T-CLSAG Implementation Security Audit

Joshua Babb*

July 28, 2025

This report describes the findings of a code audit of Salvium's T-CLSAG implementation.

Contents

1 Overview

1.1 Introduction

T-CLSAG (Two-scalar CLSAG) is a cryptographic signature scheme implemented in Salvium to support CARROT (Cryptonote Address on Rerandomizable RingCT Output Transactions) addressing, which uses dual-scalar secrets for enhanced privacy features. The implementation extends the existing CLSAG

*CypherStack

signature system to handle transactions with dual-key CARROT addresses while maintaining anonymity, linkability, and unforgeability properties. Because it protects sensitive cryptographic operations and financial transactions, rigorous scrutiny of the mathematical correctness and security implementation is essential.

1.2 Audit goals & methodology

- Evaluate mathematical compliance with the T-CLSAG specification.
- Inspect cryptographic implementation for correctness and security properties.
- Analyze multi-input transaction support and aggregation mechanisms.
- Review integration with existing RingCT and blockchain validation systems.
- Manual static analysis; no formal proofs or exhaustive testing campaigns.

2 Scope

Commit snapshot: Salvium codebase on `carrot-integration` branch as of commit `e5c9b05`. Reviewed directories:

- `salvium/src/ringct/`, `salvium/src/carrot_core/`
- `salvium/src/carrot_impl/`, `salvium/src/wallet/`
- `salvium/tests/unit_tests/ringct.cpp`

Analysis focused on core T-CLSAG functions `TCLSAG_Gen()`, `proveRctTCLSAGSimple()`, `verRctTCLSAGSimple()`, and `genRctSimpleCarrot()`.

3 Summary of findings

- Implementation uses signature array approach rather than single aggregated signature as specified.
- Mathematical operations are correctly implemented for single-input T-CLSAG signatures.
- All cryptographic security properties are maintained across the signature array architecture.
- Domain separator reuse from CLSAG represents a code hygiene issue rather than a critical security vulnerability.

- Multi-input transaction support achieved through external aggregation of individual signatures.
- Integration with blockchain validation and wallet systems is comprehensive and correct.

4 Technical Verification Details

4.1 Code Structure Verification

The T-CLSAG implementation follows a clear architectural pattern:

- **Core signature generation**: `TCLSAG_Gen()` implements the cryptographic protocol with dual-scalar support for CARROT addresses.
- **RingCT integration**: `proveRctTCLSAGSimple()` wraps the core function for transaction-level usage.
- **Verification**: `verRctTCLSAGSimple()` implements the corresponding verification algorithm with identical mathematical structure.
- **Transaction building**: `genRctSimpleCarrot()` orchestrates transaction creation using a signature array approach.

4.2 Mathematical Correctness Verification

Key mathematical operations verified in the implementation:

- **Key image computation**: $I = x \cdot H_p(K)$ correctly implemented in `hwdev.clsag_prepare_carrot()` via `rct::scalarmultKey(I,H,p)` where `H = H_p(K)` and `p` represents the private key x.
- **Challenge computation**: Proper hash chain construction using `HASH_KEY_CLSAG_ROUND` domain separator.
- **Dual-scalar signatures**: Both s_x and s_y components computed via `hwdev.clsag_sign()` and `hwdev.clsag_sign_y()`.
- **Verification equations**: Correct reconstruction of challenge values in verification function.

4.3 Security Implementation Verification

Security-critical aspects confirmed:

- **Domain separation**: Consistent use of CLSAG domain separators (though T-CLSAG-specific separators would be preferable).
- **Randomness handling**: Proper generation and use of blinding factors.

- **Input validation**: Comprehensive size and consistency checks in `TCLSAG_Gen()` function.
- **Hardware device support**: Integration with hardware security modules through `hw::device` interface.

5 Detailed findings

5.1 Architectural Deviation from Specification

Problem The implementation creates an array of individual T-CLSAG signatures rather than a single signature with internal multi-input aggregation as described in the specification. Analysis of `genRctSimpleCarrot()` in `rctSigs.cpp:1609-1619` reveals a loop that generates separate signatures for each input:

```
for (size_t i = 0 ; i < inamounts.size(); i++) {
  rv.p.TCLSAGs[i] = proveRctTCLSAGSimple(full_message,
    rv.mixRing[i], inSk[i].x, inSk[i].y, inSk[i].mask,
    a[i], pseudoOuts[i], index[i], hwdev);
}
```

The specification (t-clsag.tex:121) describes aggregation formulas $W_i = \sum_{j=1}^{m} H_n(T_j, R, L_1, \ldots, L_m) \cdot K_{i,j}$ for creating a single signature covering multiple inputs, while the implementation processes each input independently.

Recommendation Consider documenting the architectural choice in code comments to clarify the deviation from specification. Both approaches achieve equivalent security properties, but the difference should be explicitly acknowledged. It may be beneficial to add specification references explaining why the signature array approach was chosen over internal aggregation.

