Verifiable Training with Zero-Knowledge Proofs: Technical Report

A Cryptographic Framework for Privacy-Preserving Federated Learning

Tarek Salama, Zeyad Elshafey, Ahmed Elbehiry
Purdue University
Applied Cryptography, Fall 2025

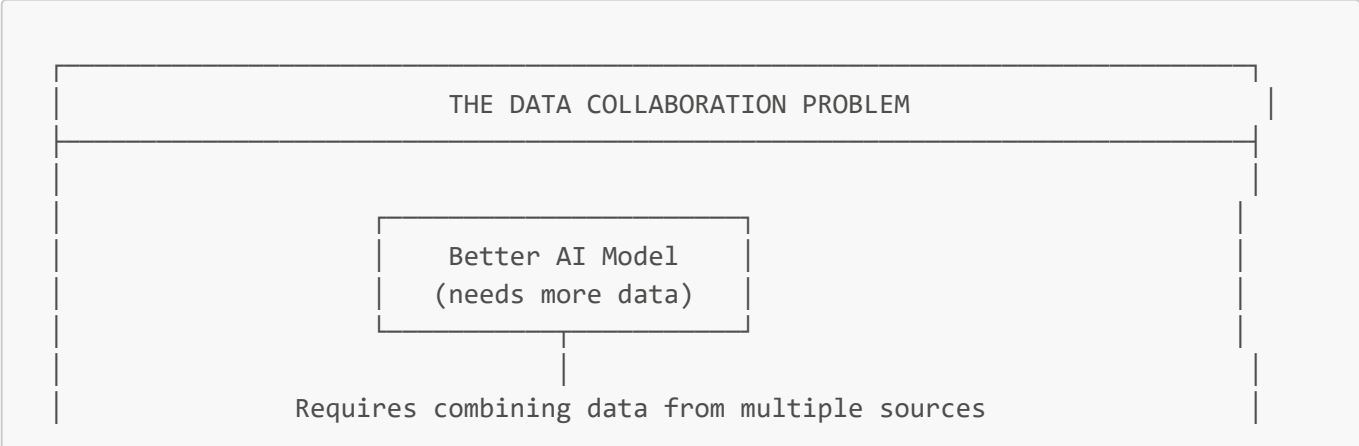
Table of Contents

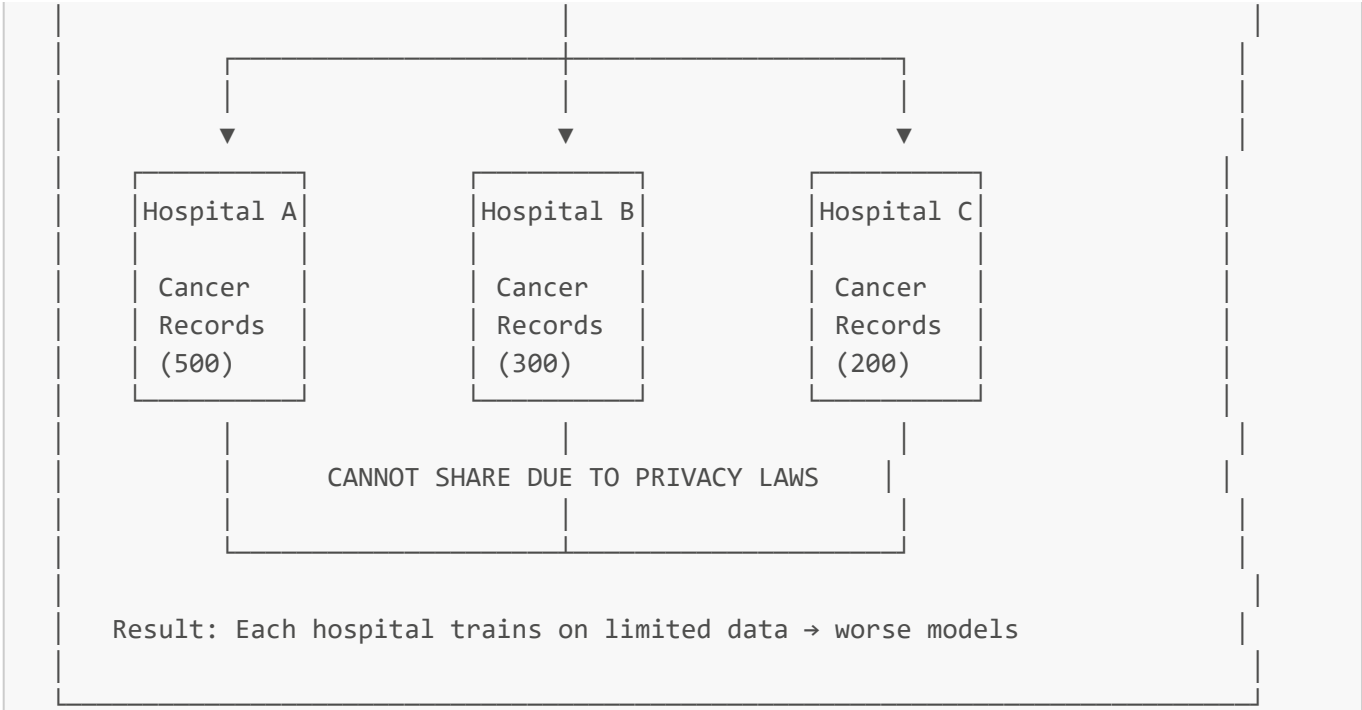
1. Introduction

1.1 The Privacy Problem in Machine Learning

Modern machine learning requires vast amounts of data to train accurate models. In many domains—healthcare, finance, government—this data is highly sensitive and protected by regulations such as HIPAA (healthcare), GDPR (Europe), and CCPA (California).

The fundamental tension: Organizations want to collaborate on AI models but cannot share their raw data.

