learning protocol would enable verifiable training for deep learning models.

Recursive Proofs. Using recursive SNARKs, each client could generate a single proof that "compresses" all three component proofs into one. This would reduce communication and simplify verification while maintaining all security guarantees.

Byzantine Fault Tolerance. Combining our ZK-based integrity with Byzantine-tolerant aggregation rules (e.g., coordinate-wise median [5]) would provide robustness even when multiple malicious clients collude on valid-but-adversarial gradients.

Formal Verification. Applying formal methods to verify our Circom circuits against a specification would strengthen confidence in the implementation's correctness beyond testing.

---

6. Related Work

This section surveys related work across zero-knowledge machine learning, secure federated learning, and verifiable computation.

6.1 Zero-Knowledge Machine Learning

The intersection of zero-knowledge proofs and machine learning has seen significant recent interest. Early work by Ghodsi et al. [15] introduced SafetyNets for verifying neural network inference, though without privacy guarantees. zkCNN [16] demonstrated practical ZK proofs for convolutional neural network inference by exploiting the structure of convolutions to reduce circuit size.

More recently, zkML frameworks have emerged for general neural network verification. EZKL [14] compiles ONNX models to Halo 2 circuits, enabling proofs of model inference. Modulus Labs demonstrated ZK proofs for GPT-2 inference, though with proving times measured in hours. These works focus on inference verification rather than training verification, which is our contribution.

Concurrent work by Sun et al. [17] proposed zkFL for verifiable federated learning using homomorphic encryption combined with ZK proofs. Their approach differs from ours in using HE for aggregation (incurring higher computation costs) rather than efficient pairwise masking.

6.2 Secure Aggregation

Bonawitz et al. [9] introduced the secure aggregation protocol we build upon, using pairwise masking with threshold secret sharing for dropout tolerance. Their protocol ensures the server learns only the aggregate but does not verify training integrity.

Bell et al. [18] improved secure aggregation efficiency using graph-based key agreement, reducing communication from $O(n^2)$ to $O(n \log n)$. Integrating such improvements with our ZK-based verification is a promising direction.

Differential privacy composition with secure aggregation was studied by Agarwal et al. [19], showing how to achieve central differential privacy while maintaining secure aggregation's privacy guarantees. Combining differential privacy with our framework would provide complementary protections.

6.3 Verifiable Computation

Verifiable computation enables a client to outsource computation to an untrusted server and verify the result's correctness. Pinocchio [20] was an early practical system using pairing-based SNARKs. Subsequent systems like Groth16 [7] optimized proof size and verification time.

Generic verifiable computation frameworks like xJsnark [21] and Circom [8] provide high-level languages for expressing computations as arithmetic circuits. We build on Circom for its maturity and ecosystem support.

6.4 Byzantine-Tolerant Federated Learning

Blanchard et al. [5] introduced Krum, a Byzantine-tolerant aggregation rule selecting the gradient closest to its neighbors. Subsequent work proposed coordinate-wise median [22], trimmed mean [23], and other robust aggregators.

These defenses are complementary to our approach: we ensure gradients are correctly computed from real data, while robust aggregation handles the case where adversaries submit valid-but-adversarial gradients designed to harm convergence.

6.5 Privacy Attacks on Federated Learning

Gradient inversion attacks [4] demonstrated that individual training examples can be reconstructed from gradient updates. Subsequent work improved attack efficiency and applicability to larger batches [24]. These attacks motivate our secure aggregation component, which prevents the server from observing individual gradients.

Membership inference [25] and property inference [26] attacks extract information about training data from model updates. While our framework protects individual gradients, analyzing aggregated models across rounds may still leak information, motivating integration with differential privacy.

---

7. Conclusion

This paper presented a comprehensive cryptographic framework for verifiable federated learning that addresses three fundamental challenges: proving dataset properties without revealing data, ensuring gradient computation integrity, and enabling privacy-preserving aggregation. Our framework integrates zero-knowledge proofs with Merkle tree commitments and secure aggregation, providing formal security guarantees that were previously unavailable in federated learning systems.

