Audit of `serai/networks/monero/`

Joshua Babb[*] Brandon Goodell[*] Rigo Salazar[*]

August 14, 2025

Contents

[*]Cypher Stack

1 Introduction

We review the Rust implementation of FROSTLASS, the `/serai-dex/serai/networks/monero/` directory in commit `48db06f` by `kayabaNerve` of the GitHub repository at `github.com/serai-dex/serai`. This directory contains a Rust implementation of Monero wallet functionality, together with a new approach to using FROST for threshold signing. All files, folders, and subfolders of `/serai-dex/serai/networks/monero/` are in scope for the audit, except not the subfolder `/serai-dex/serai/networks/monero/verify-chain/`.

This document is structured as follows:

- Introduction and suggested actions.
- General findings; negative, neutral, and positive.
- Repository structure.
- Crates overviews.
- One section detailing each crate.

1.1 Suggested Actions

The audit identified one issue that warrants concrete remediation, one area for improvement, and two lower-priority items that may be deferred.

A1. Harden transcript construction for CLSAG and Bulletproof$^+$.
The transcripts in `ringct/clsag/src/lib.rs` lines 96–123 and `ringct/bulletproofs/src/plus/transcript.rs` lines 8–17 follow the on-chain Monero protocol, but future integrators could mis-attribute their scope. Add documentation explaining which transaction fields are hashed elsewhere, so auditors can distinguish implementor responsibilities from upstream Monero protocol requirements.

A2. Tighten RPC and decoy-distribution validation (clarify scope).
The existing code already verifies block hashes, heights and Epee headers. A small improvement would be to also bound the maximum length accepted by `read_vec` in `monero-io`. Note that this is primarily a defense-in-depth improvement rather than a bug-fix.

Deferred / Documentation-Only Items:

- **Cofactor clearing.** The current `INV_EIGHT` $\to$ `*8` approach (e.g. in `primitives/src/lib.rs`) correctly matches upstream Monero's torsion-clearing. No code changes are strictly needed, but add a comment citing relevant torsion-safety references to prevent accidental removal.
- **Keccak256 vs. XOF.** Continuing to use Keccak256 preserves consensus compatibility. Switching to an XOF (e.g. SHAKE-256) could offer minor efficiency benefits if a future protocol upgrade is pursued. In the interim, document the historical reasons and trade-offs in `generators/hash_to_point.rs`.

2 Summary of Key Findings and Recommendations

The audit discusses a total of **7 issues** across the code-base.

1. **Bound the allowable length in `read_vec`.**
   `io/src/lib.rs` lines 216–219 accept an arbitrary length prefix. Allow callers to supply a maximum length (or use a compile-time constant) so a hostile stream cannot exhaust memory.

2. **Document variable-time helpers.**
   Functions such as `non_adjacent_form` and `multiexp_vartime` are intentionally variable-time and must be used only with public data. A comment beside each helper would make that contract explicit.

3. **CLSAG multisig mask channel returns a panic on misuse.**
   The documented `unwrap()` in `ringct/clsag/src/multisig.rs` is safe in normal flow, but promoting it to an explicit `Result<...>` (or `expect("mask not set")`) would surface protocol-order mistakes as recoverable errors.

4. Keccak256 vs. SHA3/XOF — see Section 1.1.

5. Mixed error-handling idioms — migrate to a uniform `Result<T,E>` where convenient.

6. Documentation gaps for cofactor clearing — add one shared comment referencing torsion-safety literature.

7. Sparse adversarial tests — malformed-input cases would further strengthen robustness.

Positive Practices

- Constant-time primitives via `curve25519-dalek`. (e.g. `generators/src/lib.rs` line 10)
- Systematic zeroisation of secret data using `zeroize`. (e.g. `primitives/src/lib.rs` line 82)
- Batch-verification logic that aborts on first non-identity result. (`bulletproofs/src/batch_verifier.rs` lines 23–53)
- Thread-safe lazy initialisation with `LazyLock`. (`generators/src/lib.rs` lines 6, 29, 35)

3 General Findings

Overall, the code is consistent, neat, clean, and efficient.

3.1 Negative Findings

- *Cofactor clearing*: We list this finding as a negative one, only by technicality, as this finding increases the risk of implementation errors in future development, although this finding is handled correctly in [**SeraiRepo**].

