Naami: A Privacy-Preserving Decentralized Expense Tracking System with Cryptographic Verifiability

Naami Development Team
ted-nami@proton.me

February 27, 2026

Abstract

We present Naami, a novel decentralized expense tracking system that combines end-to-end encryption, event sourcing, zero-knowledge proofs, and blockchain-based cryptographic commitments to provide privacy-preserving collaborative finance management. The system addresses the fundamental trilemma of privacy, verifiability, and cost-efficiency in collaborative financial applications. Our architecture introduces seven key innovations: (1) wallet-based authentication with session management, (2) group-level end-to-end encryption with dynamic key distribution, (3) defense-in-depth server-side PII protection using HMAC-SHA-256 for email lookup and AES-256-GCM for at-rest encryption of device tokens and nicknames, (4) encrypted event sourcing with causal ordering, (5) a triple-root cryptographic commitment system using Merkle Mountain Ranges with Poseidon hashing for ZK-friendly verification, (6) hierarchical two-level MMR structure with Groth16 ZK proof verification enabling unbounded session scaling beyond the 32-event circuit limit through batch-level (up to 32 events) and global-level (batch root aggregation) organization, and (7) sliding window debounce mechanism (6-hour inactivity threshold) that accumulates events before generating ZK proofs, reducing costs by 97.7% compared to real-time anchoring. The system supports a dual settlement model combining on-chain refunds (SOL transfers) with off-chain reimbursements (traditional payment methods), both integrated into the same event sourcing and cryptographic commitment pipeline. We implement the system on Solana using Light Protocol's zero-knowledge compression and on-chain Groth16 verification, achieving 99.8% cost reduction compared to Ethereum ($0.00021 per event amortized vs $0.10-$0.20) while maintaining full cryptographic verifiability. A companion tokenomics paper [25] describes the NAAMI token economic model, governance framework, and protocol fee structure. Our evaluation demonstrates that Naami can support collaborative sessions with logarithmic proof sizes $O(\log n)$ for event verification, constant on-chain storage $O(1)$, constant 256-byte ZK proofs, support for up to 1024 events per entity type (members, expenses, refunds) over the session lifetime, and complete event privacy through zero-knowledge verification, making decentralized expense tracking practical for real-world deployment.

1 Introduction

1.1 Context and Motivation

Collaborative expense tracking applications serve over 500 million users worldwide, facilitating group financial management for activities ranging from shared household expenses to travel group coordination [1, 2]. However, existing solutions face a fundamental trilemma between privacy, verifiability, and cost-efficiency:

- **Centralized systems** (e.g., Tricount, Splitwise) provide good user experience but require full trust in service providers, expose user data to potential breaches, and create single points of failure.

- **Fully on-chain systems** offer verifiability but incur prohibitive storage costs ($0.50-$2.00 per expense on Ethereum), lack privacy due to public ledger visibility, and suffer from poor latency.
- **Hybrid systems** with off-chain computation often lack cryptographic guarantees or require complex challenge-response protocols that are impractical for consumer applications.

1.2 Problem Statement

We formulate the problem as follows: *Design a collaborative expense tracking system that achieves* (P1) *end-to-end encryption such that servers cannot access expense details*, (P2) *cryptographic verifiability enabling clients to detect tampering or reordering of expense events*, (P3) *cost-efficiency with transaction costs below $0.01 per expense*, and (P4) *practical performance with sub-second latency for common operations.*

Existing approaches fail to simultaneously achieve all four properties. Centralized systems violate P1 and P2. Naive blockchain approaches violate P3 and P4. Traditional commitment schemes [4] with balanced trees require expensive rebalancing operations when appending new elements, violating P3.

1.3 Contributions

This paper makes the following contributions:

1. We present a novel architecture for privacy-preserving collaborative expense tracking that combines wallet-based authentication, group-level E2EE, encrypted event sourcing, zero-knowledge proofs, and blockchain commitments.
2. We introduce a defense-in-depth approach to personally identifiable information (PII) protection: E2EE for financial data complemented by server-side cryptographic protections (HMAC-SHA-256 for email lookup without plaintext storage, AES-256-GCM for at-rest encryption of device tokens and nicknames), ensuring metadata confidentiality even under full server compromise.
3. We introduce a triple-root cryptographic commitment system using Merkle Mountain Ranges (MMR) with Poseidon hashing for all three roots (members, expenses, refunds), enabling ZK-friendly verification with O(log n) proof sizes for event verification and O(1) on-chain storage (96 bytes total).
4. We develop a hierarchical two-level MMR structure with Groth16 ZK proof verification enabling unbounded session scaling: batch-level MMRs (up to 32 events) are aggregated into a global-level MMR using batch roots as leaves, allowing sessions to exceed the 32-event circuit limit while maintaining constant 256-byte proof size and complete event privacy through zero-knowledge verification on Solana.
5. We design a sliding window debounce mechanism with 6-hour inactivity threshold that accumulates events before generating ZK proofs, enabling a single global proof to anchor hundreds of events organized into multiple batches. This reduces transaction costs by 97.7% compared to real-time proof generation (1 proof for 100 events vs 100 individual proofs), with architectural support for up to 1024 events per entity type (members, expenses, refunds) over the session lifetime.
6. We design a dual settlement model combining on-chain refunds (SOL transfers committed to the Refunds MMR) with off-chain reimbursements (traditional payment methods) that follow the same event sourcing pattern, along with a formal claims protocol for debt tracking and multi-modal settlement.

7. We design an efficient key distribution protocol for multi-user encrypted sessions that enables dynamic member addition without requiring key rotation or historical re-encryption.
8. We implement and evaluate the system on Solana using Light Protocol's ZK compression and on-chain Groth16 verification embedded via build.rs, demonstrating 99.8% cost reduction ($0.00021 per event amortized vs $0.10-$0.20 on Ethereum) while maintaining full cryptographic verifiability and privacy.
9. We provide a comprehensive security analysis demonstrating resistance to common attacks including server compromise, network tampering, and malicious clients, with additional privacy guarantees from zero-knowledge proof soundness. The companion tokenomics paper [25] details the NAAMI token economic model and governance framework.

2 Related Work

2.1 Blockchain-Based Financial Applications

Decentralized finance (DeFi) applications [3] have demonstrated the viability of blockchain-based financial systems. However, most DeFi protocols focus on algorithmic trading and lending rather than collaborative expense tracking. Furthermore, they typically lack privacy guarantees due to public ledger visibility.

2.2 Cryptographic Commitments

Merkle Trees [4] provide efficient membership proofs with O(log n) proof sizes. However, standard Merkle Trees require complete tree reconstruction when adding elements, making them unsuitable for append-only logs. Merkle Mountain Ranges [5,6] address this limitation by maintaining multiple tree peaks, enabling efficient append operations without modifying existing structure.

2.3 End-to-End Encryption in Collaborative Systems

Signal Protocol [7] provides strong E2EE for messaging with forward secrecy. However, its complexity and key ratcheting mechanisms are unnecessary for financial data where message ordering is strictly sequential. Age encryption [8] offers simpler asymmetric encryption suitable for file encryption but lacks built-in group key management.

2.4 Event Sourcing

Event sourcing [9] is a well-established pattern for maintaining audit trails and enabling temporal queries. However, combining event sourcing with E2EE and blockchain commitments introduces novel challenges in proof generation and verification.

2.5 Zero-Knowledge Proof Systems

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) [21] enable proving computational statements without revealing inputs. Groth16 [21] provides constant-size proofs (256 bytes uncompressed) with fast verification, making it suitable for blockchain applications. Circom [22] is a widely-used domain-specific language for expressing arithmetic circuits. Poseidon [23] is a ZK-friendly hash function optimized for SNARK circuits, requiring significantly fewer constraints than traditional hash functions like SHA-256 or Keccak256. While prior work has explored ZK proofs for privacy-preserving blockchain transactions [24], our application to batched MMR updates with Poseidon hashing represents a novel approach to reducing on-chain transaction costs while maintaining verifiability.

