UNIVERSITÀ DI PISA

Report on SplitWise - Trust Project

Peer to Peer and Blockchain, 2024-2025

Professor: Laura Emilia Maria Ricci

Student: Suranjan Kumar Ghosh

November 22, 2025

1. Introduction

Managing shared expenses among groups often suffers from a lack of transparency and cumbersome settlement processes. The Trust-Splitwise project addresses this by implementing a decentralized expense-sharing application on the Ethereum blockchain. The suite, comprised of `Splitwise.sol` and `TrustToken.sol`, leverages smart contracts to provide a secure, transparent, and efficient system for collaborative finance.

By moving the core logic on-chain, the system offers a tamper-proof history of all transactions. To overcome the high gas costs of complex computations, the project employs a hybrid model where a computationally intensive debt simplification algorithm runs off-chain, with its results being securely verified and applied on-chain. This report details the architectural decisions, security model, and performance analysis of this implementation.

2. Architectural Design

Contracts Overview

- **TrustToken.sol**: ERC-20 compliant token contract
  - Fixed ETH-to-token minting rate
  - Reentrancy protection for mint function
- **Splitwise.sol**: Core expense management
  - Group creation and membership management
  - Expense registration with split types
  - Debt simplification workflow
  - Token-based settlement

Group Lifecycle

1. **Creation**: Caller invokes `createGroup()`, becoming initial member
2. **Joining**: Via `inviteMember()` (existing members) or `joinGroup()` (open groups with zero debt)
3. **Locking**: Activated during debt simplification to prevent state changes
4. **Debt Simplification**: Off-chain computation + on-chain verification
5. **Settlement**: Debt payment via ERC-20 token transfers

Architectural Features and Security

3.1 Group Locking Mechanism

A ‘Group‘ struct with an ‘isLocked‘ state, controlled by ‘onlyWhenUnlocked‘ modifier, prevents state changes (e.g., adding members) during critical operations like debt simplification. This ensures data consistency.

```
modifier onlyWhenUnlocked(uint256 gid) {
    require(!groups[gid].isLocked, "Group is locked");
    _;
}
```

3.2 Off-Chain Calculation & Commit-Verify Pattern

Debt simplification runs off-chain for gas efficiency. Integrity is ensured by a commit-verify pattern:

- **Greedy Algorithm**: A quadratic-time ($O(n^2)$) greedy algorithm iteratively cancels pairwise debts to produce simplified edges based on the Splitwise specification.
- **Future Optimization**: A more advanced data-structure-aware approach (e.g., maintaining max-heap and min-heap views over creditor/debtor net balances) could asymptotically approach $O(n \log n)$ because heap-based priority queues support $\log n$ inserts/extracts. Coordinating two heaps would let the off-chain simplifier repeatedly match the largest creditor with the largest debtor, emit a transfer, update both heaps, and stop once residual balances drop below a tolerance threshold. The on-chain verifier would still only need to validate at most n signed-balance edges, so gas exposure remains bounded while the off-chain workload scales more gracefully as group size grows.[1]
- **Commit Phase**: A ‘keccak256‘ hash of the simplified ‘edges‘ data is committed on-chain.

[1] https://en.wikipedia.org/wiki/Priority_queue

```
function commitSimplification(bytes32 edgesHash) public {
    submittedSimplifications[edgesHash] = true;
}

```

- **Apply Phase**: ‘applySimplification‘ verifies submitted edges against the committed hash before updating debts, preventing tampering.

3.3 Expense Hashing and Salting

To ensure the integrity and uniqueness of every expense registered, the system employs a cryptographic salting mechanism. Each Group struct contains an expenseNonce, which is an incrementing counter unique to that group. This nonce is included in the data that is hashed to create the unique expenseHash for each transaction.

```
// The expenseNonce acts as a salt in the hashing process.
bytes32 expenseHash = keccak256(abi.encode(ExpenseData({
...
nonce: groups[gid].expenseNonce++
})));
```

3.4 Reentrancy Protection

Critical functions (`settleDebt`, `mint`) use OpenZeppelin’s ‘ReentrancyGuard‘. This prevents re-entry attacks, safeguarding against unauthorized fund transfers and ensuring function atomicity.

```
function settleDebt() public nonReentrant { ... }
function mint() public payable nonReentrant { ... }
```

3.5 SafeERC20 Integration

OpenZeppelin’s ‘SafeERC20‘ library is integrated for all ERC-20 token interactions. This provides secure wrappers for token functions, mitigating common vulnerabilities and ensuring reliable transfers.

```
using SafeERC20 for IERC20;
// ...
token.safeTransferFrom(msg.sender, creditor, amount);
```

3.6 Data Structures and Optimization

Data management utilizes ‘uint256‘ for amounts and custom ‘structs‘ (e.g., ‘Group‘) for organized data storage. Structs help prevent ”stack too deep” errors and can optimize storage packing. ‘unchecked‘ arithmetic further improves gas efficiency.

```
struct Group {
    address[] members;
    mapping(address => bool) isMember;
    bytes32 debtGraphHash;
    uint256 expenseNonce;
    bool isLocked;
}
mapping(uint256 => mapping(address => mapping(address => uint256)))
   public debts;
```