  By multiplying all external inputs (`EdwardsPoint` commitments) by `INV_EIGHT` then re-multiplying by 8, the code correctly ensures those points lie in the primary subgroup. This is critical to avoid torsion-based forgeries, because every valid `EdwardsPoint` key comes equipped with 7 nearly-indistinguishable copies which only vary by a torsion element.

  A common pain point with Edwards25519 elliptic curve group, clearing cofactors lead to a significantly greater risk of implementation errors. Many developers do not understand the point of multiplying by `INV_EIGHT` only to multiply by 8, and may skip this step.

  We emphasize that this step appears to be performed correctly throughout [**SeraiRepo**]. While there are methods to avoid torsion elements (see `ristretto.group`, for example), these carry their own implementation problems.

- *Transcript construction and Malleability:* When signing *arbitrary* messages, the `monero-serai` crate may have some linkability or malleability vulnerabilities, which we now describe; however, when signing Monero messages in particular, or messages containing sufficient transcript data, we see no linkability or malleability issues. Moreover, the approaches used in `monero-serai` are up to industry standards when compared to IETF RFC 9591.

  We list this finding as a negative finding, similarly to the clearing of cofactors, based on technicality alone.

  The Fiat-Shamir transform requires "hashing the complete transcript" every time a challenge is computed to prevent malleability, and hashing and randomness is often dependent on the state of a transcript in `monero-serai`. In FROSTLASS, we aggressively apply this heuristic, to the detriment of our notation, in order to construct security proofs. Also, the transcript in `monero-serai` is aggressively updated with new data as it comes in, to prevent these problems.

  For example, signing depends on FROST algorithms, which are out of scope of this review. However, these algorithms are written to be consistent with IETF RFC 9591, which uses different transcript formats than `monero-serai`. Thus, the signers' main transcript must be forked and manipulated into a IETF-friendly form before calling FROST algorithms. In this example, the Fiat-Shamir transcript heuristic suggests that the entire FROST transcript be appended to the `monero-serai` transcript. A slight relaxation of this heuristic would allow only the FROST output to be appended to the `monero-serai` transcript. However, none of the output of these data are directly appended to the transcript.

  Indeed, `monero-serai` transcripts are only *mostly* created in a consistent way with the Fiat-Shamir heuristic, with certain modifications. The `monero-serai` transcripts are also not easily human-readable. Verifying that `monero-serai` is computing all hashes in FROSTLASS correctly is a nontrivial task.

Instead of following this heuristic to the letter, `monero-serai` appends derived data to the transcript which is sensitively dependent on the output of the FROST algorithm. Moreover, constructing valid signatures requires semi-honest participation in a way which would either reveal malleability to any honest participants, or would fail to yield a valid signature. Additionally, in the context of the Monero cryptocurrency, Monero messages include much (if not the remainder) of the missing transcript data. Thus, for `monero-serai`, the inclusion of the FROST transcript itself appears to not be necessary.

For example, some data can be deterministically hashed from transcript data to be independent from other deterministically hashed data from a later point in the transcript, preventing malleability without including all data in the transcript. If some $x = H(\texttt{transcript})$ is computed at some point, and then some $y = H(0 \mid\mid x)$ and $z = H(1 \mid\mid x)$ is computed later, y and z are independent and therefore need not be included in each others' hashes. On the other hand, if y is embedded in the hash for z, say $z = H(1 \mid\mid x \mid\mid z)$, proofs can exploit second pre-image resistance in H to argue that the order of computations must have gone $x \rightarrow y \rightarrow z$. This allows constructing formal proofs of security easier, in general.

Without the direct inclusion of the FROST transcript, though, future FROSTLASS implementations built to match `monero-serai` may accidentally introduce unforeseen malleability attacks not present in `monero-serai`. Moreover, without these transcript inclusions in FROSTLASS, writing unforgeability proofs is more difficult. Similarly, without these inclusions in `monero-serai`, verifying the immunity of the implementation to malleability is not directly possible.

For improved clarity and auditability, we recommend that future revisions of the code include detailed documentation specifying which fields are deliberately omitted from the transcript and why such omissions do not compromise security. This documentation should clearly reference the relevant code sections and explain the rationale (e.g., compatibility with Monero's messaging formats) so that future auditors can verify that no malleability or linkability vulnerabilities exist.

- Some data is randomly hashed from `keccak256`, or sampled from a pseudorandom number generator, when it could be more efficiently extracted from an extendable output function (XOF) like `shake256`. For example, aggregation coefficients μ and ρ. Using an XOF should bring marginal efficiency improvements.