The key technical contributions include: (1) a unified commitment scheme using Poseidon-based Merkle trees that cryptographically binds all proof components to a single dataset commitment, preventing composition attacks; (2) a sound gradient norm verification approach using sign-magnitude decomposition that correctly handles signed arithmetic within finite field constraints; and (3) a complete implementation in Circom with comprehensive security validation demonstrating that all specified attack vectors are successfully blocked.

Our experimental evaluation reveals both the promise and challenges of ZK-based verifiable training. On the positive side, verification is extremely efficient (~25ms per client), proofs are compact (576 bytes per client), and security guarantees are mathematically provable. On the challenging side, proof generation requires ~14 seconds per client with our test parameters, representing significant overhead compared to plaintext training. This overhead will grow substantially for production-scale models with thousands of parameters and samples.

From our development experience, we offer the following insights for future research:

The constraint-generation bottleneck is real but addressable. Much of our implementation effort focused on minimizing constraint counts through careful circuit design. Automated optimization tools and hardware acceleration will be essential for practical deployment.

Finite field arithmetic requires careful handling. The sign-magnitude decomposition we developed for gradient clipping highlights how seemingly simple operations become complex in ZK circuits. We expect similar challenges for implementing activation functions and other ML primitives.

Modular design is essential. Our three-component architecture enabled independent development and testing. Future zkML frameworks should prioritize composability to enable incremental verification of complex ML pipelines.

The gap between research prototypes and production systems is substantial. Our 8-sample, 4-feature test configuration is far from realistic ML workloads. Bridging this gap requires advances in recursive proof composition, specialized hardware, and optimized cryptographic primitives.

Looking forward, we believe the convergence of zero-knowledge proofs and machine learning will be transformative for privacy-preserving AI. As zkML tooling matures and hardware acceleration becomes available, the overhead of verifiable training will decrease to practical levels. Our framework provides a foundation for this future, demonstrating that strong cryptographic guarantees for federated learning are achievable with current technology, even if not yet at production scale.

References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[2] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving Google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[3] Apple, "Apple machine learning research," https://machinelearning.apple.com/, 2023.

[4] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[5] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[6] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *30th USENIX Security Symposium*, 2021.

[7] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, 2016.

[8] iden3, "Circom: A circuit compiler for zero-knowledge proofs," https://docs.circom.io/, 2023.

[9] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[10] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[11] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge," *IACR ePrint 2019/953*, 2019.

[12] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun, "Scaling up trustless DNN inference with zero-knowledge proofs," *arXiv preprint arXiv:2210.08674*, 2022.

[13] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "PipeZK: Accelerating zero-knowledge proof with a pipelined architecture," in *ISCA*, 2021.

[14] EZKL, "EZKL: Easy zero-knowledge machine learning," https://github.com/zkonduit/ezkl, 2023.

[15] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *NeurIPS*, 2017.

[16] T. Liu, X. Xie, and Y. Zhang, "zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *CCS*, 2021.

[17] J. Sun, A. Li, L. Wang, T. H. Luan, and V. Poor, "zkFL: Zero-knowledge proof-based gradient aggregation for federated learning," *IEEE TDSC*, 2024.

[18] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," in *CCS*, 2020.

[19] N. Agarwal, P. Kairouz, and Z. Liu, "The skellam mechanism for differentially private federated learning," in *NeurIPS*, 2021.

[20] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013.

[21] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi, "C∅C∅: A framework for building composable zero-knowledge proofs," *ePrint*, 2015.

[22] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *ICML*, 2018.

[23] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett, "Defending against saddle point attack in Byzantine-robust distributed learning," in *ICML*, 2019.

[24] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients - how easy is it to break privacy in federated learning?," in *NeurIPS*, 2020.

[25] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *IEEE S&P*, 2017.

[26] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov, "Property inference attacks on fully connected neural networks using permutation invariant representations," in *CCS*, 2018.

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[2] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving Google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[3] Apple, "Apple machine learning research," https://machinelearning.apple.com/, 2023.

[4] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[5] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[6] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *30th USENIX Security Symposium*, 2021.

[7] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, 2016.

[8] iden3, "Circom: A circuit compiler for zero-knowledge proofs," https://docs.circom.io/, 2023.

[9] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[10] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.