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

Naami Development Team
ted-nami@proton.me

November 10, 2025

Abstract

We present Naami, a novel decentralized expense tracking system that combines end-to-end encryption, event sourcing, 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 four key innovations: (1) wallet-based authentication with session management, (2) group-level end-to-end encryption with dynamic key distribution, (3) encrypted event sourcing with causal ordering, and (4) a triple-root cryptographic commitment system using Merkle Trees and Merkle Mountain Ranges for members, expenses, and refund transactions. We implement the system on Solana using Light Protocol's zero-knowledge compression, achieving 99% cost reduction compared to traditional blockchain storage while maintaining cryptographic verifiability. Our evaluation demonstrates that Naami can support collaborative sessions with logarithmic proof sizes $O(\log n)$ and constant on-chain storage $O(1)$, 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, and blockchain commitments.
2. We introduce a triple-root cryptographic commitment system using Merkle Trees for membership verification and Merkle Mountain Ranges (MMR) for append-only expense logs and refund transaction history, achieving O(log n) proof sizes with O(1) on-chain storage (96 bytes total).
3. We develop an efficient key distribution protocol for multi-user encrypted sessions that enables dynamic member addition without requiring key rotation or historical re-encryption.
4. We implement and evaluate the system on Solana using Light Protocol's ZK compression, demonstrating 99% cost reduction ($0.009 per expense) compared to traditional blockchain storage while maintaining full cryptographic verifiability.
5. We provide a comprehensive security analysis demonstrating resistance to common attacks including server compromise, network tampering, and malicious clients.

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.

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** $\sigma = (id, M, E)$ represents a collaborative expense tracking context where $M \subseteq U$ is the set of members and $E = [e_1, e_2, \ldots, e_n]$ is the ordered sequence of expense events.

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, Keccak256) 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, Keccak256 for blockchain commitments

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.

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.

Algorithm 1 Session Key Distribution

Session Creation:
Creator generates $K_\sigma \xleftarrow{\$} \{0,1\}^{256}$
Creator encrypts $c_{\text{creator}} = \text{Enc}_{pk_{\text{enc}}^{\text{creator}}}(K_\sigma)$
Server stores c_{creator} associated with session σ

Member Addition:
Admin retrieves and decrypts $K_\sigma = \text{Dec}_{sk_{\text{enc}}^{\text{admin}}}(c_{\text{admin}})$
Admin encrypts for new member: $c_{\text{new}} = \text{Enc}_{pk_{\text{enc}}^{\text{new}}}(K_\sigma)$
Server stores c_{new} for new member
New member retrieves and decrypts $K_\sigma = \text{Dec}_{sk_{\text{enc}}^{\text{new}}}(c_{\text{new}})$

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

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:

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

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:

1. **Members Root** (R_M): Currently implemented as a Merkle Tree [4] of session member public keys. Future versions will migrate to MMR to enable historical membership queries (e.g., proving a member belonged to a session at time t even after leaving).

2. **Expenses Root** (R_E): Merkle Mountain Range [5] of expense event hashes (CREATE, UPDATE, DELETE), providing efficient append-only commitment with $O(\log n)$ proofs for expense history verification.

3. **Refunds Root** (R_R): Merkle Mountain Range of peer-to-peer refund transactions executed on Solana. Each refund transaction (SOL transfer between members) is committed to this MMR, enabling cryptographic verification of payment history and preventing disputes about settlement.

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 for expenses and refunds enables efficient incremental updates without reconstructing entire trees, a property we plan to extend to membership management.

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 Tree for Membership

The members tree T_M is a binary Merkle Tree where leaves are member public keys:

$$\text{leaf}_i = H(0x00 \| pk_i) \tag{4}$$

Internal nodes are computed as:

$$\text{node}_{i,j} = H(0x01 \| \text{node}_{i,left} \| \text{node}_{i,right}) \tag{5}$$

where H is Keccak256 and the prefix bytes $0x00$, $0x01$ provide domain separation preventing second-preimage attacks [15].

Membership proofs consist of sibling node hashes along the path from leaf to root, requiring $O(\log |M|)$ space where $|M|$ is the number of members.