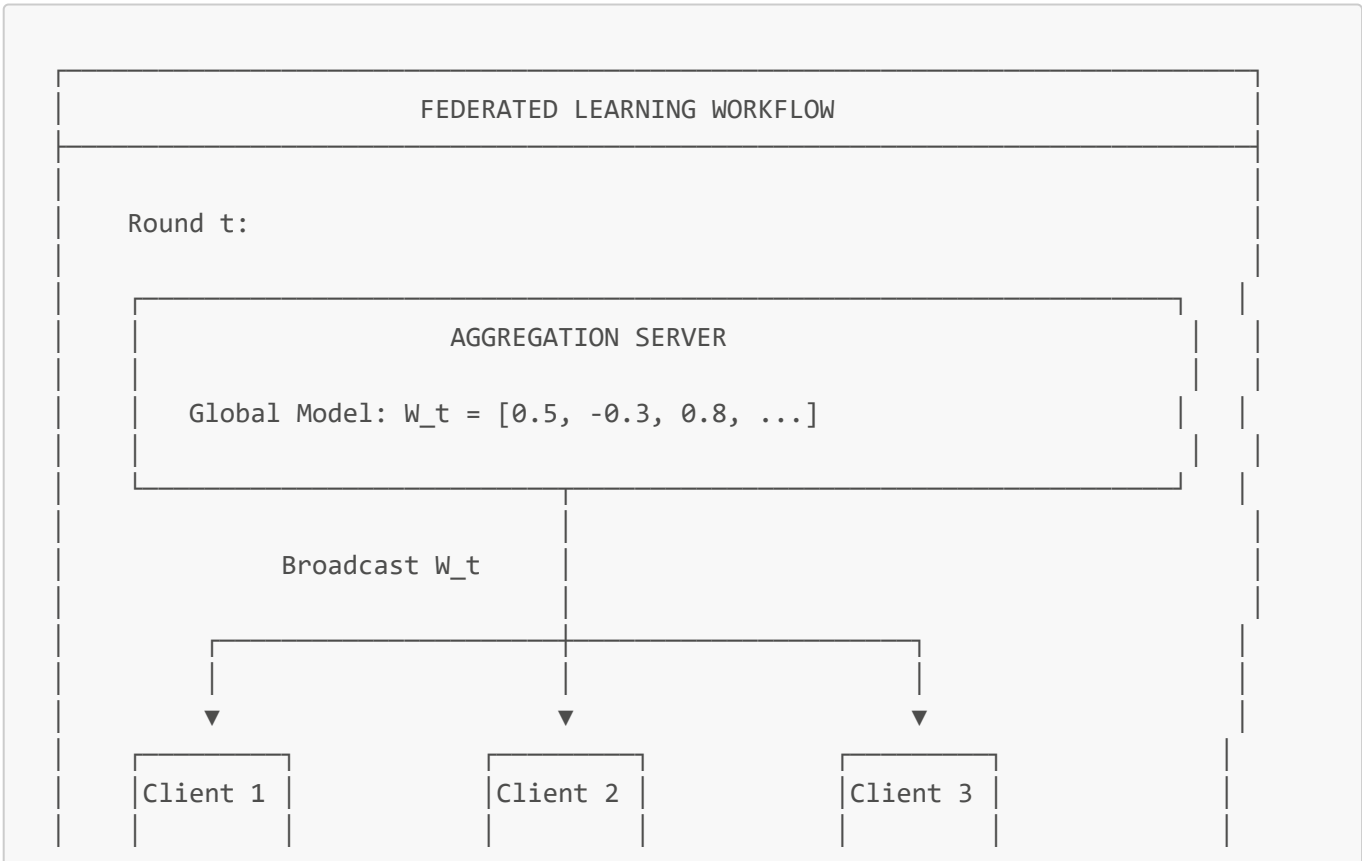


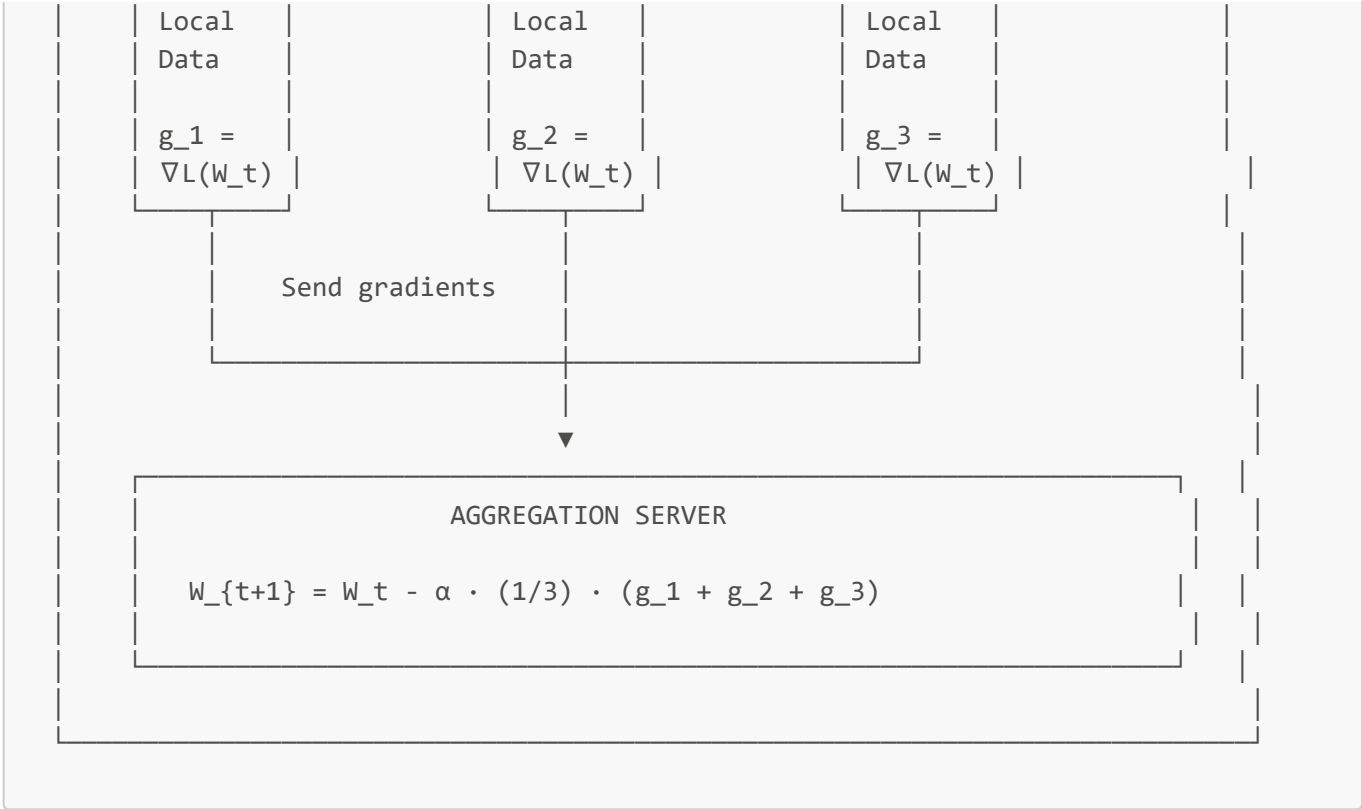


1.2 Federated Learning: A Partial Solution

Federated Learning (FL) addresses this by keeping data local:

1. A central server distributes a global model to all participants
2. Each participant trains on their local data
3. Participants send **gradients** (model updates) instead of raw data
4. Server aggregates gradients to update the global model
5. Repeat until convergence





1.3 Why Federated Learning Alone Is Insufficient

While FL keeps raw data local, it introduces new attack vectors:

Attack	Description	Consequence
Gradient Inversion	Server reconstructs training data from gradients	Privacy breach
Model Poisoning	Malicious client sends bad gradients	Corrupted model
Free-riding	Client claims to train but doesn't	Unfair burden
Data Fabrication	Client trains on fake/unauthorized data	Invalid model

Our Goal: Create a system where:

- ☑ Clients prove training correctness without revealing data
- ☑ Server cannot learn individual gradients
- ☑ Malicious clients cannot poison the model
- ☑ Everything is mathematically verifiable

1.4 Our Contribution

We present a **cryptographic framework** combining:

1. **Zero-Knowledge Proofs (ZKPs)** — Prove statements without revealing secrets
2. **Merkle Tree Commitments** — Bind proofs to specific datasets
3. **Secure Aggregation** — Aggregate gradients without seeing individuals

The result: **Verifiable, private federated learning** with formal security guarantees.

---

2. Background & Preliminaries

This section provides the cryptographic and mathematical foundations needed to understand our system. We explain each concept from first principles.

2.1 Finite Fields and Modular Arithmetic

> **Definition (Finite Field):** A finite field $\mathbb{F}_p$ is a set of integers ${0, 1, 2, ..., p-1}$ where p is a prime number, with addition and multiplication performed modulo p.

Why We Need This: ZK proofs operate over finite fields because:

1. All numbers have fixed size (no overflow issues)
2. Division is well-defined (every non-zero element has an inverse)
3. Enables efficient cryptographic operations

Example: In $\mathbb{F}_7$ (the field with 7 elements):

- $5 + 4 = 9 \mod 7 = 2$
- $5 \times 4 = 20 \mod 7 = 6$
- $5^{-1} = 3$ (because $5 \times 3 = 15 \mod 7 = 1$)

Our Field: We use the BN254 scalar field: $$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

This is a 254-bit prime, providing approximately 128 bits of security.

2.2 Elliptic Curves

> **Definition (Elliptic Curve):** An elliptic curve over a field $\mathbb{F}_p$ is the set of points (x, y) satisfying: $$y^2 = x^3 + ax + b \pmod{p}$$ plus a special "point at infinity" $\mathcal{O}$.