Key Security Aspects and Controls

- **Arithmetic Over/Underflow**: Mitigated by Solidity 0.8.28’s built-in checks, with ‘unchecked‘ blocks reserved for provably safe operations.
- **Phishing via `tx.origin`**: Prevented by zero usage of `tx.origin` and strict `msg.sender` authentication.
- **Front-Running**: While expense registration faces MEV risk, debt simplification utilizes a commit-reveal scheme for control.
- **Gas Limit DoS**: Controlled by a 50-member group size cap and the bounded $O(n^2)$ simplification workload.
- **Timestamp Attacks**: Not applicable, as no time-dependent logic exists within the contract.

Gas Cost Evaluation

Hardhat testing provided insights into the gas efficiency of ‘Splitwise‘ and ‘TrustToken‘ contracts.

Deployment and Core Operations

- **Deployment Addresses**: ‘TrustToken‘ at ‘0x5Fb...‘, ‘Splitwise‘ at ‘0xe7f...‘.
- **Deployment Costs**: ‘Splitwise‘ averaged 1.85M gas (1.9% block limit), ‘TrustToken‘ 0.63M gas (0.6% block limit).

Operation	Min Gas	Max Gas	Avg Gas
Group Creation	-	-	156,967
Invite Member	79,932	79,944	79,941
Register Expense	61,607	132,687	93,280
Commit Simplification	-	-	72,966
Apply Simplification	96,488	1,426,696	366,582
Settle Debt	48,972	50,376	50,090
Token Approval	46,371	46,683	46,664
Token Minting	53,083	70,183	55,856

Table 2.1: Detailed Gas Costs for Key Operations

Layer-2 Deployment Outlook

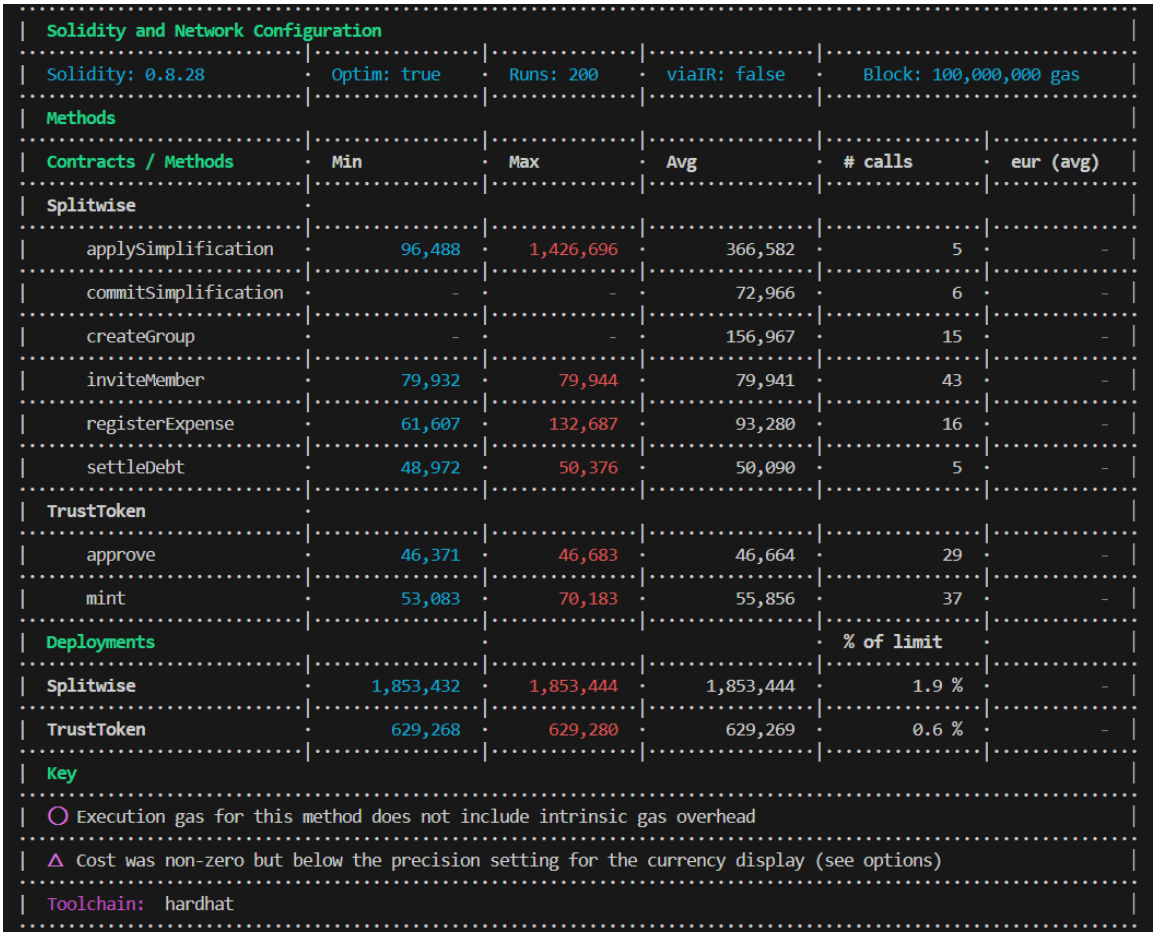
While the current prototype targets Ethereum mainnet compatibility, even routine interactions can become cost-prohibitive when base fees spike. Ethereum’s ecosystem now includes ”Ethereum-backed” layer-2 networks where the observed average transaction cost is roughly $0.002 compared to $0.14 on layer 1, with many rollups advertising around $0.01 user-facing