3 System Model and Threat Model

3.1 System Model

We consider a system with three classes of participants:

- **Users** ($U = \{u_1, u_2, \ldots, u_m\}$) create and manage expenses through client applications. Each user possesses cryptographic key pairs for authentication and encryption.
- **API Server** (S) coordinates operations, stores encrypted data, and manages blockchain interactions. The server is semi-trusted: it correctly executes protocols but may attempt to access encrypted data.
- **Blockchain** (B) serves as a tamper-proof ledger storing cryptographic commitments. We assume the blockchain is secure (honest majority) and provides finality within a reasonable timeframe.

A **session** $\sigma = (id, M, E, R, C)$ represents a collaborative expense tracking context where $M \subseteq U$ is the set of members, $E = [e_1, e_2, \ldots, e_n]$ is the ordered sequence of expense events, $R = [r_1, r_2, \ldots, r_p]$ is the ordered sequence of refunds and reimbursements, and C is the set of active claims derived from balance computation. Settlement occurs through two modes: on-chain refunds (SOL transfers between members) and off-chain reimbursements (traditional payment methods such as bank transfers or cash), both recorded as events in R.

3.2 Threat Model

We consider the following adversarial capabilities:

1. **Honest-but-Curious Server**: The API server correctly executes protocols but attempts to learn information from encrypted data, network traffic, or access patterns.
2. **Network Attacker**: An adversary can observe, delay, or reorder network messages but cannot break cryptographic primitives (standard Dolev-Yao model [10]).
3. **Malicious User**: A subset of users may collude to violate system properties. We require security as long as at least one honest user exists in each session.
4. **Blockchain Reorganization**: Temporary blockchain reorganizations may occur but eventually finalize (within expected confirmation depth).

We assume users maintain secure control of their cryptographic keys and that cryptographic primitives (Ed25519, X25519, ChaCha20-Poly1305, Poseidon, Groth16) are secure.

3.3 Security Goals

- **Confidentiality**: Expense details remain confidential from the server and non-member users.
- **Integrity**: Tampering with expense history is cryptographically detectable.
- **Authenticity**: Each expense event is verifiably created by an authorized session member.
- **Availability**: The system remains operational despite server unavailability (clients can verify using blockchain state).

4 Architecture

4.1 Overview

Figure 1 presents the high-level architecture. The system follows a strict layered architecture with five distinct layers:

1. **Client Layer**: Web and mobile applications perform cryptographic operations (encryption, signing) before sending data
2. **API Layer**: HTTP handlers validate and route requests using Elysia.js framework
3. **Business Logic Layer**: Services orchestrate operations without direct database access
4. **Data Access Layer**: Stores handle database operations (PostgreSQL, Redis) while Repositories manage external services (Solana, Light Protocol)
5. **Storage & Blockchain**: Persistent storage and immutable blockchain commitments

This separation ensures clean architecture: handlers never contain business logic, services never access databases directly, and stores are isolated from external service calls. All encryption occurs client-side, ensuring the API handles opaque encrypted data throughout.

4.2 Wallet-Based Authentication

Traditional web applications rely on username/password authentication, which creates password management burdens and centralized identity control. We employ wallet-based authentication where users authenticate using cryptographic signatures.

4.2.1 Authentication Protocol

The authentication protocol proceeds as follows:

1. User initiates authentication with public key pk_u
2. Server generates challenge $c = H(\text{nonce} \| \text{timestamp})$
3. User signs challenge: $\sigma = \text{Sign}(sk_u, c)$
4. Server verifies $\text{Verify}(pk_u, c, \sigma) = 1$
5. Upon successful verification, server issues JWT token τ encoding user identity and expiration

We use Ed25519 [11] for signatures due to its small signature size (64 bytes), fast verification, and resistance to side-channel attacks. JWT tokens [12] enable stateless session management with cryptographic integrity.

4.2.2 Session Management

JWT tokens include:

- User identifier (blockchain address)
- Issuance and expiration timestamps
- HMAC signature under server secret key

Token expiration enforces session timeout (typically 24 hours), requiring re-authentication. This limits the window of vulnerability from token compromise while maintaining reasonable user experience.

4.3 End-to-End Encryption

4.3.1 Cryptographic Primitives

We employ the following cryptographic primitives:

- **Signatures**: Ed25519 for authentication and non-repudiation
- **Key Agreement**: X25519 Diffie-Hellman for session key distribution
- **Authenticated Encryption**: ChaCha20-Poly1305 for expense data encryption
- **Password-Based Key Derivation**: PBKDF2 with SHA-256 for key encryption key derivation
- **Key Hardening**: BLAKE2b for additional key strengthening
- **Hashing**: BLAKE2b for deterministic secret derivation, Poseidon for MMR commitments (ZK-friendly)
- **Zero-Knowledge Proofs**: Groth16 for batched MMR update verification with constant proof size (256 bytes)

Each user maintains two keypairs: (1) Ed25519 keypair $(sk_{\text{sig}}, pk_{\text{sig}})$ for signatures, and (2) X25519 keypair $(sk_{\text{enc}}, pk_{\text{enc}})$ for encryption.

Key Storage and Recovery: Private keys are stored server-side in encrypted form to enable multi-device access and account recovery:

1. User provides password pwd during registration
2. System derives deterministic secret: $s = \text{BLAKE2b}(\text{BLAKE2b}(pwd), \text{authId})$
3. Key Encryption Key (KEK) is derived using a two-stage process:
   (a) PBKDF2 derivation: $k_{\text{PBKDF2}} = \text{PBKDF2}(s, \text{salt}, n)$ with SHA-256, where n is a high iteration count following current security recommendations
   (b) BLAKE2b hardening: $k_{\text{KEK}} = \text{BLAKE2b}(k_{\text{PBKDF2}}, \text{salt})$
4. Private keys are encrypted: $c_{\text{keys}} = \text{ChaCha20-Poly1305}(k_{\text{KEK}}, sk_{\text{sig}} \| sk_{\text{enc}})$
5. Server stores: $\{c_{\text{keys}}, \text{salt}, pk_{\text{sig}}, pk_{\text{enc}}\}$

This approach provides password-based key escrow without revealing keys to the server: only users with knowledge of pwd can derive k_{KEK} and decrypt private keys. The high iteration count combined with BLAKE2b hardening makes brute-force attacks computationally expensive while maintaining compatibility with standard browser APIs.

4.3.2 Session Key Distribution

Each session σ has a symmetric session key K_σ used to encrypt all expense data. The key distribution protocol is:

This protocol enables efficient member addition without requiring key rotation or re-encryption of historical data. Each member independently decrypts their copy of K_σ using their private key.

Algorithm 1 Session Key Distribution

```
1: Session Creation:
2: Creator generates $K_\sigma \xleftarrow{\$} \{0,1\}^{256}$
3: Creator encrypts $c_{\text{creator}} = \text{Enc}_{pk_{\text{enc}}^{\text{creator}}}(K_\sigma)$
4: Server stores $c_{\text{creator}}$ associated with session $\sigma$
5:
6: Member Addition:
7: Admin retrieves and decrypts $K_\sigma = \text{Dec}_{sk_{\text{enc}}^{\text{admin}}}(c_{\text{admin}})$
8: Admin encrypts for new member: $c_{\text{new}} = \text{Enc}_{pk_{\text{enc}}^{\text{new}}}(K_\sigma)$
9: Server stores $c_{\text{new}}$ for new member
10: New member retrieves and decrypts $K_\sigma = \text{Dec}_{sk_{\text{enc}}^{\text{new}}}(c_{\text{new}})$
```

4.3.3 Expense Encryption

Expense events are encrypted as:

$$(c, \tau, \text{iv}) = \text{ChaCha20-Poly1305}(K_\sigma, m, \text{iv}) \tag{1}$$

where m is the expense plaintext (amount, description, metadata), c is the ciphertext, τ is the authentication tag, and iv is a random nonce. The authenticated encryption ensures both confidentiality and integrity.