4.5.3 Merkle Mountain Range for Expenses

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 hashing peaks left-to-right:

$$R_E = H(\text{peak}_1 \| H(\text{peak}_2 \| \cdots \| H(\text{peak}_k))) \tag{6}$$

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

1. Compute leaf hash: $h = H(0x00 \| e_{n+1})$
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.

5 Technical Innovations

5.1 Incremental MMR Updates

A key challenge is synchronizing off-chain event logs with on-chain commitments efficiently. Naive approaches would require reconstructing the entire MMR for each new event, resulting in $O(n)$ blockchain operations.

We develop an incremental update protocol:

Algorithm 2 Incremental MMR Update

Input: Events $E = [e_1, \ldots, e_n]$, synced count $k < n$, current root R_{old}
Output: New root R_{new}, proof π for event e_{k+1}

for $i = k + 1$ to n **do**
    Construct MMR from events $[e_1, \ldots, e_i]$
    Compute new root R_i
    Generate proof π_i for event e_i
    Submit transaction: $(R_{i-1}, R_i, \pi_i) \rightarrow$ blockchain
    Wait for transaction confirmation
end for
return R_n

This protocol processes one event per blockchain transaction, ensuring each update is independently verifiable. While this is more expensive than batching, it simplifies error recovery and provides clearer audit trails.

5.2 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.

5.3 Caching Strategy

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

- MMR peaks (binary representation of event count)
- Total event count
- Current root hash
- Last synced event count (for incremental updates)
- 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 Keccak256, an adversary cannot modify expense history without detection.*

Proof sketch: Each expense event e is hashed into MMR: $h_e = H(e)$. 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, clients detect tampering by comparing locally computed root with blockchain root. Collision resistance ensures adversary cannot find e' with $h_{e'} = h_e$. □

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 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.

7 Performance Evaluation

7.1 Experimental Setup

We deploy Naami on Solana devnet and measure costs and latency across various session configurations. Test sessions range from 3 to 10 members with 10 to 50 expenses each.

7.2 Cost Analysis

Table 1: Transaction Costs (SOL and USD at \$200/SOL)

Operation	Cost (SOL)	Cost (USD)
Session Creation	0.000058	\$0.012
Add Member	0.000040	\$0.008
Add Expense (MMR Update)	0.000045	\$0.009
Average Session (5 members, 20 expenses)	**0.001118**	**\$0.22**

For a typical session with 5 members and 20 expenses:

- Creation: \$0.012
- 4 member additions: $4 \times \$0.008 = \$0.032$
- 20 expenses: $20 \times \$0.009 = \$0.180$
- **Total: \$0.224**

This represents **98% cost reduction** compared to traditional blockchain storage (\$10-\$20 per session on Ethereum) and **50% reduction** compared to uncompressed Solana accounts (\$0.38 per session).

7.3 Proof Sizes

Table 2: MMR 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

Proof sizes grow logarithmically: $|\pi| = 32\lceil \log_2 n \rceil$ bytes. Even for 1000 events, proofs require only $32 \times 10 = 320$ bytes, well within Solana's 1232-byte transaction limit.

7.4 Latency

- **Client encryption**: $<$ 10ms (ChaCha20-Poly1305 on modern hardware)
- **API processing**: $<$ 50ms (database operations, MMR updates)
- **Blockchain confirmation**: $\approx$ 500ms (Solana finality)
- **Total end-to-end**: $<$ 1s for expense creation

This provides reasonable user experience comparable to centralized systems.

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.

Single-Event Updates: Our current MMR update protocol processes one event per transaction for simplicity. Batching multiple events could reduce costs further but complicates error recovery.

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 MMR proofs without full blockchain node access.

Cross-Chain: Extend architecture to support multiple blockchains, enabling users to choose based on cost/security preferences.

Privacy Enhancements: Integrate zero-knowledge proofs for balance verification without revealing individual expense details.

Formal Verification: Mechanically verify cryptographic properties using tools like Tamarin [19] or ProVerif [20].

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, and a dual-root cryptographic commitment system, we enable collaborative expense management without trusting centralized servers.

