CARROT `carrot_core` Implementation Audit

Joshua Babb*

February 17, 2026

This report describes the findings of a code audit of the C++ implementation of the CARROT addressing scheme for Monero in the `carrot_core` directory of `https://github.com/jeffro256/monero/tree/carrot_core` as of the commit hash `d9842e89`.

Contents

*Cypher Stack

1 Overview

1.1 Introduction

CARROT (Cryptonote Address on Rerandomizable RingCT Output Transactions) is an addressing scheme for Monero that upgrades the legacy CryptoNote addressing system while preserving backward compatibility. It introduces a dual-scalar key hierarchy, rerandomizable one-time addresses using generators G and T, three-byte view tags for accelerated scanning, Janus attack resistance via anchor-based ephemeral key recomputation, and burning bug resistance via input-context binding.

The `carrot_core` directory implements the core protocol logic: key derivation, address construction, enote building, the scan algorithm, output set finalization, and hardware device abstraction.

1.2 Audit goals & methodology

- Evaluate mathematical compliance of `carrot_core` functions against the CARROT specification.
- Inspect cryptographic implementation for correctness and security properties.
- Verify the security properties specified in the CARROT document (Janus resistance, burning bug resistance, scan binding, forward secrecy, and subgroup protection).
- Manual static analysis and testing, but no formal proofs.

2 Scope

Commit snapshot: `pq_secure_ki` branch of the `jeffro256/carrot` CARROT specification repository, `carrot_core` branch of the `jeffro256/monero` Monero repository. Reviewed directories:

- `carrot/`
- `monero/src/carrot_core/`
- `carrot-rs/carrot-crypto/src/` (Rust library, cross-reference only)

3 Summary of results

- The security properties defined in the specification were found to be present in the implementation.

- Both C++ and Rust use Blake2b keyed mode per RFC 7693; the specification notation uses concatenation syntax. Both implementations appear structurally consistent with the specification, though equivalence is not proven here.
- The enote scan algorithm tracks the specification closely; one specification step (Step 18) is not included but is mathematically redunant.
- Step 18 of the specification (prime-order subgroup membership check on $K_s^{j\prime}$) is not included in either the C++ or the Rust implementation, but it is mathematically redundant.
- All domain separator constants in `config.h` match the specification byte-for-byte.

4 Technical verification details

Hashing functions in `hash_functions.cpp` call a common `hash_base()` that configures Blake2b with `"Monero"` personalization in keyed mode per RFC 7693. `derive_scalar` reduces a 64-byte hash output modulo the Ed25519 subgroup order.

4.1 Key hierarchy and address derivation

Key derivation functions in `account_secrets.cpp` match the specification for all secret-key derivations, and the public-key formulas match Section 5.3. Subaddress derivation follows the three-stage chain in Section 6.1 (preimage_1, preimage_2, scalar).

4.2 Enote construction

Enote utility functions in `enote_utils.cpp` implement specification Sections 7.1–7.9: ephemeral key derivation, ECDH, view tag, sender-receiver secret, output extensions, one-time address construction and recovery, amount blinding factor, encryption masks, and Janus anchor special.

4.3 Scan algorithm verification

The enote scan algorithm (specification Section 8.1) is implemented across `scan.cpp`, `scan_unsafe.cpp`, `enote_utils.cpp`,and `device_ram_borrowed.cpp`, distributed across coinbase, external, and internal entry points. Step 18 is not included; see Section 5.2.

4.4 Domain separators and hash functions

All domain separator constants in `config.h` were verified against the specification. The C++ constants include coinbase extension separators, address index preimages, and a generate-image preimage.

4.5 Security property verification

- **Janus attack resistance**: Normal Janus recomputation derives D_e' and verifies it against D_e; special Janus provides a keyed-hash fallback.
- **Burning bug resistance**: Input context is bound into the sender-receiver secret derivation.
- **Enote scan binding**: Different k_v or s_{vb} yield different view tags and blinding factors.
- **Forward secrecy**: Internal enotes use symmetric s_{vb} (no ECDH); external enotes rely on DDH.
- **Small subgroup protection**: K_o is validated via `rct::isInMainSubgroup()` at output finalization.

4.6 Output rules, device layer, and cross-implementation

Output finalization enforces minimum outputs, mandatory self-send, K_o subgroup membership, uniqueness, and deterministic sorting. Coinbase outputs also check K_o subgroup membership.

Four abstract device interfaces in `device.h` delegate all secret-key operations through virtual dispatch and include virtual destructors.Concrete RAM-backed implementations live in `device_ram_borrowed.h/.cpp`.

Cross-implementation differences observed include coinbase domain separators, address index staging (C++ three-step vs. Rust two-step), exception granularity, and device image key coverage.

5 Observations

5.1 Blake2b keyed mode vs. specification notation

The specification uses concatenation notation for hash inputs. Both C++ and Rust use Blake2b keyed mode per RFC 7693. This is structurally different, but intended to implement the same domain-separated hashing semantics.

5.2 Step 18 subgroup check

Step 18 of the specification requires $K_s^{j\prime}$ to be in the prime-order subgroup. This check is not present in either the C++ or Rust implementation.

The check is mathematically redundant: K_o is validated in the prime-order subgroup at output finalization, and the extension components are subgroup elements derived from scalars and generators. By group closure, $K_s^{j\prime}$ remains in the subgroup.

5.3 Coinbase extension path divergence

The C++ implementation uses dedicated coinbase domain separators for extension scalars. The Rust reference uses the standard extension separators together with a clear commitment (blinding factor of 1). The two implementations therefore use different domain separator strings and different inputs for coinbase extension derivation.

5.4 Comment text

The comment im `derive_bytes_3` states a 2-byte output, but the function produces 3 bytes. The code is correct but the comment is not.

6 Conclusion

The `carrot_core` library closely tracks the CARROT specification. Key derivations, enote constructions, scan algorithms, and domain separators were checked for consistency with the specification. The only protocol-level deviation identified was the exclusion of Step 18 (redundant by construction). Outside of the scope of `carrot_core` itself, a divergence regarding the domain separator used in the coinbase extension path was also observed between the C++ `carrot_core` and the Rust `carrot-rs`.