Why We Need This: Elliptic curves provide:

1. **Discrete logarithm hardness:** Given P and $Q = kP$, finding k is computationally infeasible
2. **Efficient group operations:** Point addition and scalar multiplication
3. **Pairing-friendly curves:** Enable advanced cryptographic operations

BN254 Curve: We use the Barreto-Naehrig curve with parameters optimized for pairings:

- 254-bit field
- ~128-bit security level
- Efficient pairing operations (needed for Groth16)

2.3 Cryptographic Hash Functions

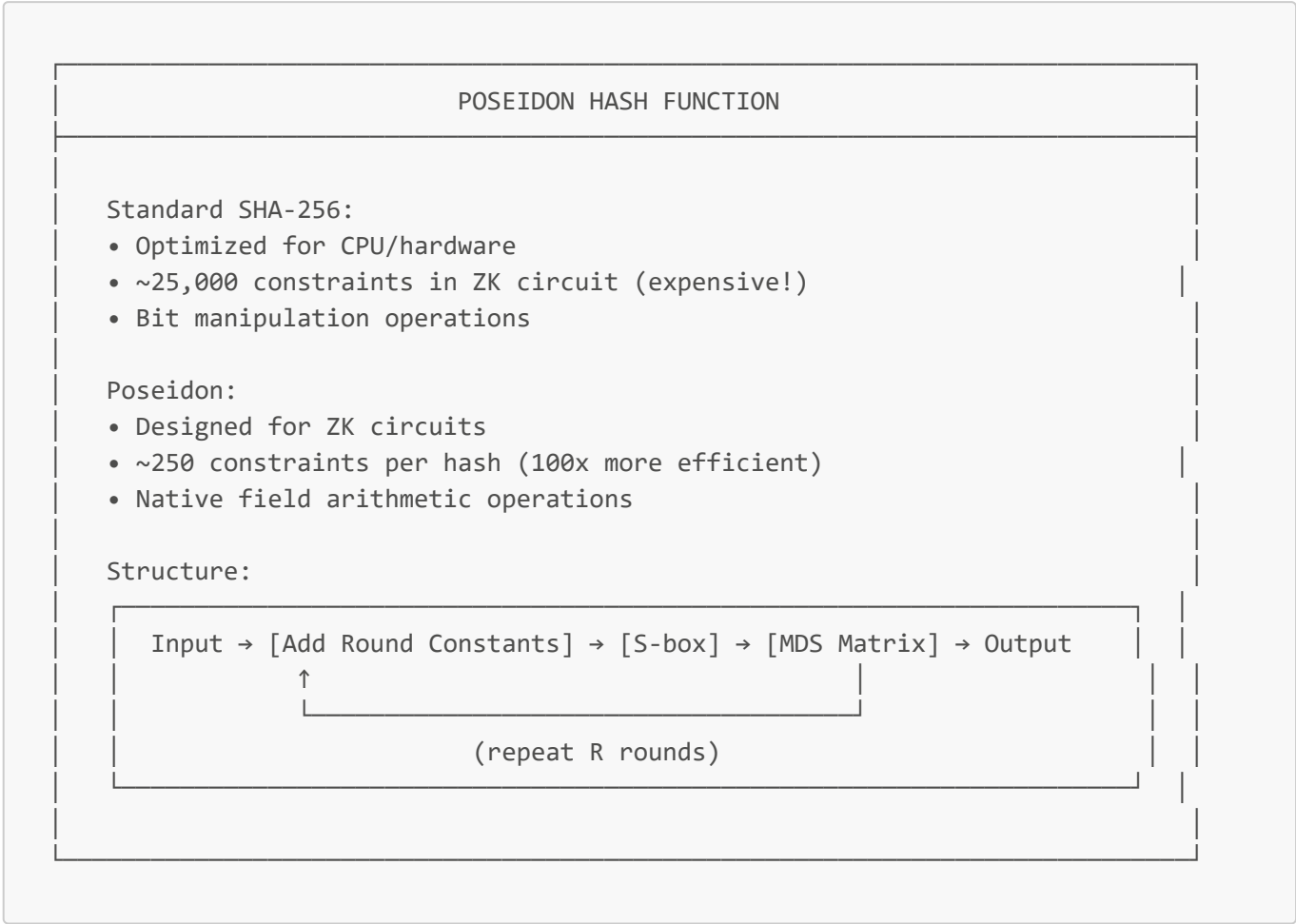
> **Definition (Hash Function):** A hash function $H: {0,1}^* \rightarrow {0,1}^n$ maps arbitrary-length inputs to fixed-length outputs.

Security Properties:

1. **Pre-image resistance:** Given h, hard to find m such that $H(m) = h$

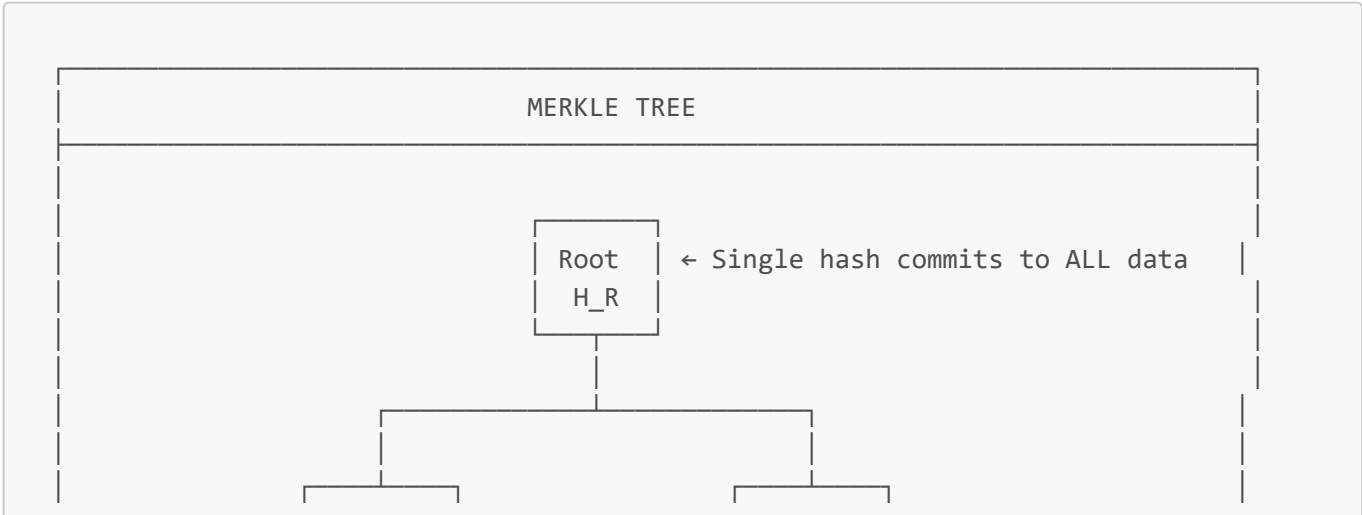
2. **Second pre-image resistance:** Given m_1, hard to find $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$
3. **Collision resistance:** Hard to find any $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$