4.4 Event Sourcing

4.4.1 Event Model

We represent expense state using event sourcing [9]. Each expense e is defined by a sequence of immutable events $E_e = [v_1, v_2, \ldots, v_k]$ where each event v_i is one of:

- CREATE($data$): Initial expense creation
- UPDATE($data$): Modification of expense attributes
- DELETE(): Logical deletion (soft delete)

Current expense state is computed by reducing the event sequence:

$$\text{state}(e) = \text{reduce}(E_e, \emptyset) \tag{2}$$

This approach provides:

1. Complete audit trail: All modifications are permanently recorded
2. Temporal queries: State can be reconstructed at any historical point
3. Conflict resolution: Causal ordering prevents inconsistencies

The event sourcing model extends uniformly to all entity types. Expense events include split mode metadata (equal, proportional, or exact amounts) enabling flexible cost distribution among participants. Reimbursement events—both on-chain SOL transfers and off-chain traditional payments—follow the same CREATE/UPDATE/DELETE lifecycle, ensuring that all settlement activity participates in the same cryptographic commitment pipeline.

4.4.2 Causal Ordering

Events are causally ordered using Lamport timestamps [14] implicit in the event chain. Each event (except CREATE) references its predecessor:

$$v_i = (type, data, prev = \mathrm{id}(v_{i-1}), timestamp) \tag{3}$$

This creates a linked structure that enables detection of missing or reordered events. The server cannot reorder events without invalidating the causal chain.

4.5 Cryptographic Commitments

4.5.1 Triple-Root System

We maintain three independent cryptographic commitment structures within each session using Merkle Mountain Ranges (MMR) with Poseidon hashing for ZK-friendly verification:

1. **Members Root** (R_M): Merkle Mountain Range of session member public keys, enabling efficient member addition/removal with $O(\log n)$ proofs and supporting historical membership queries (e.g., proving a member belonged to a session at a specific event count).

2. **Expenses Root** (R_E): Merkle Mountain Range of expense event hashes (CREATE, UPDATE, DELETE), providing efficient append-only commitment with $O(\log n)$ proofs for expense history verification. The hierarchical two-level structure enables sessions to scale beyond 32 events.

3. **Refunds Root** (R_R): Merkle Mountain Range of all settlement events, encompassing both on-chain refunds (SOL transfers between members on Solana) and off-chain reimbursements (traditional payment methods such as bank transfers or cash). Both settlement types are committed to the same MMR, enabling unified cryptographic verification of payment history regardless of settlement mode. Claims—logical debt records derived from balance computation—exist as an application-layer construct above the settlement MMR, tracking outstanding debts until resolved through either settlement mode.

All three roots are stored in a single compressed Solana account, requiring only 96 bytes total on-chain storage (32 bytes per root) regardless of session size or activity level. The MMR-based approach with Poseidon hashing enables both efficient incremental updates without reconstructing entire trees and ZK-friendly circuit verification.

Each MMR root can be updated independently via zero-knowledge proofs. Each ZK proof transaction can include 1-32 events (circuit limit) and costs 0.000045 SOL ($0.009) regardless of event count within that range, maintaining complete privacy (events are not revealed in proofs). This separation of concerns provides several benefits: (1) membership changes don't trigger re-verification of financial data, (2) expense modifications are isolated from payment history, and (3) each commitment can be updated independently at different frequencies based on activity patterns.

4.5.2 Merkle Mountain Range with Poseidon

All three roots (members, expenses, refunds) are implemented as Merkle Mountain Ranges using Poseidon hash function. We use Poseidon instead of traditional cryptographic hashes (SHA-256, Keccak256) because it is specifically designed for zero-knowledge circuits, requiring 8-10x fewer constraints in SNARK circuits.

Implementation Note: The Solana program exclusively uses Poseidon-based ZK proofs for all MMR root updates. Early development versions included Keccak-256 based MMR verification code, but this has been completely removed in favor of the ZK-only architecture with Poseidon hashing, which is specifically optimized for SNARK circuits and enables the hierarchical MMR structure.

Poseidon Hashing: For event leaves, we use:

$$\text{leaf}_i = \text{Poseidon}(0, e_i) \tag{4}$$

where prefix 0 distinguishes leaves. Internal nodes are computed as:

$$\text{node}(left, right) = \text{Poseidon}(1, left, right) \tag{5}$$

where prefix 1 provides domain separation preventing second-preimage attacks. Poseidon operates over a prime field (BN254 scalar field) making it ideal for Groth16 proofs.

4.5.3 MMR Structure and Operations

Standard Merkle Trees are inefficient for append-only logs because adding an element requires reconstructing the entire tree and updating the root. We employ Merkle Mountain Ranges [5,6], which maintain multiple tree "peaks" enabling efficient appends.

An MMR for n events consists of $\lfloor \log_2 n \rfloor + 1$ peaks corresponding to the binary representation of n. For example, $n = 5 = 0b101$ yields two peaks covering 4 and 1 events respectively.

The MMR root is computed by combining peaks left-to-right using Poseidon. For k peaks, the root is:

$$R_E = \begin{cases} \text{peak}_1 & \text{if } k = 1 \\ \text{Poseidon}(1, \text{peak}_1, R'_E) & \text{if } k > 1 \end{cases} \tag{6}$$

where R'_E is recursively computed from peaks $[\text{peak}_2, \ldots, \text{peak}_k]$.

Append Operation: Adding event e_{n+1} requires only:

1. Compute leaf hash: $h = \text{Poseidon}(0, e_{n+1})$
2. If $n + 1$ is a power of 2: merge all peaks into single peak
3. Otherwise: create new rightmost peak
4. Recompute root from updated peaks

This operation is $O(\log n)$ compared to $O(n)$ for full tree reconstruction.

Verification Protocol: To verify inclusion of event e_i:

1. Verifier has root R_E (from blockchain)
2. Prover provides sibling hashes along path from e_i to its peak
3. Verifier recomputes peak hash
4. Verifier recomputes root from peaks and compares to R_E

Proof size is $O(\log n)$ hashes, requiring $\approx 32 \log_2 n$ bytes.

4.6 Settlement Mechanisms

Naami implements a dual settlement model that accommodates both on-chain and off-chain payment methods, unified under the same cryptographic commitment framework.

4.6.1 Dual Settlement Model

Settlement of outstanding balances occurs through two complementary modes:

1. **On-Chain Refunds**: Direct SOL transfers between session members executed on the Solana blockchain. These transactions are natively verifiable and immediately committed to the Refunds MMR (R_R).

2. **Off-Chain Reimbursements**: Traditional payment methods (bank transfers, cash, mobile payments) recorded as reimbursement events. Off-chain reimbursements follow the same event sourcing pattern as on-chain refunds and are integrated into the Refunds MMR when anchored, providing equivalent cryptographic guarantees once committed.

4.6.2 Claims Protocol

A **claim** $c = (\text{debtor}, \text{creditor}, \text{amount}, \text{status})$ represents an outstanding debt between two session members, derived from balance computation across all expense events. The claims protocol operates as follows:

1. **Balance Computation**: After each expense event, the system computes per-member balances based on expense amounts and split modes (equal, proportional, or exact amounts).

2. **Debt Derivation**: Outstanding debts are derived from the balance matrix, identifying which members owe amounts to others.

3. **Settlement**: Debtors settle claims through either on-chain refunds or off-chain reimbursements. Both modes create settlement events recorded in the Refunds event sequence R.

4. **Verification**: All settlement events participate in the same MMR commitment pipeline, enabling cryptographic verification of the complete settlement history.

The high-level flow is: expenses $\rightarrow$ balance computation $\rightarrow$ claims derivation $\rightarrow$ settlement (on-chain or off-chain) $\rightarrow$ cryptographic anchoring via Refunds MMR.

4.6.3 Protocol Fee Collection

An on-chain fee vault collects protocol fees on settlement transactions. These fees are distinct from Solana network fees (transaction costs paid to validators) and represent protocol-level revenue used to sustain development and operations. The fee vault is controlled by a dedicated fee authority account. The full economic model, including fee rates, redistribution mechanisms, and governance of fee parameters, is described in the companion tokenomics paper [25].