Status This represents a valid architectural decision rather than an implementation error. The signature array approach maintains all required cryptographic properties while potentially offering implementation simplicity benefits.

5.2 Domain Separator Reuse from CLSAG

Problem The T-CLSAG implementation reuses domain separators from the existing CLSAG implementation. Analysis of `rctSigs.cpp` shows usage of `HASH_KEY_CLSAG_AGG_0`, `HASH_KEY_CLSAG_AGG_1`, and `HASH_KEY_CLSAG_ROUND` constants. These constants are defined in `cryptonote_config.h:357-359` as:

```
const unsigned char HASH_KEY_CLSAG_ROUND[] = "CLSAG_round";
const unsigned char HASH_KEY_CLSAG_AGG_0[] = "CLSAG_agg_0";
const unsigned char HASH_KEY_CLSAG_AGG_1[] = "CLSAG_agg_1";
```

While this does not create practical security vulnerabilities due to structural differences in hash inputs and mutually exclusive protocol usage, it violates cryptographic hygiene best practices for domain separation.

Recommendation Recommended fix is to define T-CLSAG specific domain separators such as `HASH_KEY_TCLSAG_AGG_0`, `HASH_KEY_TCLSAG_AGG_1`, and `HASH_KEY_TCLSAG_ROUND`. This change can be implemented during routine maintenance without affecting security or functionality.

Status Low priority code hygiene issue. The reuse does not pose immediate security risks due to protocol isolation and different hash input structures.

5.3 Mathematical Equivalence in Single-Input Case

Problem While the implementation differs architecturally from the specification, analysis reveals mathematical equivalence for single-input scenarios. The `mu_P` and `mu_C` aggregation scalars computed in `rctSigs.cpp:439-440` are calculated as:

```
mu_P = hash_to_scalar(mu_P_to_hash);
mu_C = hash_to_scalar(mu_C_to_hash);
```

where `mu_P_to_hash` contains the domain separator `HASH_KEY_CLSAG_AGG_0` followed by public keys $P[0..n-1]$ and commitments $C[0..n-1]$, and `mu_C_to_hash` contains `HASH_KEY_CLSAG_AGG_1` followed by the same public keys, key image I, auxiliary point D, and commitment offset. For single-input transactions ($m = 1$), these produce results mathematically equivalent to the specification's $W_i = H_n(T_1, R, L_1) \cdot K_{i,1}$ and $\tilde{W} = H_n(T_1, R, L_1) \cdot L_1$ when the hash inputs are structured identically. However, this equivalence has not been formally documented or proven within the codebase.

Recommendation Consider adding mathematical documentation demonstrating the equivalence between the implementation approach and specification formulas for the single-input case. Specifically, document how the implementation's computation:

```
mu_P * P[i] + mu_C * C[i]
```

relates to the specification's aggregation formula
$W_i = \sum_{j=1}^{m} H_n(T_j, R, L_1, \ldots, L_m) \cdot K_{i,j}$ when $m = 1$. The equivalence holds because both approaches compute a weighted combination of the same cryptographic elements, but use different organizational structures.

5.4 Security Property Preservation

Problem While the implementation uses a different architecture than specified, analysis confirms that all required cryptographic security properties are

maintained. Anonymity is preserved through individual signature anonymity, linkability is maintained via deterministic key image computation $L_j = x_j \cdot H_p(K_j)$, and unforgeability is ensured through the challenge-response protocol requiring knowledge of both x and y components. However, formal security proofs for the signature array approach are not provided.

Recommendation Consider developing formal security arguments demonstrating that the signature array approach maintains equivalent security properties to the specification's single-signature model. It may be beneficial to document the security property preservation explicitly in technical documentation.

Status Security properties verified through analysis, but formal documentation of the security equivalence between approaches would strengthen confidence in the implementation choice.

5.5 Integration Quality and Test Coverage

Problem Analysis reveals comprehensive integration with blockchain validation systems and wallet transaction building processes. The implementation correctly integrates with T-CLSAG transaction types through `is_rct_tclsag(rv.type)` checks and provides proper serialization support via the `rv.p.TCLSAGs` vector. Test coverage in `tests/unit_tests/ringct.cpp:302-483` includes comprehensive T-CLSAG signature generation, verification, and edge case handling with tests for invalid signatures, tampered components, and boundary conditions. The test suite validates both `proveRctTCLSAGSimple()` and `verRctTCLSAGSimple()` functions extensively. However, specific multi-input test scenarios that exercise the signature array architecture are limited.

Recommendation Consider expanding test coverage to include explicit multi-input transaction scenarios that validate the signature array approach. It may be beneficial to add performance benchmarks comparing single-input and multi-input transaction processing to ensure scalability characteristics are acceptable.

Status Integration quality is excellent, but enhanced testing of multi-input scenarios would provide additional confidence in the implementation's robustness across various transaction patterns.