fees.[2] Migrating the Splitwise contracts to an optimistic-rollup chain (e.g., Base or OP Mainnet) would therefore reduce the per-call cost of ‘applySimplification‘ by two orders of magnitude without sacrificing Ethereum’s settlement guarantees. Operationally, this migration requires only redeploying the contracts plus updating the Hardhat network configuration and addresses consumed by the CLI scripts because the project already adheres to the EVM execution model.

Key Observations

- ‘applySimplification‘ shows high variability (96k-1.4M gas), reflecting its complexity based on the debt graph.
- Routine operations like ‘inviteMember‘ (80k gas) and ‘settleDebt‘ (50k gas) are consistent and efficient.
- The system effectively handles complex debt graphs with up to 20 members, demonstrated by stress tests (1.85s execution).

[2] https://ethereum.org/en/layer-2/



Solidity and Network Configuration				
Solidity: 0.8.28	Optim: true	Runs: 200	viaIR: false	Block: 100,000,000 gas
Methods				
Contracts / Methods	**Min**	**Max**	**Avg**	**# calls**	**eur (avg)**
Splitwise					
applySimplification	96,488	1,426,696	366,582	5	-
commitSimplification	-	-	72,966	6	-
createGroup	-	-	156,967	15	-
inviteMember	79,932	79,944	79,941	43	-
registerExpense	61,607	132,687	93,280	16	-
settleDebt	48,972	50,376	50,090	5	-
TrustToken					
approve	46,371	46,683	46,664	29	-
mint	53,083	70,183	55,856	37	-
Deployments				**% of limit**	
Splitwise	1,853,432	1,853,444	1,853,444	1.9 %	-
TrustToken	629,268	629,280	629,269	0.6 %	-
Key					
○ Execution gas for this method does not include intrinsic gas overhead					
△ Cost was non-zero but below the precision setting for the currency display (see options)					
Toolchain: hardhat					

Figure 2.1: Gas Consumption Report from Hardhat Testing

3. User Manual

This section outlines deployment and interaction with the TRUST Splitwise contracts using the provided Hardhat scripts.

Deployment

Run the following commands in sequence to deploy and set up the system on a local network:

```
npx hardhat run scripts/deploy.js           --network localhost
npx hardhat run scripts/create_group.js     --network localhost
npx hardhat run scripts/add_expense.js      --network localhost
npx hardhat run scripts/simplify_debts.js   --network localhost
npx hardhat run scripts/check_balance.js    --network localhost
npx hardhat run scripts/cleanup.js          --network localhost
```

Interaction

Each script invokes the corresponding smart-contract functions as follows:

- **deploy.js**
  - Deploys `TrustToken` (mintable ERC-20)
  - Deploys `Splitwise` (expense manager)
- **create_group.js**
  - Calls `splitwise.createGroup()` → emits `GroupCreated`
- **add_expense.js**
  - Invokes `splitwise.inviteMember()` (to add participants)

  - Mints via `trustToken.mint()`
  - Approves via `trustToken.approve()`
  - Logs an expense with `splitwise.registerExpense()`
- **simplify_debts.js**
  - Off-chain computes simplified edges
  - Calls `splitwise.commitSimplification()`
  - Calls `splitwise.applySimplification()`
- **check_balance.js**
  - Reads each user's token balance via `trustToken.balanceOf()`
- **cleanup.js**
  - Removes `tmp-contract-addresses.json` to reset for future runs

End-to-End Scenario at a Glance

1. Suranjan creates group → invites Kenny, Aashish, Natesh.
2. Groceries: Suranjan logs 100 TRST exact split → Kenny 20, Aashish 30, Natesh 10.
3. Gas: Kenny logs 50 TRST exact split → Aashish 20, Natesh 20.
4. Simplify: off-chain plan {Aashish→Suranjan 50; Natesh→Kenny 20; Natesh→Suranjan 10}.
5. Commit & apply on-chain via two calls.
6. Natesh settles his 20+10 debts → final balances verify zero.

4. Conclusion

The Trust-Splitwise implementation achieves:

- Secure expense tracking with cryptographic guarantees
- Gas-efficient operations through optimization
- Scalable architecture supporting 20+ members
- Robust security against common vulnerabilities