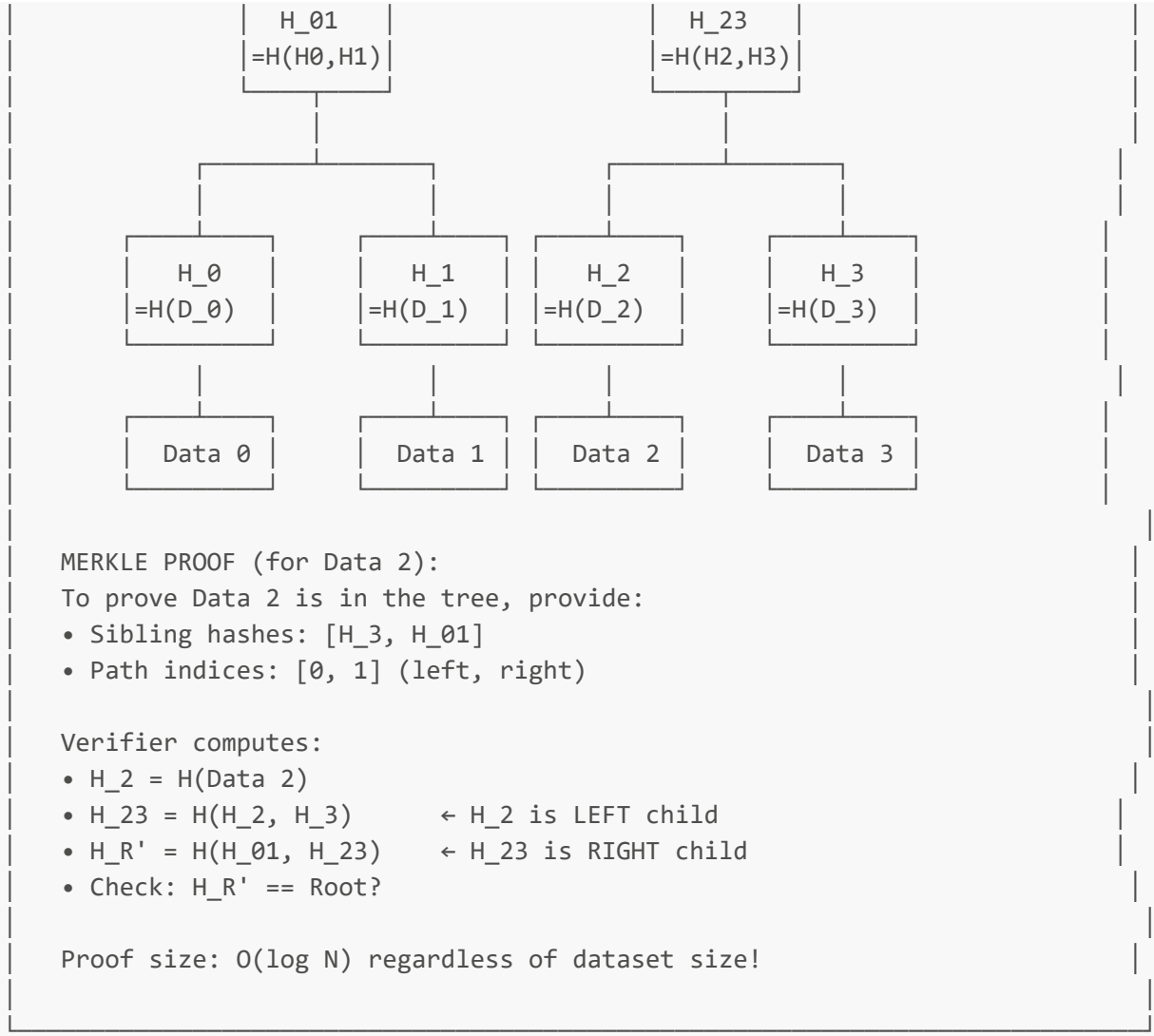
Poseidon Hash: We use Poseidon, a hash function designed for ZK circuits:



2.4 Merkle Trees

> **Definition (Merkle Tree):** A Merkle tree is a binary tree where:
>
> - Each leaf node contains the hash of a data block
> - Each internal node contains the hash of its two children
> - The root is a single hash representing the entire dataset





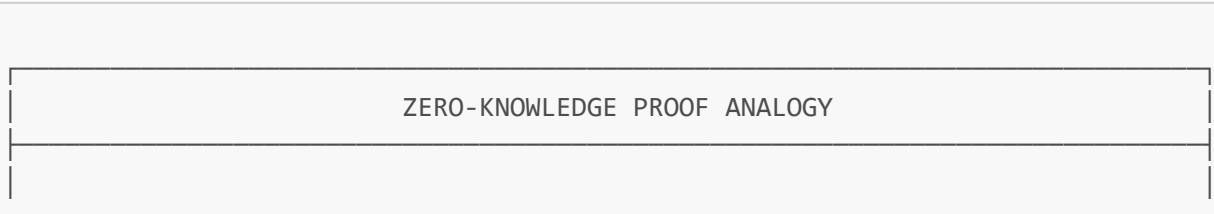
Why Merkle Trees: They provide:

- **Commitment:** The root is a binding commitment to the entire dataset
- **Efficient membership proofs:** $O(\log N)$ proof size
- **Tamper evidence:** Any change to data changes the root

2.5 Zero-Knowledge Proofs

> **Definition (Zero-Knowledge Proof):** A zero-knowledge proof for a statement x with witness w is a protocol where:
>
> 1. **Completeness:** If x is true and prover knows w, verifier accepts
> 2. **Soundness:** If x is false, prover cannot convince verifier (except with negligible probability)
> 3. **Zero-Knowledge:** Verifier learns nothing beyond the truth of x

Intuition: "I can prove I know the solution to a puzzle without showing you the solution."



```
│   EXAMPLE: Proving you know the combination to a safe                        │
│                                                                              │
│   WITHOUT ZK:                                                                │
│   "The combination is 37-42-15"                                              │
│   → Verifier learns the secret!                                              │
│                                                                              │
│   WITH ZK:                                                                   │
│   1. Prover puts a message in the safe, locks it                             │
│   2. Verifier writes a random challenge on paper                             │
│   3. Prover opens safe (without showing combination)                         │
│   4. Prover shows the message + challenge match                              │
│   → Verifier is convinced, but never learns the combination!                 │
│                                                                              │
│   In our system:                                                             │
│   • Secret: Training data, gradient values                                   │
│   • Statement: "I trained correctly with clipped gradients"                  │
│   • Proof: ~200 bytes that convince anyone the statement is true             │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

2.6 zk-SNARKs and Groth16

> **Definition (zk-SNARK):** Zero-Knowledge Succinct Non-interactive ARgument of Knowledge
>
> - **Succinct:** Proof size is small (constant) regardless of computation size
> - **Non-interactive:** Single message from prover to verifier
> - **ARgument of Knowledge:** Prover must "know" the witness (not just that it exists)

Groth16 is the most efficient zk-SNARK for our use case:

- Proof size: 3 group elements (~200 bytes)
- Verification time: 3 pairings (~10ms)
- Prover time: $O(n \log n)$ where n = number of constraints

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                          GROTH16 WORKFLOW                                    │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│   SETUP PHASE (one-time, trusted):                                           │
│   ┌──────────────────────────────────────────────────────────────────────┐   │
│   │  Circuit (R1CS) + Random τ → Proving Key (pk) + Verification Key (vk)│   │
│   │                                                                      │   │
│   │  ⚠ τ must be destroyed! ("toxic waste")                              │   │
│   │  If attacker knows τ, they can forge proofs.                         │   │
│   └──────────────────────────────────────────────────────────────────────┘   │
│                                                                              │
│   PROVING PHASE (per proof):                                                 │
│   ┌──────────────────────────────────────────────────────────────────────┐   │
│   │  Prover:                                                             │   │
│   │    Input: pk, public inputs x, private witness w                     │   │
│   │    Output: proof π = (A, B, C) ∈ G₁ × G₂ × G₁                        │   │
```

```
│   │                                                                   │   │
│   │  Computation: O(n log n) group operations                         │   │
│   └─────────────────────────────────────────────────────────────────┘   │
│                                                                         │
│   VERIFICATION PHASE:                                                   │
│   ┌─────────────────────────────────────────────────────────────────┐   │
│   │  Verifier:                                                      │   │
│   │    Input: vk, public inputs x, proof π                          │   │
│   │    Check: e(A, B) = e(α, β) · e(Σ xᵢLᵢ, γ) · e(C, δ)            │   │
│   │                                                                 │   │
│   │  Computation: 3 pairings (~10ms)                                │   │
│   │  Result: Accept or Reject                                       │   │
│   └─────────────────────────────────────────────────────────────────┘   │
│                                                                         │
└─────────────────────────────────────────────────────────────────────────┘
```

2.7 R1CS: Rank-1 Constraint System

> **Definition (R1CS):** A system of equations of the form: $$(\vec{a} \cdot \vec{s}) \times (\vec{b} \cdot \vec{s}) = (\vec{c} \cdot \vec{s})$$ where $\vec{s}$ is the witness vector and $\vec{a}, \vec{b}, \vec{c}$ are coefficient vectors.