5 Technical Innovations

5.1 Hierarchical Two-Level MMR Architecture

A key innovation in Naami is the hierarchical two-level MMR structure that enables sessions to scale beyond the 32-event circuit limit while maintaining constant proof sizes and ZK verification guarantees.

5.1.1 Circuit Constraint and Scalability Challenge

Our Groth16 ZK circuit (`mmr_full.circom`) is designed to prove MMR transitions for up to 32 events maximum. This constraint arises from:

- Circuit complexity: larger circuits require exponentially more constraints
- Proving time: currently 2.5-3 seconds for 32 events; would increase to 10+ seconds for 64 events
- Witness generation memory: scales with circuit size

Without a hierarchical structure, sessions would be permanently limited to 32 events total. The hierarchical approach addresses this fundamental scalability constraint.

5.1.2 Two-Level MMR Structure

The solution organizes events hierarchically across two levels:

1. **Level 0 (Batch Level)**: Events are organized into batches of up to 32 events each. Each batch forms a complete MMR with its own root R_{batch}. Batch roots are computed using Poseidon hashing: $R_{\text{batch}} = \text{MMR}_{\text{Poseidon}}([h_1, \ldots, h_k])$ where $k \leq 32$.
2. **Level 1 (Global Level)**: Batch roots are aggregated into a global MMR where each completed batch root serves as a leaf. The global root R_{global} combines all batch roots: $R_{\text{global}} = \text{MMR}_{\text{Poseidon}}([R_{\text{batch}_1}, R_{\text{batch}_2}, \ldots, R_{\text{batch}_n}])$.

Epoch System: We introduce an epoch counter to track which level the on-chain root represents:

- **Epoch 0**: First batch (1-32 events). The on-chain root stores the batch root directly.
- **Epoch $k > 0$**: Multiple batches exist. The on-chain root stores the global root aggregating $k + 1$ batch roots.

ZK Proof Types: The same circuit (`mmr_full.circom`) generates proofs for both levels, but the interpretation differs:

- **Batch Proofs** (epoch unchanged): For updates within the current batch (total events ≤ 32). Circuit private inputs are individual event hashes.
- **Global Root Proofs** (epoch increments): For batch rollover (total events > 32). Circuit private inputs are batch roots treated as "events" in the global MMR.

Circuit Design: Our Circom circuit (`mmr_full.circom`) proves the statement: "I know events E such that building an MMR from E yields root R_{new} and the previous state had root R_{prev} with fewer events." The circuit:

1. Takes as private inputs: event hashes $[h_1, \ldots, h_{32}]$ (padded with zeros)
2. Takes as public inputs: R_{prev}, R_{new}, previous event count, final event count
3. Implements MMR construction with Poseidon hashing
4. Verifies: $R_{\text{new}} = \text{MMR}(\text{events}[0 : \text{finalCount}])$
5. Verifies: $R_{\text{prev}} = \text{MMR}(\text{events}[0 : \text{previousCount}])$ (consistency check)

Algorithm 2 Hierarchical MMR Root Update with ZK Proof

```
Input: Session state (current epoch $\epsilon$, synced event count), new events to add
Output: ZK proof $\pi$, new root $R_{\text{new}}$, updated epoch $\epsilon'$

// Off-chain: Determine proof type
totalEvents ← syncedEventCount + |newEvents|

if totalEvents $\leq 32$ then
    // Batch proof (level 0): Building or extending first batch
    $\epsilon' \leftarrow 0$ (epoch unchanged)
    Compute event hashes: $h_i = \text{Poseidon}(0, e_i)$ for all events [1..totalEvents]
    Build batch MMR: $R_{\text{new}} = \text{MMR}_{\text{Poseidon}}([h_1, \ldots, h_{\text{totalEvents}}])$
    Circuit private inputs: event hashes $[h_1, \ldots, h_{32}]$ (padded with zeros)
else
    // Global root proof (level 1): Batch rollover
    $\epsilon' \leftarrow \epsilon + 1$ (epoch increments)
    Organize events into completed batches of 32 + current partial batch
    Compute batch roots: $R_{\text{batch}_i}$ for each completed batch
    Build global MMR: $R_{\text{new}} = \text{MMR}_{\text{Poseidon}}([R_{\text{batch}_1}, \ldots, R_{\text{batch}_n}])$
    Circuit private inputs: batch roots as "events" (padded with zeros)
endif

// Generate ZK proof
Public inputs: $\{R_{\text{prev}}, R_{\text{new}}, \text{previousCount}, \text{finalCount}, \epsilon'\}$
$\pi \leftarrow$ Groth16.Prove(mmr_full.circom, public, private)

// On-chain: Submit and verify
Submit transaction with proof $\pi$ and public inputs (340 bytes total)
Solana program verifies proof and updates session root and epoch
```

On-Chain Verification: The Solana program embeds the Groth16 verification key at compile time via `build.rs`, enabling efficient on-chain verification in the `update_*_root_with_zk` instructions. Verification requires:

- Deserializing the 256-byte Groth16 proof (uncompressed)
- Deserializing 84-byte public inputs (previous root, final root, event counts, epoch)
- Groth16 pairing check (constant time, $\approx$1ms on Solana)
- Root consistency validation

Key Properties:

- **Constant transaction cost**: Each ZK proof transaction costs 0.000045 SOL ($0.009) regardless of event count (1-32 events per proof)
- **Unbounded scaling**: The two-level structure enables sessions to grow beyond 32 events. Batch roots (level 0) are aggregated into a global MMR (level 1)
- **Privacy preservation**: Event details are never revealed on-chain (zero-knowledge property)
- **Constant proof size**: All proofs are 256 bytes (Groth16 property)

5.1.3 Epoch Transition System

The epoch counter tracks which MMR level the on-chain root represents:

Table 1: Epoch Transition Examples

Previous Count	Final Count	Current Epoch	Expected Epoch	Epoch Changes	Type
0	10	0	0	No	Batch update
10	32	0	0	No	Batch update (complete)
32	33	0	1	Yes	Global root (rollover)
33	64	1	1	No	Batch update (2nd batch)
64	65	1	2	Yes	Global root (rollover)

The epoch increments when: (1) final event count exceeds 32, OR (2) previous event count is a multiple of 32 AND a new batch begins. This deterministic rule enables verification of correct hierarchical MMR updates.

ZK Proof Types:

- **Batch Proofs** (epoch unchanged): For updates within the current batch ($\leq$ 32 events total). Circuit processes individual event hashes.
- **Global Root Proofs** (epoch increments): For batch rollover or sessions with $>$ 32 events. Circuit processes batch roots as "events", proving the global root transition.

This hierarchical approach enables:

- **Unbounded scaling**: Sessions can grow beyond 32 events without circuit redesign
- **Constant proof size**: Both batch and global proofs remain 256 bytes (Groth16 property)
- **Efficient updates**: Only the current batch needs reconstruction; completed batches remain immutable
- **Logarithmic overhead**: Global root updates require $O(\log n)$ batch root hashes, where n is the number of batches

5.2 Hierarchical MMR Synchronization

The hierarchical structure requires careful synchronization between off-chain event logs and on-chain commitments. We maintain:

- **syncedEventCount**: Number of events already anchored on-chain
- **syncedEpoch**: Last epoch that was synchronized (distinguishes batch vs global updates)
- **syncedGlobalRoot**: Global root at the time of last synchronization (used as previousRoot for ZK proofs)

Synchronization Protocol:

1. New events are added to the current batch (level 0)
2. When batch reaches 32 events, it's marked as completed and a new batch starts
3. Global root (level 1) is recalculated from all batch roots
4. ZK proof is generated:
    - If totalEvents ≤ 32: Batch proof with event hashes
    - If totalEvents > 32: Global root proof with batch roots as "events"
5. Proof is anchored on-chain, updating syncedEventCount and syncedEpoch

This protocol ensures that:

- Completed batches never change (immutability)
- Only the current batch requires reconstruction (efficiency)
- Global root proofs can batch multiple batch completions (cost efficiency)
- Clients can verify integrity by comparing local global root with on-chain root

5.3 Event Accumulation with Sliding Window Debounce

A critical optimization in Naami is the event accumulation mechanism that dramatically reduces blockchain transaction costs by batching events before generating ZK proofs.