3.2 Neutral Findings

- `UnreducedScalar` operations do not guarantee constant-time execution. This is both positive and negative. Variable-time operations are only used in signature verification, and constant-time operations are only used in signing. Indeed, constant-time operations do not leak private information at the cost of reduced efficiency.

- Commitment masking and signature unforgeability relies on the discrete logarithm assumption in the Ed25519 group.
  - The discrete logarithm assumption is not thought to be hard in the face of post-quantum attackers, placing a "shelf life" on all schemes which reduce to this assumption.

  - It is not clear whether the tightness gaps endemic in forking-based proofs of unforgeability reduce effective security for ring signatures below tolerable levels.
  - While industry standards are shifting, it remains standard to use discrete logarithm based protocols at least for the next few years. It will be important in the long run to investigate migrations away from this assumption.

- **Node Trust and Validation**: Since `monero-rpc` can operate on untrusted nodes, it attempts some validation (for example, verifying that a transaction's hash matches the requested hash). However, in general, the node is unavoidably relied upon for essential data about chain state. Thus, users should abandon interacting with a node whenever a mismatch from protocol expectations occur (see `InvadidNode`). This is not avoidable with current implementations of the Monero protocol.

- **Confidentiality of Queries**: Repeatedly querying the node about certain output indexes or unconfirmed transactions can leak usage patterns. Future or external solutions may integrate local caches or batch requests to reduce fingerprinting. Again, this is a known problem associated with trusting others' nodes and not running your own full node.

- **Pruned Transactions**: If the node can only supply pruned data, the library raises `RpcError::PrunedTransaction` if unpruned data is strictly required. Users must ensure they only proceed when they can meaningfully handle a pruned transaction (e.g. scanning coinbase outputs).

- **Batch verification correctness**: Accumulated scalars and points are carefully updated to ensure the final check equals the identity only if all aggregated proofs are valid. If any proof is malformed, the final multi-exponentiation will yield a non-identity point.

- **External Randomness.** The crate depends on externally provided randomness (via `rand_core` or through `OsRng` in tests). The Fiat–Shamir heuristic is used for non-interactive challenge derivation.

- **Custom serialization formats.** The library follows Monero on-chain protocols and uses custom formats, distinct from Monero protocol standards, for Serai-specific functionality. These are usually well-documented but special care should be taken to justify custom data formats and fully document each instance of a novel data structure.

3.3 Positive Findings

The following practices are used throughout.

- Use of `LazyLock` for thread-safe lazy initialization.

- Constant-time group operations from `curve25519-dalek`.

- Constant-time hash-to-point mapping.

- Implement `Zeroize` and `ZeroizeOnDrop` for sensitive data.

- Perform careful bounds checking when handling anonymity sets/tuples.
- Matches Monero behavior explicitly, even in the case of “wrong” behavior which is otherwise consistent with legacy Monero implementations.
- Use utility types for vectors of `Scalar` and `EdwardsPoint` for safe indexing and algebraic operations.
- Seal code with `sealed` modules, enforcing that certain traits are private to certain modules, so that only types defined within the same module can implement it.
- Use `Arc<Mutex>` for safe concurrent access during authentication of state and for nonce management.
- Manage thread-safe connections through `Arc<Mutex<(Option<(WwwAuthenticateHeader, u64)>, Client)>>` in the `monero-rpc` crate.
- Non-constant time primitives, specifically for use in signature verification, which explicitly match Monero’s default wallet implementation.
- Enforce canonical forms by requiring fully reduced values modulo the group order, canonical forms for point coordinates, and validation of prime-order subgroup membership.
- Thoroughly tested, including tests against Monero default wallet implementation, even recreating known “bad” or suboptimal behaviors in the Monero default wallet implementation.
- A zero hash result in `keccak256_to_scalar` causes a panic. This prevents a variety of malleability issues.
- **Batch verification correctness**: The `monero-bulletproofs` crate accumulates points and scalars carefully to ensure the final check equals the identity only if all aggregated proofs are valid. If any proof is malformed, the final multi-exponentiation will yield a non-identity point.
- **Randomness:** The `monero-bulletproofs` crate depends on externally provided randomness. In testing, this randomness is provided via `rand_core` or through `OsRng`. This is a positive finding because it is industry standard to avoid implementing random ness.
- **Transcripting:** A version of the Fiat-Shamir heuristic is employed for non-interactive challenge derivation, with regular updates to the transcript. The transcript is constructed in an exceedingly thorough way, but not precisely following the Fiat-Shamir heuristic, and not in a human-readable way; see Negative Findings, above.