Why R1CS: Groth16 works on R1CS. We write circuits in Circom, which compiles to R1CS.

Example: To prove $x^2 + y^2 = z$:

```
Constraint 1: x × x = x²
Constraint 2: y × y = y²
Constraint 3: x² + y² = z (linear, rewritten as: (x² + y²) × 1 = z)
```

2.8 Gradient Descent and Differential Privacy

> **Definition (Stochastic Gradient Descent):** An optimization algorithm that updates model weights: $$W_{t+1} = W_t - \alpha \cdot \nabla L(W_t; B_t)$$ where α is the learning rate and B_t is a mini-batch.

Gradient Clipping is a technique to bound gradient magnitudes: $$g' = g \cdot \min\left(1, \frac{\tau}{|g|_2}\right)$$

This ensures $|g'|_2 \leq \tau$, which:

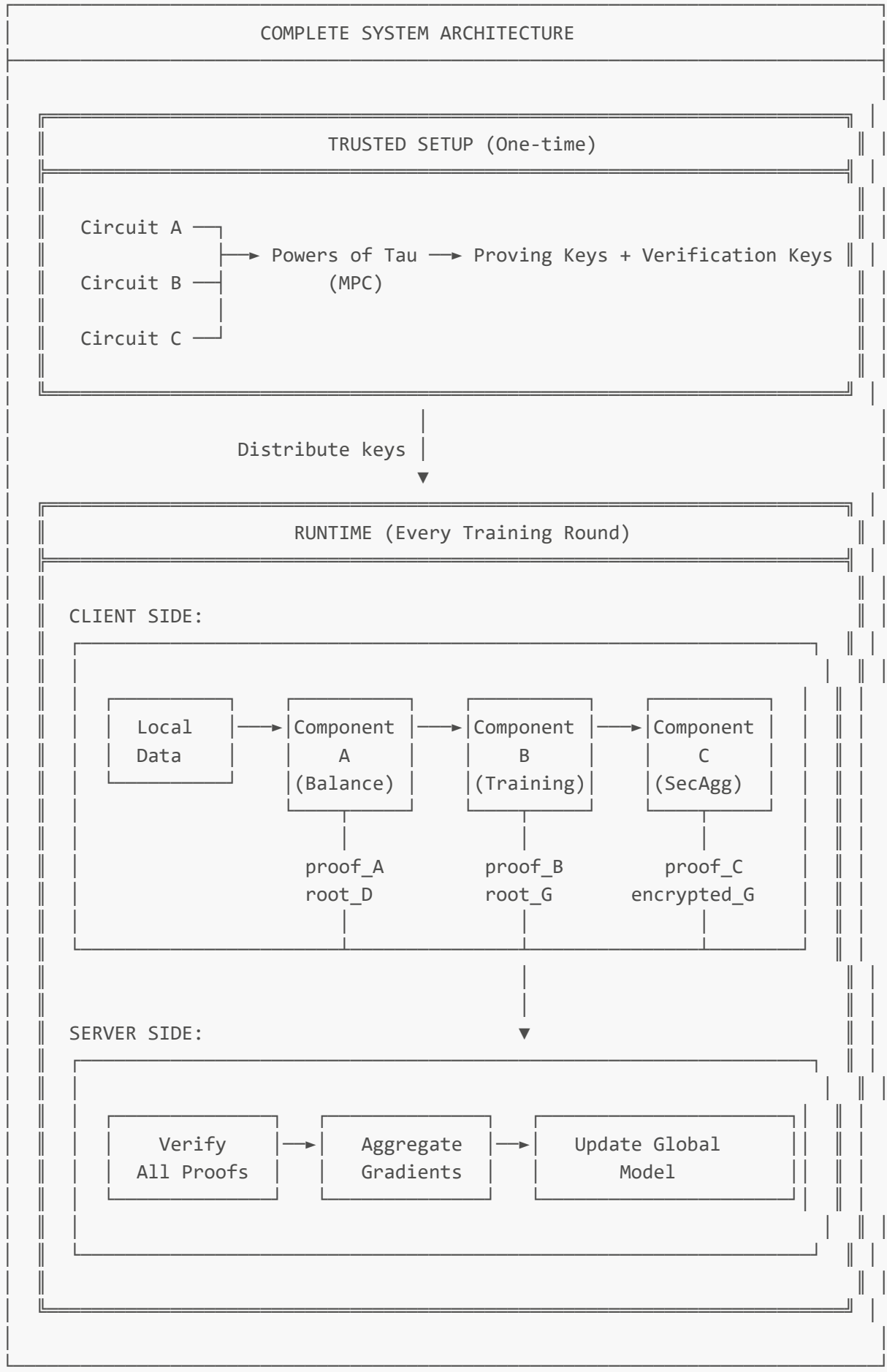
1. Prevents model poisoning attacks
2. Enables differential privacy guarantees
3. Stabilizes training

---

3. System Architecture

3.1 High-Level Overview

Our system implements a complete federated learning pipeline with zero-knowledge verification:



3.2 Component Overview

Component	Circuit File	Purpose	Public Outputs
A	`balance_unified.circom`	Prove dataset properties (class balance)	`root_D`, `c0`, `c1`
B	`sgd_step_v5.circom`	Prove training integrity (batch membership, clipping)	`root_D`, `root_G`, `tauSquared`
C	`secure_agg_client.circom`	Prove masked gradient well-formedness	`root_G`, `masked_update`

3.3 Trust Model

Assumptions:

- Clients are mutually distrustful (some may be malicious)
- Server is honest-but-curious (follows protocol but tries to learn)
- Cryptographic primitives are secure (Poseidon, BN254, Groth16)
- Trusted setup was performed correctly (toxic waste destroyed)

Security Goals:

- **Privacy:** Server learns only aggregate gradient
- **Integrity:** Only valid training updates are accepted
- **Accountability:** Proofs create audit trail

---

4. Component A: Balance Proof

4.1 Motivation

Before training, we want to verify dataset properties without revealing the data. For example, a medical dataset should have balanced classes (similar numbers of healthy/sick patients) to train an unbiased model.

Problem: How can a client prove "My dataset has 60 healthy and 68 sick patients" without revealing which patients are which?

Solution: Zero-knowledge balance proof.

4.2 Circuit Design

```
template BalanceProofUnified(N, DEPTH, MODEL_DIM) {
    // PUBLIC INPUTS
    signal input client_id;
    signal input root;        // Merkle root of dataset
    signal input N_public;    // Total samples
    signal input c0;          // Class 0 count
    signal input c1;          // Class 1 count
```

```
    // PRIVATE INPUTS
    signal input features[N][MODEL_DIM];
    signal input labels[N];
    signal input siblings[N][DEPTH];
    signal input pathIndices[N][DEPTH];

    // CONSTRAINTS...
}
```

4.3 Constraints Explained

Constraint 1: Label Booleanity

```
for (var i = 0; i < N; i++) {
    labels[i] * (labels[i] - 1) === 0;
}
```

This enforces $\text{label}_i \in {0, 1}$ using the algebraic identity: $$b \cdot (b - 1) = 0 \iff b \in {0, 1}$$

Constraint 2: Count Verification

```
signal partialSums[N + 1];
partialSums[0] <== 0;
for (var i = 0; i < N; i++) {
    partialSums[i + 1] <== partialSums[i] + labels[i];
}
partialSums[N] === c1;
```

Since labels are binary, summing them gives the count of 1s.