5.3.1 Problem: Real-Time Anchoring Cost

Without accumulation, each event would trigger immediate ZK proof generation and blockchain anchoring:

- User creates expense $\rightarrow$ API generates proof (2.5s) $\rightarrow$ Submit to Solana ($0.009)
- High costs: 100 expenses = 100 transactions = $0.900
- Poor user experience: waiting for proof generation blocks response
- Inefficient: many small proofs instead of fewer large proofs

Algorithm 3 Event Accumulation with Sliding Debounce

```
Input: New event $e$, session $\sigma$, entity type $T$
State: Pending workflow $W_{\sigma,T}$ with scheduled time $t_{\text{scheduled}}$

// Add event to hierarchical MMR cache immediately (user gets instant feedback)
Update MMR cache with event $e$

// Debounce workflow scheduling
$t_{\text{now}} \leftarrow$ CurrentTime()
delay $\leftarrow$ 6 hours

if $W_{\sigma,T}$ exists then
  if $t_{\text{now}} - t_{\text{scheduled}} <$ delay then
    // Still within debounce window: cancel and reschedule
    Cancel workflow $W_{\sigma,T}$
    Schedule new workflow $W'_{\sigma,T}$ at $t_{\text{now}} +$ delay
  else
    // Workflow will execute soon: skip (avoid duplicate)
    Return
  endif
else
  // No pending workflow: schedule new one
  Schedule workflow $W_{\sigma,T}$ at $t_{\text{now}} +$ delay
endif
```

5.3.2 Solution: Sliding Window Debounce

We implement a sliding window debounce mechanism with 6-hour inactivity threshold:

Key Properties:

- **Scope:** Per-session and per-entity-type (members, expenses, refunds are independent)
- **Sliding window:** Each new event resets the 6-hour timer
- **Cancellation:** Previous scheduled proofs are cancelled when new events arrive
- **Instant feedback:** MMR cache updates immediately; users don't wait for blockchain anchoring

Example Timeline:

- t_0: User creates expense 1 $\rightarrow$ Schedule proof at $t_0 + 6h$
- $t_0 + 2h$: User creates expense 2 $\rightarrow$ Cancel previous, schedule at $t_0 + 8h$
- $t_0 + 3h$: User creates expenses 3-10 $\rightarrow$ Each reschedules to 6h after last event
- $t_0 + 9h$: 6 hours of inactivity $\rightarrow$ Proof generated for all 10 expenses
- Result: **1 proof** ($0.009) instead of 10 proofs ($0.090)

Cost Impact: For active sessions where events cluster temporally, this mechanism reduces transaction costs by orders of magnitude. A session with 100 expenses created over a few hours requires only 1-2 proofs instead of 100.

Limitation - Maximum Events per Entity Type: The hierarchical structure supports up to 32 batch roots per global proof. Since each batch contains up to 32 events:

- Maximum events per entity type (members, expenses, refunds): $32 \times 32 = 1024$ events total
- This is a lifetime limit per session, not per time period
- Sessions requiring more than 1024 expenses/refunds/members would need circuit redesign
- In practice, this limit is sufficient for typical expense tracking (e.g., a year-long session with 3 expenses/day = 1095 events would exceed the limit)

5.4 Zero-Knowledge Compression

Traditional Solana accounts require rent payments proportional to account size (≈0.00079 SOL per 113 bytes, or $0.16 at $200/SOL). For 1000 sessions, this totals $160 in storage costs alone.

We leverage Light Protocol's ZK Compression [18], which stores account state in a Merkle tree with only the root on-chain. Account updates require validity proofs but eliminate rent entirely.

Cost Comparison (per session):

- Traditional account: $0.16 storage + $0.01 operations = $0.17
- Compressed account: $0.00 storage + $0.01 operations = $0.01

This achieves **94% cost reduction** in storage while maintaining full verifiability through ZK proofs.

5.5 Caching Strategy

Reconstructing MMR from database events is expensive ($O(n \log n)$ operations). We employ a Redis-based caching layer storing:

- Hierarchical MMR structure (completed batches, current batch, global root)
- Total event count and current epoch
- Synced state (syncedEventCount, syncedEpoch, syncedGlobalRoot)
- Last update timestamp for monitoring

Cache entries have 30-day TTL, balancing memory usage with reconstruction cost. Sessions remain cached for an extended period to minimize database load for active sessions, while inactive sessions are automatically evicted. If cache is invalidated or expired, the complete MMR can be reconstructed from persistent storage (PostgreSQL), reducing reconstruction from seconds to milliseconds for cached sessions.

6 Security Analysis

6.1 Confidentiality

Theorem 1 (Expense Confidentiality): *Under the IND-CPA security of ChaCha20-Poly1305, an adversary with access to the server database cannot distinguish expense plaintexts.*

Proof sketch: All expense data is encrypted with session key K_σ. The server stores (c, τ, iv) tuples but never learns K_σ (encrypted asymmetrically for each member). Since ChaCha20-Poly1305 is IND-CPA secure [16] with random nonces, ciphertexts leak no information about plaintexts. □

6.2 Integrity

Theorem 2 (Event Integrity): *Under the collision resistance of Poseidon hash and the soundness of Groth16, an adversary cannot modify expense history without detection.*

Proof sketch: Each expense event e is hashed into MMR: $h_e = \text{Poseidon}(0, e)$. The MMR root R_E is a cryptographic commitment to all events. Modifying event $e' \neq e$ produces different hash $h_{e'} \neq h_e$, changing the MMR root. Since R_E is stored on immutable blockchain and updated via ZK proofs, adversaries face two barriers: (1) Collision resistance of Poseidon ensures they cannot find e' with $h_{e'} = h_e$, and (2) Groth16 soundness ensures they cannot generate valid proofs for incorrect root transitions. Clients detect tampering by comparing locally computed roots with blockchain roots. □

6.3 Authenticity

Theorem 3 (Event Authenticity): *Under the SUF-CMA security of Ed25519, an adversary cannot create events attributed to honest users.*

Proof sketch: Each event includes signature $\sigma = \text{Sign}(sk_u, h_e)$ where h_e commits to event data. Ed25519 is SUF-CMA secure [17], meaning adversary cannot forge signatures without secret key sk_u. Server verifies signatures before accepting events. □

6.4 ZK Proof Soundness

Theorem 4 (ZK Proof Integrity): *Under the soundness of Groth16, an adversary cannot submit false MMR root transitions to the blockchain.*

Proof sketch: Each on-chain root update requires a valid Groth16 proof π demonstrating that $R_{\text{new}} = \text{MMR}(\text{events}[0 : \text{finalCount}])$ and $R_{\text{prev}} = \text{MMR}(\text{events}[0 : \text{previousCount}])$ for some event sequence. Groth16 provides computational soundness [21]: under the discrete logarithm assumption on BN254, the probability that an adversary can generate a valid proof for a false statement is negligible. Therefore, all on-chain roots cryptographically commit to valid MMR constructions, ensuring integrity even when the server generates proofs. □

Privacy Property: Zero-knowledge proofs provide an additional privacy layer beyond E2EE. Even if an adversary observes all blockchain transactions, they learn only:

- Previous and new MMR roots (32-byte hashes)
- Event counts (previous and final)
- Session identifier

They do *not* learn:

- Individual event contents (encrypted client-side)
- Individual event hashes (private inputs to circuit)
- Event ordering within batches
- Event types (member/expense/refund distinguishable only by which root updated)

This provides on-chain privacy orthogonal to E2EE, preventing even passive blockchain observers from learning sensitive metadata.