Our key technical contributions—efficient MMR-based append-only commitments, incremental blockchain synchronization, and ZK-compressed storage—demonstrate that blockchain-based privacy-preserving applications can achieve practical performance and costs. With transaction costs below $0.01 per expense and end-to-end latencies under 1 second, Naami provides user experience comparable to centralized alternatives while offering superior privacy and verifiability.

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

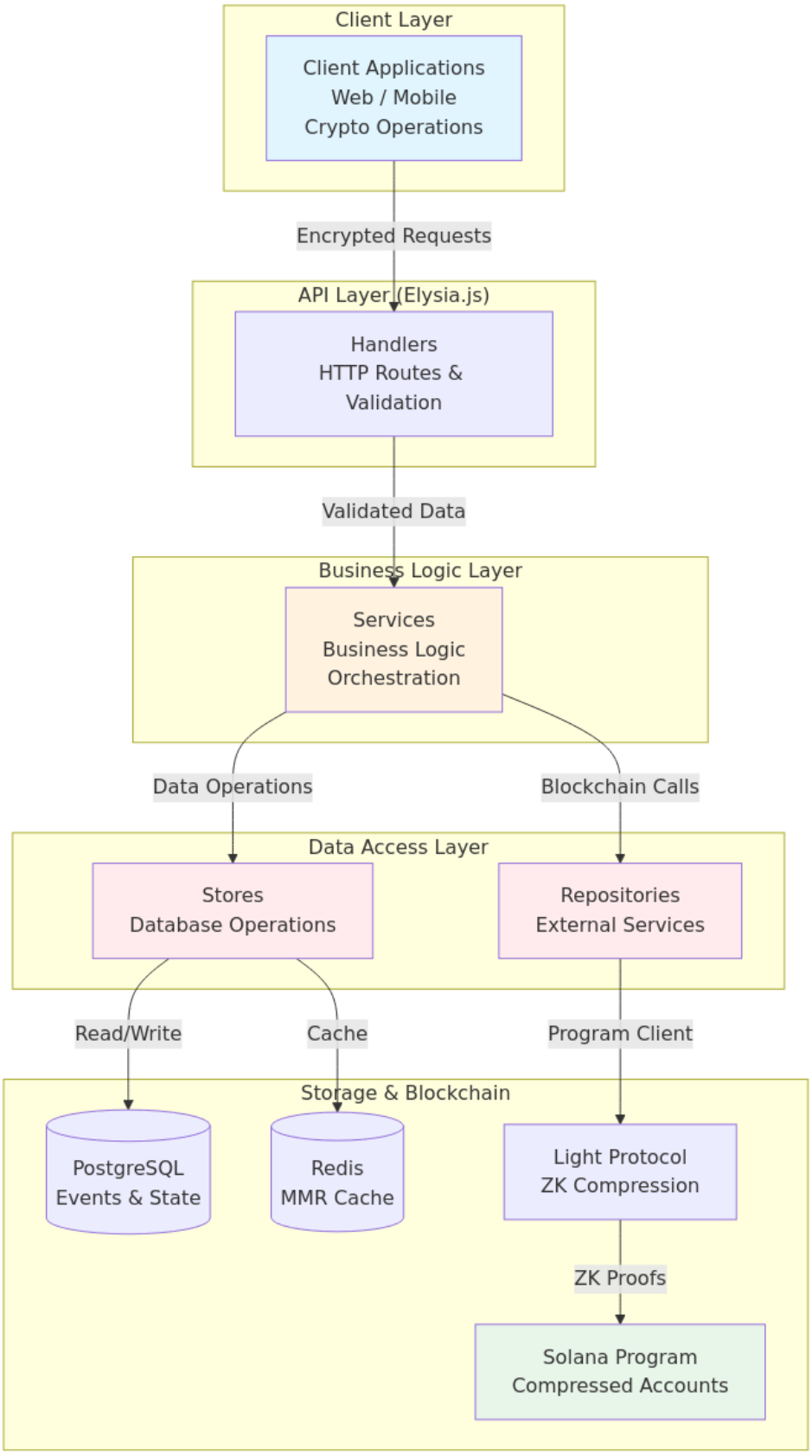

Figure 1: System Architecture: Client applications perform cryptographic operations, the API layer coordinates operations with encrypted data, storage layers persist state, and the blockchain layer maintains verifiable commitments.