Constraint 3: Total Consistency

```
c0 + c1 === N_public;
N_public === N;
```

Constraint 4: Merkle Membership

```
for (var i = 0; i < N; i++) {
    leafHashers[i] = VectorHash(MODEL_DIM + 1);
    // Hash features || label
    for (var j = 0; j < MODEL_DIM; j++) {
        leafHashers[i].values[j] <== features[i][j];
    }
    leafHashers[i].values[MODEL_DIM] <== labels[i];
```

```
    // Verify Merkle proof
    merkleProofs.leafHashes[i] <== leafHashers[i].hash;
}
```

4.4 Security Properties

Property	Guarantee
Soundness	Cannot prove false counts (would require Poseidon collision)
Zero-Knowledge	Verifier learns only `(root, c0, c1)`, not individual labels
Binding	Same `root` must be used in Component B

---

5. Component B: Training Integrity Proof

5.1 Motivation

During federated learning, clients compute gradients locally. We need to verify:

1. The gradient was computed on the committed dataset
2. The gradient was properly clipped (bounded magnitude)
3. The gradient matches a public commitment

Challenge: Gradients can be negative, but ZK circuits operate over positive field elements.

5.2 The Signed Arithmetic Problem

In a finite field $\mathbb{F}_p$, there are no "negative" numbers. A negative value $-x$ is represented as $p - x$.

Problem: To verify $|g|^2 \leq \tau^2$, we need to square gradient components. But:

- If $g_i = -5$ is stored as $p - 5$
- Then $g_i^2 = (p-5)^2 \mod p \neq 25$

This breaks norm computation!

5.3 Solution: Sign-Magnitude Decomposition

We represent each gradient component as: $$g_i = g_i^{+} - g_i^{-}$$

where $g_i^{+}, g_i^{-} \geq 0$ and at most one is non-zero.

Example:

g_i	g_i^{+}	g_i^{-}
42	42	0
-37	0	37

g_i	g_i^{+}	g_i^{-}
0	0	0

Now the squared norm is: $$|g|^2 = \sum_i (g_i^{+})^2 + (g_i^{-})^2$$

This works because exactly one term is non-zero per component.

5.4 Circuit Design (v5)

```
template VerifyClippingSound(DIM) {
    signal input gradPos[DIM];
    signal input gradNeg[DIM];
    signal input tauSquared;

    signal output gradient[DIM];
    signal output normSquared;
    signal output valid;

    // STEP 1: At most one non-zero per component
    for (var j = 0; j < DIM; j++) {
        gradPos[j] * gradNeg[j] === 0;
    }

    // STEP 2: Compute norm squared
    signal partialNorm[DIM + 1];
    partialNorm[0] <== 0;

    for (var j = 0; j < DIM; j++) {
        posSquared[j] <== gradPos[j] * gradPos[j];
        negSquared[j] <== gradNeg[j] * gradNeg[j];
        partialNorm[j + 1] <== partialNorm[j] + posSquared[j] + negSquared[j];
    }

    normSquared <== partialNorm[DIM];

    // STEP 3: Verify clipping
    component lt = LessThan(128);
    lt.in[0] <== normSquared;
    lt.in[1] <== tauSquared + 1;
    valid <== lt.out;

    // STEP 4: Reconstruct gradient for commitment
    for (var j = 0; j < DIM; j++) {
        gradient[j] <== gradPos[j] - gradNeg[j];
    }
}
```

5.5 Full Training Circuit

```
template TrainingStepV5(BATCH_SIZE, MODEL_DIM, DEPTH) {
    // PUBLIC INPUTS
    signal input client_id;
    signal input root_D;      // Dataset commitment (binds to Component A)
    signal input root_G;      // Gradient commitment (binds to Component C)
    signal input tauSquared;  // Clipping threshold

    // PRIVATE INPUTS
    signal input gradPos[MODEL_DIM];
    signal input gradNeg[MODEL_DIM];
    signal input features[BATCH_SIZE][MODEL_DIM];
    signal input labels[BATCH_SIZE];
    signal input siblings[BATCH_SIZE][DEPTH];
    signal input pathIndices[BATCH_SIZE][DEPTH];

    // STEP 1: Verify batch membership
    component batchVerifier = BatchMerkleProofPreHashed(BATCH_SIZE, DEPTH);
    batchVerifier.root <== root_D;

    for (var i = 0; i < BATCH_SIZE; i++) {
        // Same leaf hash as Component A!
        leafHash[i] = VectorHash(MODEL_DIM + 1);
        for (var j = 0; j < MODEL_DIM; j++) {
            leafHash[i].values[j] <== features[i][j];
        }
        leafHash[i].values[MODEL_DIM] <== labels[i];

        batchVerifier.leafHashes[i] <== leafHash[i].hash;
        // ... siblings and path indices
    }

    // STEP 2: Verify gradient clipping (SOUND!)
    component clipVerifier = VerifyClippingSound(MODEL_DIM);
    // ... inputs
    clipVerifier.valid === 1;

    // STEP 3: Verify gradient commitment
    component gradHash = VectorHash(MODEL_DIM);
    for (var j = 0; j < MODEL_DIM; j++) {
        gradHash.values[j] <== clipVerifier.gradient[j];
    }
    root_G === gradHash.hash;
}
```

5.6 Why v5 is Sound (Security Analysis)

Previous versions (v1-v4) had a critical vulnerability:

```
// v4 (VULNERABLE)
signal input normSquared;  // Prover provides this!
```

```
// Prover can lie: claim normSquared = 100 when actual ||g||² = 10000
// The circuit trusts this value → clipping check passes incorrectly
```

v5 fixes this by computing `normSquared` inside the circuit:

```
// v5 (SOUND)
// normSquared is COMPUTED, not INPUT
partialNorm[j + 1] <== partialNorm[j] + posSquared[j] + negSquared[j];
normSquared <== partialNorm[DIM];

// Prover cannot lie because the value is derived from gradPos/gradNeg
// And gradPos/gradNeg are bound to root_G via gradient reconstruction
```