6.5 Attack Resistance

Server Compromise: If the server is compromised, adversaries gain access to encrypted expense data, encrypted session keys, and encrypted E2EE private keys. However:

- Session keys are encrypted individually for each member using their X25519 public key
- E2EE private keys (Ed25519 and X25519) are stored encrypted with a Key Encryption Key (KEK) derived from the user's password via PBKDF2 with a high iteration count (following current security recommendations) followed by BLAKE2b hardening
- Without knowledge of users' passwords, adversaries cannot derive the KEK and thus cannot decrypt private keys
- Without private keys, session keys remain undecryptable, protecting all expense data

This defense-in-depth approach ensures that server compromise alone is insufficient—adversaries would additionally require users' passwords. Blockchain roots enable clients to verify data integrity independently, detecting any tampering attempts.

Network Attacks: TLS encryption protects data in transit. Message authentication prevents tampering. Replay attacks are mitigated by nonces and timestamps in signatures.

Member Removal: When a member is removed, they retain local copy of K_σ and can decrypt past expenses. This is acceptable in our threat model (legitimate members should access historical data). Future work may implement key rotation for stronger forward secrecy.

Hardware wallet secured deployment further strengthens the attack surface by ensuring that program upgrade authority and token mint authority are held on a dedicated hardware device with BIP44 key derivation, preventing software-based key extraction.

6.6 Server-Side PII Protection

While end-to-end encryption protects financial data (expense amounts, descriptions, metadata), the system also handles personally identifiable information (PII) that exists outside the E2EE envelope: email addresses used for account lookup, device tokens for push notifications, and user-chosen nicknames. We apply defense-in-depth cryptographic protections to these metadata fields.

6.6.1 Threat Model for PII

PII data differs from expense data in that it must be partially accessible to the server for operational purposes (e.g., email-based user lookup, push notification delivery). Full E2EE is therefore not applicable. Instead, we minimize plaintext exposure through targeted cryptographic techniques.

6.6.2 HMAC-SHA-256 for Email Lookup

Email addresses are never stored in plaintext. Instead, a keyed HMAC is computed:

$$h_{\text{email}} = \text{HMAC-SHA-256}(K_{\text{server}}, \text{normalize}(\text{email})) \tag{7}$$

where K_{server} is a server-side secret key and normalize applies case-folding and whitespace trimming. The HMAC digest h_{email} is stored in place of the email address. User lookup proceeds by computing the HMAC of the query email and matching against stored digests. This provides:

- **Lookup functionality**: Exact-match queries remain possible via HMAC comparison

- **No plaintext exposure**: Database compromise reveals only HMAC digests, which are computationally infeasible to invert without K_{server}
- **Deterministic mapping**: The same email always produces the same digest, enabling indexed lookups

6.6.3 AES-256-GCM for At-Rest Encryption

Device tokens (used for push notifications) and user nicknames are encrypted at rest using AES-256-GCM:

$$(c, \tau, \text{iv}) = \text{AES-256-GCM}(K_{\text{server}}, m, \text{iv}) \tag{8}$$

where m is the plaintext (device token or nickname). Unlike HMAC, AES-256-GCM is a reversible encryption scheme, allowing the server to decrypt these values when needed for operational purposes (sending push notifications, displaying nicknames to session members). The authentication tag τ ensures integrity, detecting any tampering with the ciphertext.

An encrypted resource system provides a generalized framework for at-rest encryption of server-side data, applying consistent cryptographic protections across all PII categories.

6.6.4 Metadata Confidentiality

Theorem 5 (Metadata Confidentiality): *Under the PRF security of HMAC-SHA-256 and the IND-CPA security of AES-256-GCM, an adversary with full database access but without knowledge of K_{server} cannot recover email addresses, device tokens, or nicknames.*

Proof sketch: Email addresses are protected by HMAC-SHA-256, which under PRF security produces outputs indistinguishable from random—an adversary cannot invert digests without K_{server}. Device tokens and nicknames are protected by AES-256-GCM, which under IND-CPA security ensures ciphertexts leak no information about plaintexts. Combined with Theorem 1 (expense confidentiality via E2EE), this establishes a layered defense: financial data is protected by client-side E2EE with session keys, while metadata is protected by server-side cryptography with K_{server}. Full server compromise (database access) is insufficient to recover either category of sensitive data without the corresponding keys. □

This defense-in-depth approach ensures that even in the event of a complete database breach, an adversary obtains only encrypted ciphertexts and HMAC digests, with no path to plaintext PII.

7 Performance Evaluation

7.1 Experimental Setup

We deploy Naami on Solana devnet and measure costs, latency, and ZK proof generation time across various session configurations. Test sessions range from 3 to 10 members with 10 to 50 expenses each. ZK proofs are generated off-chain using snarkjs with Groth16 and the BN254 elliptic curve.

7.2 Cost Analysis

Important Note: All root updates in Naami use ZK proofs exclusively with a hierarchical two-level MMR structure and sliding window debounce mechanism. The system combines two cost optimization strategies:

1. **Hierarchical Structure:** Events are organized into batches (level 0, up to 32 events each), and batch roots are aggregated into a global root (level 1). This enables sessions to scale beyond the 32-event circuit limit.
2. **Sliding Window Debounce:** Events accumulate for 6 hours of inactivity before triggering proof generation. A single global proof can anchor hundreds of events organized across multiple batches, dramatically reducing transaction costs (e.g., 1 proof for 100 events instead of 100 individual proofs).

The key cost insight: *each ZK proof transaction costs 0.000045 SOL ($0.009)* regardless of how many events are included (limited only by circuit constraints: 32 events for batch proofs, 32 batch roots for global proofs).

Table 2: Transaction Costs (SOL and USD at $200/SOL)

Operation	Cost (SOL)	Cost (USD)
Session Creation	0.000058	$0.012
ZK Proof Update (per transaction)	0.000045	$0.009

Note: Each ZK proof transaction costs 0.000045 SOL ($0.009) and can include 1-32 events.

Cost Model: Each ZK proof requires one blockchain transaction costing $0.009. The 6-hour debounce mechanism accumulates events, generating a single proof after inactivity periods.

Small Session (24 events created within 6 hours: 4 members + 20 expenses):

- Session creation: $0.012
- After 6h of member inactivity: 1 proof for 4 members (batch level): $0.009
- After 6h of expense inactivity: 1 proof for 20 expenses (batch level): $0.009
- **Total: $0.030** (2 proofs for 24 events)

Medium Session (100 expenses created within 6 hours):

- Session creation: $0.012
- 100 expenses organized into 4 batches (32 + 32 + 32 + 4 events)
- After 6h inactivity: 1 global proof anchoring all 4 batch roots: $0.009
- **Total: $0.021** (1 proof for 100 events)
- **Cost per event: $0.00021** (amortized)

Large Active Session (500 expenses over 3 days with activity bursts):

- Session creation: $0.012
- Day 1: 150 expenses → 6h inactivity → 1 global proof (5 batches): $0.009
- Day 2: 200 expenses → 6h inactivity → 1 global proof (7 batches): $0.009
- Day 3: 150 expenses → 6h inactivity → 1 global proof (5 batches): $0.009
- **Total: $0.039** (3 proofs for 500 events)
- **Cost per event: $0.000078** (amortized)

Comparison with Alternatives:

- **Ethereum (direct storage):** \$10-\$20 per session → Naami achieves **99.8% cost reduction**
- **Solana (uncompressed accounts):** \$0.38 per session → Naami achieves **94% cost reduction**
- **Without debounce mechanism:** 100 events = 100 proofs = \$0.900 → Debounce achieves **97.7% cost reduction**

The combination of hierarchical MMR structure, ZK compression, and sliding window debounce enables practical blockchain-based expense tracking with costs orders of magnitude lower than alternatives.

Protocol Fees: In addition to Solana network fees (paid to validators for transaction processing), the protocol collects fees on settlement transactions via an on-chain fee vault. These protocol fees fund ongoing development, infrastructure, and ecosystem growth. The separation between network fees and protocol fees ensures transparency: users can independently verify both cost components on-chain. The complete economic model, including fee rates and governance of fee parameters, is described in the companion tokenomics paper [25].