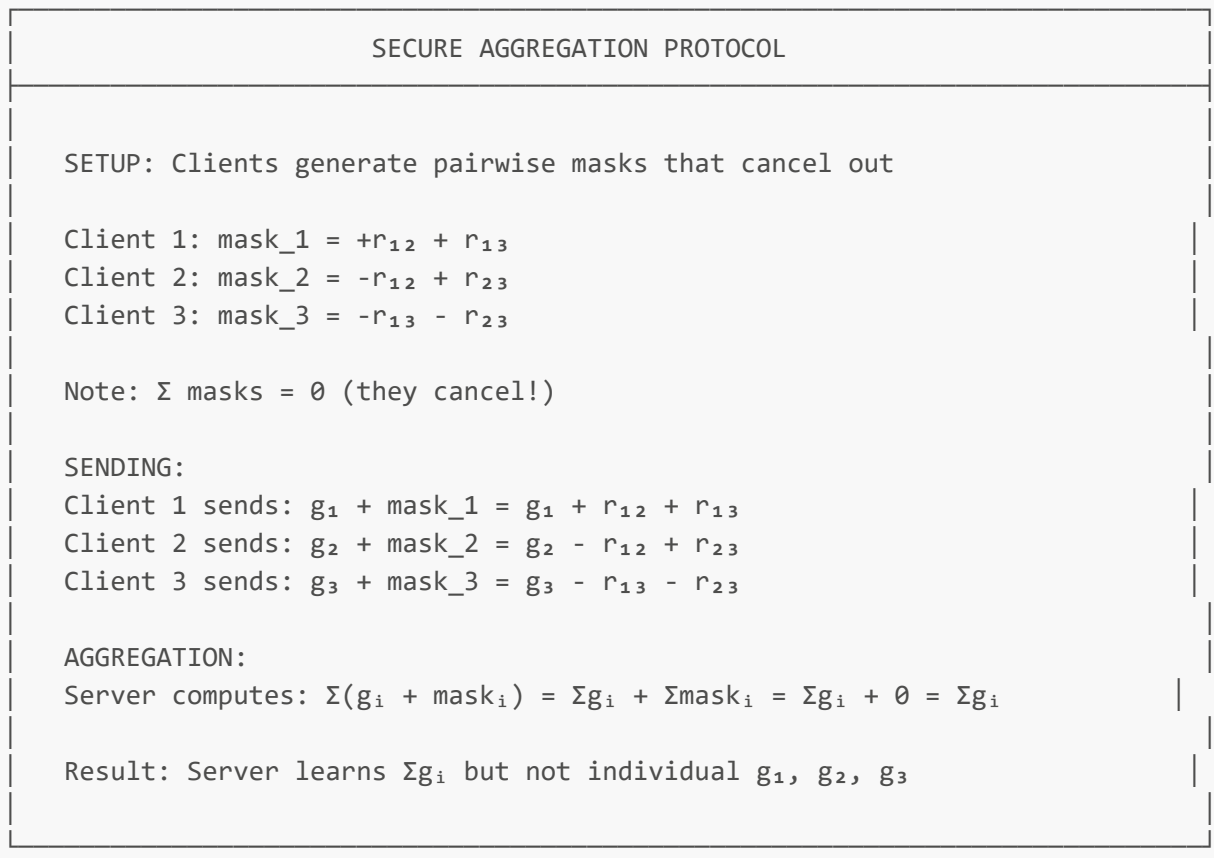
---

6. Component C: Secure Aggregation

6.1 Motivation

Even with ZK proofs, if the server sees individual gradients, it might infer private information. **Secure Aggregation** ensures the server only learns the sum of gradients.

6.2 Basic Protocol



6.3 Dropout Tolerance

Problem: If Client 2 drops out, the masks don't cancel: $$\sum_{i \neq 2} \text{mask}*i = r{12} + r_{13} - r_{13} - r_{23} = r_{12} - r_{23} \neq 0$$

Solution: Use threshold secret sharing so surviving clients can reconstruct dropped masks.

6.4 ZK Well-Formedness Proof

Component C proves:

1. The gradient matches commitment `root_G` (from Component B)
2. The mask was correctly derived from shared secrets
3. The masked update equals `gradient + mask`

---

7. Cryptographic Binding Between Components

7.1 The Binding Problem

A malicious client might try to:

- Prove balance on Dataset X
- Train on Dataset Y
- Submit gradient from Dataset Z

We prevent this through **cryptographic binding**.

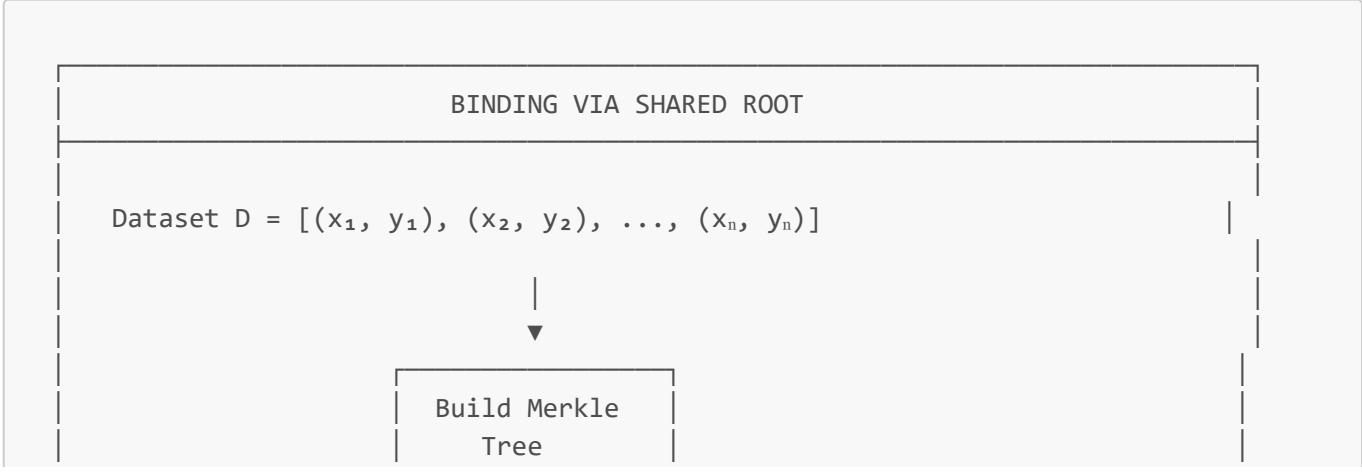
7.2 Unified Leaf Hash Structure

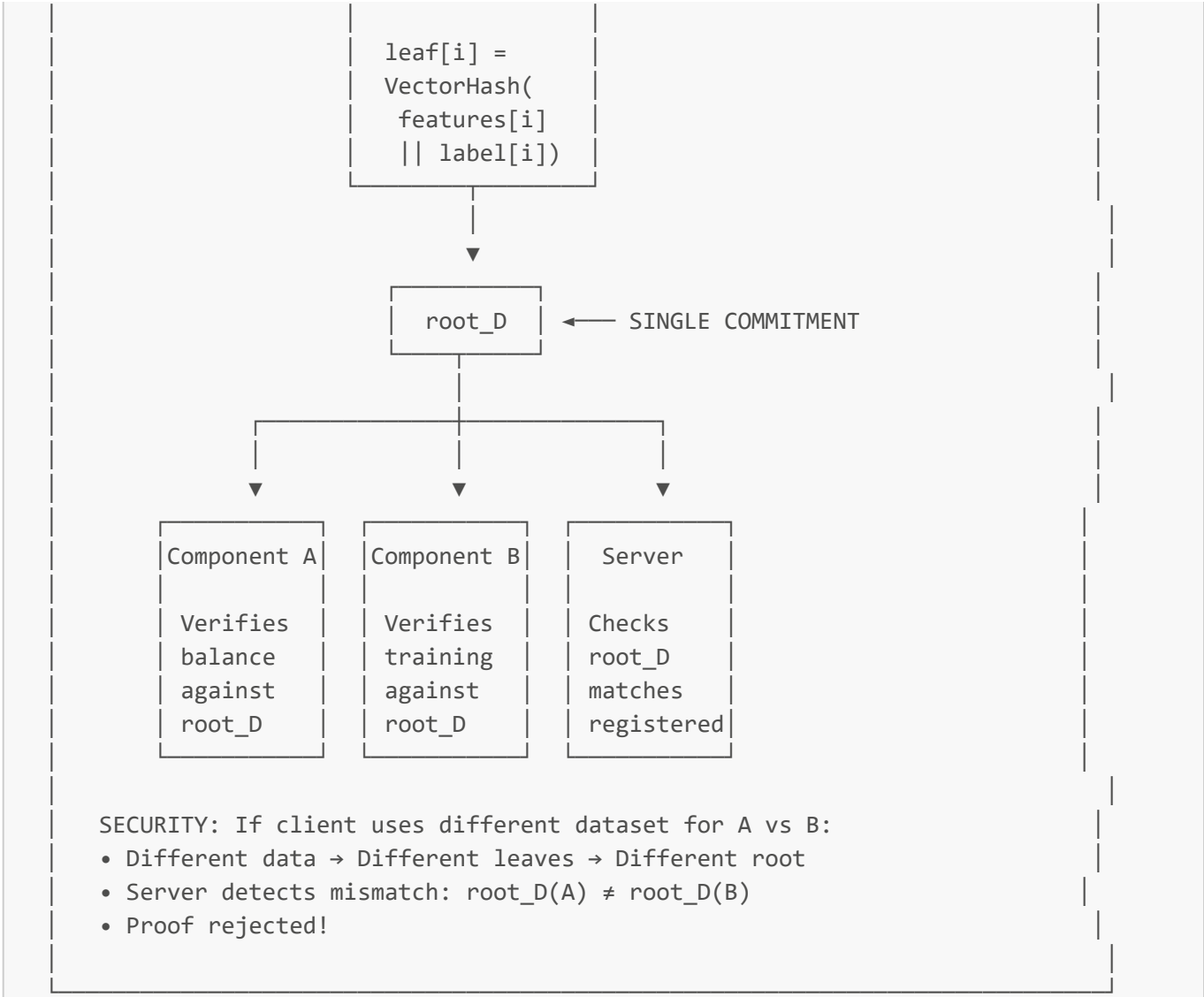
Critical Design Decision: Components A and B use identical leaf hashing:

```
// In BOTH balance_unified.circom AND sgd_step_v5.circom:
leaf_hash = VectorHash(features || label)
```