7.3 Proof Sizes

We distinguish between two types of proofs in Naami:

MMR Inclusion Proofs (for verifying individual events):

Table 3: MMR Inclusion Proof Size Growth

Events (n)	Proof Size (hashes)	Size (bytes)
1	0	0
2	1	32
4	2	64
8	3	96
16	4	128
32	5	160
64	6	192

MMR inclusion proof sizes grow logarithmically: $|\pi| = 32\lceil \log_2 n \rceil$ bytes. Even for 1000 events, proofs require only $32 \times 10 = 320$ bytes.

ZK Batching Proofs (for updating on-chain roots):

Table 4: ZK Proof Characteristics

Component	Size (bytes)
Groth16 Proof (uncompressed)	256
Public Inputs (roots + counts + epoch)	84
Total On-Chain Data per Transaction	**340**
Events per Proof (circuit limit)	1-32
Cost per Proof Transaction	\$0.009

The key advantage of Groth16 is constant proof size: whether proving 1 event or 32 events, the proof is always 256 bytes. Both batch proofs (level 0) and global root proofs (level 1) maintain constant 256-byte proof size due to Groth16 properties, enabling unbounded session scaling through hierarchical aggregation.

7.4 Latency

Event Creation (User-Facing):

- **Client encryption**: < 10ms (ChaCha20-Poly1305 on modern hardware)
- **API processing**: < 50ms (database operations, MMR cache updates)
- **User confirmation**: < 100ms

ZK Proof Generation and Anchoring (Asynchronous):

- **Proof generation**: 2.5-3.0 seconds per proof (constant, regardless of 1-32 events, off-chain)
- **Blockchain submission**: < 100ms
- **Blockchain confirmation**: ≈ 500ms (Solana finality)
- **Total anchoring latency**: $\approx 3 - 4$ seconds (asynchronous)

The key architectural insight is decoupling user-facing operations (< 100ms) from blockchain anchoring (≈ 3s). Users experience instant feedback when creating expenses, while ZK proof generation and anchoring occur asynchronously in the background.

7.5 Scalability

Storage Scaling: Each expense event requires ≈ 500 bytes in PostgreSQL (encrypted payload + metadata). For 10,000 sessions with 20 expenses each: $10{,}000 \times 20 \times 500 = 100$MB, easily manageable.

Computation Scaling: MMR operations are $O(\log n)$, remaining efficient even for large sessions. A session with 1000 expenses requires only $\log_2 1000 \approx 10$ hash operations per proof.

Blockchain Scaling: On-chain storage is $O(1)$ per session (just the roots), regardless of session size. This enables linear scaling to millions of sessions.

8 Discussion

8.1 Limitations

Member Removal: Removed members retain session keys and can decrypt historical data. Implementing key rotation would require re-encrypting all historical events, which is expensive. For most use cases (e.g., shared trip expenses), this is acceptable as members legitimately participated.

ZK Proof Generation Latency: Groth16 proof generation requires 2.5-3 seconds per batch off-chain. While this is acceptable with asynchronous workflows, it introduces anchoring delay. Faster proof systems (e.g., PLONK, Halo2) could reduce latency at the cost of larger proof sizes.

Batch Size Limit: Our circuit supports maximum 32 events per batch due to circuit complexity constraints (Circom compilation time and witness generation memory). While individual batches are limited to 32 events, the hierarchical structure enables unbounded event scaling through global root aggregation.

Session Lifetime Event Limit: The hierarchical MMR structure supports up to 1024 events per entity type (members, expenses, refunds) over the entire session lifetime (32 batches

$\times$ 32 events per batch = 1024 events) due to circuit input limitations. Sessions requiring more events would need circuit redesign to support additional hierarchical levels. In practice, this accommodates most use cases (e.g., 1024 expenses supports daily expense tracking for nearly 3 years at 1 expense/day), though very active corporate sessions might reach this limit.

Anchoring Delay: The 6-hour debounce introduces latency between event creation and blockchain anchoring. While users receive instant feedback ($<$ 100ms) via MMR cache updates, cryptographic anchoring occurs only after inactivity periods. For applications requiring real-time on-chain verification, the debounce delay could be reduced at the cost of increased transaction frequency and higher costs.

Blockchain Dependency: The system requires Solana blockchain availability for verification. During blockchain downtime, clients can still create expenses (stored locally) but cannot cryptographically verify integrity until blockchain access is restored.

8.2 Future Work

Key Rotation: Implement efficient key rotation protocols enabling member removal with forward secrecy.

Light Clients: Develop lightweight mobile clients that can verify Groth16 proofs and MMR proofs without full blockchain node access, enabling trustless verification on resource-constrained devices.

Cross-Chain: Extend architecture to support multiple blockchains (Ethereum, Polygon, Arbitrum) with unified ZK proof system, enabling users to choose based on cost/security preferences.

Formal Verification: Mechanically verify cryptographic properties using tools like Tamarin [19] or ProVerif [20], and formally verify the Circom circuit (`mmr_full.circom`) using tools like circomspect or Aleo's verification frameworks to provide machine-checked guarantees of circuit correctness.

Alternative Proof Systems: Explore newer proof systems like PLONK or Halo2 that offer faster proof generation (sub-second) with slightly larger proof sizes, potentially improving user experience for real-time scenarios.

Adaptive Debounce Mechanisms: Develop adaptive debounce strategies that adjust inactivity thresholds based on session activity patterns. For example, high-frequency sessions could use shorter debounce windows (1-2 hours) to reduce anchoring delay, while low-frequency sessions maintain longer windows (6+ hours) for optimal cost efficiency. Machine learning models could analyze historical session behavior to predict optimal debounce parameters.

Token Economics: The companion tokenomics paper [25] describes the NAAMI token model including tier-based access control, governance mechanisms, and protocol fee redistribution. Future work includes formal economic analysis, simulation of token velocity under various adoption scenarios, and cross-chain token bridging.

8.3 Broader Applications

While this paper focuses on expense tracking, the architectural patterns and cryptographic techniques presented are applicable to various collaborative finance and record-keeping scenarios:

- **Corporate Expense Reports**: Organizations can leverage the E2EE and event sourcing architecture for employee expense report submission and approval workflows, maintaining privacy while ensuring auditability and compliance.

- **Meal Voucher Management**: Restaurant ticket (ticket restaurant) allocation and redemption tracking with cryptographic verification, preventing double-spending while preserving user privacy.

- **Shared Subscriptions**: Groups managing shared subscriptions (streaming services, software licenses) can use the same membership and payment tracking mechanisms with transparent contribution history.
- **Microfinance and Lending Circles**: Community-based lending (tontines, ROSCAs) benefit from transparent commitment tracking without exposing sensitive financial data to centralized authorities.
- **Non-Profit Fundraising**: Charitable organizations can use the architecture for transparent donation tracking with donor privacy, providing cryptographic proof of fund allocation.
- **Supply Chain Cost Allocation**: Multi-party supply chain participants can track and verify cost contributions across complex workflows without revealing proprietary pricing to competitors.

The combination of E2EE, event sourcing, and blockchain commitments provides a general framework for any multi-party collaborative financial record-keeping where privacy, verifiability, and cost-efficiency are essential. The modular architecture allows adaptation to domain-specific requirements while maintaining core security properties.

9 Conclusion

We have presented Naami, a privacy-preserving decentralized expense tracking system that achieves the seemingly conflicting goals of confidentiality, verifiability, and cost-efficiency. By combining wallet-based authentication, group-level end-to-end encryption, encrypted event sourcing, hierarchical zero-knowledge proof verification, sliding window debounce mechanism, and a triple-root cryptographic commitment system, we enable collaborative expense management without trusting centralized servers.

Our key technical contributions—efficient MMR-based append-only commitments with Poseidon hashing, hierarchical two-level MMR architecture enabling unbounded event scaling beyond the 32-event circuit limit, Groth16 zero-knowledge proof verification with constant 256-byte proofs, sliding window debounce with 6-hour inactivity threshold enabling event accumulation, and Light Protocol ZK-compressed storage—demonstrate that blockchain-based privacy-preserving applications can achieve practical performance and costs. With amortized costs as low as \$0.00021 per event (100 events for \$0.021 total), proof generation under 3 seconds, user-facing latencies under 100ms, and support for up to 1024 events per entity type over the session lifetime, Naami provides user experience and cost structure comparable to centralized alternatives while offering superior privacy, verifiability, and on-chain confidentiality through zero-knowledge proofs.

The integration of zk-SNARKs for hierarchical MMR verification represents a novel approach to scaling blockchain-based commitment schemes while maintaining cryptographic verifiability and privacy. Our Poseidon-based MMR implementation with Circom circuits demonstrates the viability of ZK-friendly data structures for practical applications.

We believe this work establishes a foundation for privacy-preserving collaborative finance applications and hope it inspires further research in zero-knowledge proof systems for cryptographic commitment schemes and decentralized systems.

Acknowledgments

We thank the Solana Foundation and Light Protocol team for their support and technical guidance. We also acknowledge the broader cryptography and blockchain research communities whose foundational work made this system possible.

References

[1] Tricount. *Group Expense Manager.* https://www.tricount.com/

[2] Splitwise. *Split Bills and Expenses with Friends.* https://www.splitwise.com/

[3] S. Chen et al. *A Survey on Decentralized Finance (DeFi).* ACM Computing Surveys, 2023.

[4] R. Merkle. *A Digital Signature Based on a Conventional Encryption Function.* CRYPTO 1987.

[5] Grin Developers. *Merkle Mountain Range.* https://docs.grin.mw/wiki/chain-state/merkle-mountain-

[6] P. Todd. *Merkle Mountain Ranges.* https://github.com/opentimestamps/opentimestamps-server/blob

[7] M. Marlinspike and T. Perrin. *The Double Ratchet Algorithm.* Signal Messenger, 2016.

[8] F. Valsorda. *age: A Simple, Modern, and Secure File Encryption Tool.* https://age-encryption.org/

[9] M. Fowler. *Event Sourcing.* https://martinfowler.com/eaaDev/EventSourcing.html

[10] D. Dolev and A. Yao. *On the Security of Public Key Protocols.* IEEE Transactions on Information Theory, 1983.

[11] D. Bernstein et al. *High-Speed High-Security Signatures.* Journal of Cryptographic Engineering, 2012.

[12] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT).* RFC 7519, 2015.

[13] F. Denis. *The Sodium Crypto Library (libsodium).* https://libsodium.gitbook.io/

[14] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 1978.

[15] S. Lucks. *A Failure-Friendly Design Principle for Hash Functions.* ASIACRYPT 2005.

[16] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols.* RFC 8439, 2018.

[17] D. Bernstein. *Ed25519: High-Speed High-Security Signatures.* https://ed25519.cr.yp.to/

[18] Light Protocol. *ZK Compression for Solana.* https://docs.lightprotocol.com/

[19] S. Meier et al. *The TAMARIN Prover for the Symbolic Analysis of Security Protocols.* CAV 2013.

[20] B. Blanchet. *Modeling and Verifying Security Protocols with ProVerif.* Cambridge University Press, 2016.

[21] J. Groth. *On the Size of Pairing-Based Non-interactive Arguments.* EUROCRYPT 2016. https://eprint.iacr.org/2016/260

[22] iden3. *Circom: Circuit Compiler for Zero-Knowledge Proofs.* https://docs.circom.io/

[23] L. Grassi et al. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems.* USENIX Security 2021. https://www.poseidon-hash.info/

[24] E. Ben-Sasson et al. *Zerocash: Decentralized Anonymous Payments from Bitcoin.* IEEE S&P 2014.

[25] Naami Development Team. *NAAMI Token: Tokenomics and Governance*. Companion paper, 2026.

[26] Solana Labs. *SPL Token-2022 Program.* `https://spl.solana.com/token-2022`

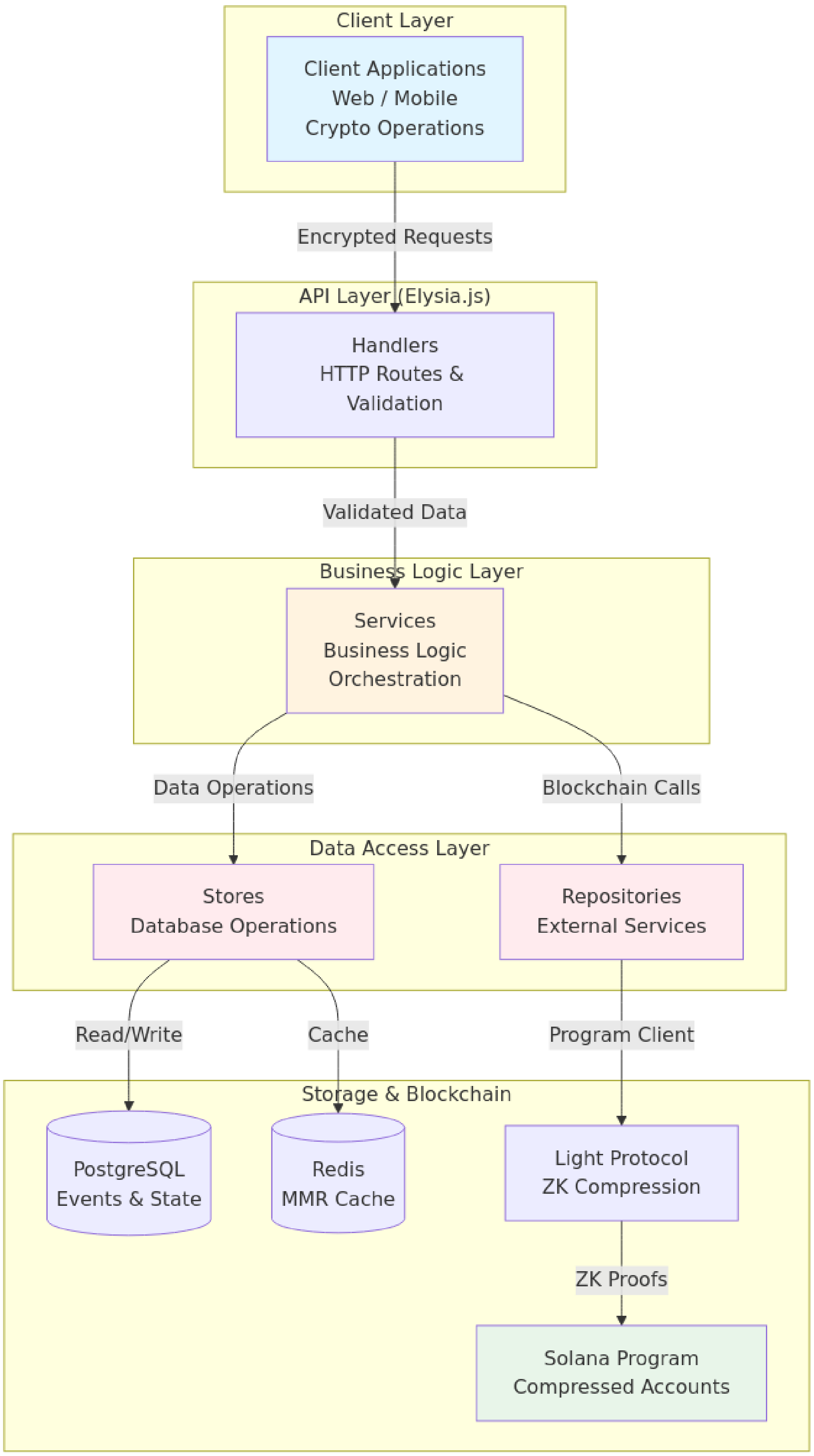

Figure 1: System Architecture: Client applications perform cryptographic operations, the API layer coordinates operations with encrypted data, storage layers persist state, and the blockchain layer maintains verifiable commitments.