This means:

- Both components produce the same Merkle root for the same dataset
- A proof for Component A binds to the exact same data used in Component B
- Cannot swap datasets without detection





7.3 Gradient Commitment Binding

Similarly, `root_G` binds Component B to Component C:

```
Component B: Computes gradient, outputs root_G = VectorHash(gradient)
Component C: Takes root_G as input, proves masked_gradient is valid
```

The server verifies that `root_G` from both proofs match.

---

8. Security Analysis

8.1 Threat Model

Adversary	Capabilities	Goal
Malicious Client	Control their own inputs, see public outputs	Learn others' data, poison model

Adversary	Capabilities	Goal
Honest-but-Curious Server	See all proofs and public values	Learn individual gradients
External Attacker	See network traffic, public blockchain	Break privacy or integrity

8.2 Security Properties

Theorem 1 (Soundness): A malicious client cannot produce a valid proof for:

- A batch not in the committed dataset
- A gradient exceeding the clipping bound
- A gradient not matching the commitment

Proof sketch: Each property is enforced by R1CS constraints. Breaking soundness requires either:

1. Finding a Poseidon collision (computationally infeasible)
2. Breaking Groth16 soundness (requires knowledge of toxic waste)

Theorem 2 (Zero-Knowledge): The proofs reveal nothing beyond public outputs.

Proof sketch: Groth16 is proven zero-knowledge under the DDH assumption on BN254.

Theorem 3 (Binding): A client cannot prove balance on dataset X while training on dataset Y.

Proof sketch: Both components compute `root_D` using identical `VectorHash(features || label)`. Different datasets yield different roots with overwhelming probability (collision resistance).

8.3 What We Do NOT Prove

Property	Status	Reason
Gradient correctness	❌ Not proven	Would require zkML (very expensive)
Data authenticity	❌ Not proven	Client could use synthetic data
Honest feature values	❌ Not proven	Features are private inputs

These are inherent limitations of our threat model, not implementation flaws.

---

9. Implementation Details

9.1 Technology Stack

Component	Technology	Purpose
Circuit Language	Circom 2.0.0	Define ZK circuits
Proof System	Groth16 (snarkjs)	Generate/verify proofs
Curve	BN254	Elliptic curve for pairings

Component	Technology	Purpose
Hash Function	Poseidon	ZK-friendly hashing
Runtime	Node.js	Witness generation, testing

9.2 Circuit Parameters

```
// Component B parameters
BATCH_SIZE = 8     // Samples per training step
MODEL_DIM = 16     // Gradient dimensions
DEPTH = 7          // Merkle tree depth (supports 128 samples)
```

9.3 Constraint Counts

Circuit	Non-linear Constraints	Linear Constraints
Balance (N=128)	~20,000	~25,000
Training v5	23,133	32,500
Secure Agg	~15,000	~20,000

9.4 VectorHash Implementation

For vectors larger than Poseidon's native capacity (16 elements), we use chunked hashing:

```
template VectorHash(DIM) {
    signal input values[DIM];
    signal output hash;

    var CHUNK_SIZE = 16;

    if (DIM <= CHUNK_SIZE) {
        // Hash directly
        component hasher = PoseidonHashN(DIM);
        // ...
    } else {
        // Hash in chunks, then hash the chunk hashes
        var NUM_CHUNKS = (DIM + CHUNK_SIZE - 1) / CHUNK_SIZE;

        for (var c = 0; c < NUM_CHUNKS; c++) {
            chunkHasher[c] = PoseidonHashN(chunk_length);
            // ...
        }

        component finalHasher = PoseidonHashN(NUM_CHUNKS);
        // ...
    }
}
```

---

10. Performance Evaluation

10.1 Proof Generation Time

Circuit	Constraints	Proving Time	Memory
Training v5	23,133	~30 seconds	~2 GB
Balance	~20,000	~25 seconds	~1.5 GB

Tested on: Intel i7-10700K, 32GB RAM

10.2 Verification Time

Metric	Value
Verification time	~10 ms
Proof size	192 bytes
Public inputs	4 field elements

10.3 Comparison with Alternatives

Approach	Proof Size	Verify Time	Trusted Setup
Groth16 (ours)	192 B	10 ms	Yes
PLONK	~400 B	15 ms	Universal
STARKs	~50 KB	50 ms	No
Bulletproofs	~1 KB	100 ms	No

We chose Groth16 for its optimal proof size and verification time, accepting the trusted setup requirement.

---

11. Limitations & Future Work

11.1 Current Limitations

1. **No gradient correctness verification:** We prove the gradient is clipped and committed, but not that it's the correct gradient for the given batch. Full verification would require zkML.
2. **Trusted setup required:** Groth16 needs a ceremony. Compromise of toxic waste breaks soundness.
3. **Fixed circuit parameters:** Changing `BATCH_SIZE` or `MODEL_DIM` requires recompilation and new trusted setup.
4. **Computational overhead:** Proof generation is expensive (~30s), limiting real-time applications.

11.2 Future Directions

1. **zkML Integration:** Prove gradient correctness using frameworks like EZKL or zkLLVM.
2. **Universal Setup:** Migrate to PLONK for reusable trusted setup.
3. **Hardware Acceleration:** GPU/FPGA proving for real-time proofs.
4. **Recursive Proofs:** Aggregate multiple training rounds into a single proof.

---

12. Conclusion

We presented a comprehensive framework for **verifiable federated learning** using zero-knowledge proofs. Our system enables:

- **Privacy:** Individual data and gradients remain confidential
- **Integrity:** Mathematical guarantees that training was performed correctly
- **Accountability:** Cryptographic audit trail for all computations
- **Interoperability:** Standard Groth16 proofs verifiable by any party

The three-component architecture (Balance, Training, Secure Aggregation) provides modular security guarantees while maintaining cryptographic binding through shared commitments.

Our v5 Training circuit resolves the critical signed arithmetic challenge through sign-magnitude decomposition, achieving **sound** gradient clipping verification—a property missing in earlier versions.

This work demonstrates that **privacy and verifiability are not mutually exclusive** in distributed machine learning systems.

---

13. References

1. Boneh, D., et al. "Zk-SNARKs: Security and Efficiency." (2013)
2. Groth, J. "On the Size of Pairing-based Non-interactive Arguments." EUROCRYPT (2016)
3. Grassi, L., et al. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems." USENIX Security (2021)
4. McMahan, B., et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data." AISTATS (2017)
5. Bonawitz, K., et al. "Practical Secure Aggregation for Privacy-Preserving Machine Learning." CCS (2017)
6. Abadi, M., et al. "Deep Learning with Differential Privacy." CCS (2016)
7. Circom Documentation. https://docs.circom.io/
8. snarkjs Documentation. https://github.com/iden3/snarkjs

Purdue University — Applied Cryptography — Fall 2025