4 Repository Structure

The repository at `github.com/serai-dex/serai` is organized into several top-level directories, including `/serai-dex/serai/networks/`, which in turn contains `/serai-dex/serai/networks/monero/`. Figure 1 describes the target directory structure, where asterisks indicate crates (named in parentheses); the only folder not in scope of this audit is `verify-chain`.

```
/serai-dex/serai/
|-- out_of_scope_files
|-- networks/
    |-- out_of_scope_files
    |-- monero*/ (monero-serai)
        |-- generators*/ (monero-generators)
        |-- io*/ (monero-io)
        |-- primitives*/ (monero-primitives)
        |-- ringct/
            |-- borromean*/ (monero-borromean)
            |-- bulletproofs*/ (monero-bulletproofs)
            |-- clsag*/ (monero-clsag)
            |-- mlsag*/ (monero-mlsag)
        |-- rpc*/ (monero-rpc)
            |-- simple-request*/ (monero-simple-request-rpc)
        |-- src/
        |-- tests/
        |-- verify-chain*/
        |-- wallet*/ (monero-wallet)
            |-- address*/ (monero-address)
```

Figure 1: Directory structure of the target repository. Everything within `/serai-dex/serai/networks/monero` is in scope, except the subfolder `/serai-dex/serai/networks/monero/verify-chain/`. Asterisks indicate crates. Crate names are included in parentheses.

In the following, we omit `/serai-dex/serai/networks/monero/` from directory references for clarity, with the understanding that all directory references use this as a prefix.

The crates are evocatively named by the data they handle, and mutually depend on each other. Crate dependency is not isomorphic to the file structure. Figure 2 displays these dependencies in the transitive reduction of the directed graph of crate dependencies. Readers should be aware that the transitive reduction of a directed graph G is constructed by removing as many edges as possible without changing the *reachability relation* on pairs of vertices. Thus, not all edges corresponding to direct crate dependency are displayed in Figure 2. For example, `monero-serai` depends directly on `monero-generators`, `monero-io`, and `monero-primitives`. In Section 5, we present a look at each crate.

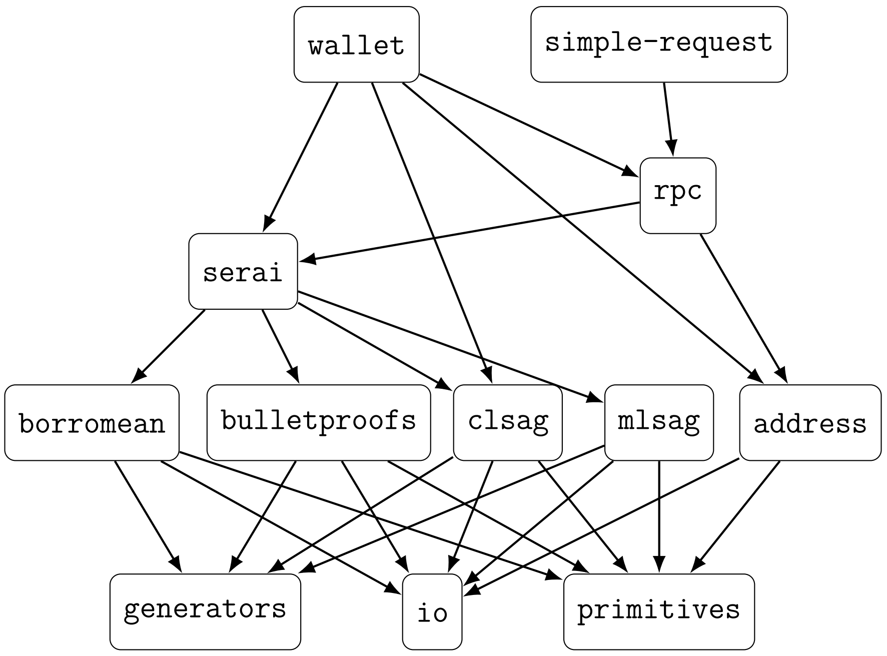

Figure 2: The transitive reduction of the graph of crates and their dependencies. Not all edges corresponding to a direct dependency are displayed in a transitive reduction. For example, `monero-serai` depends directly on `monero-generators`, `monero-io`, and `monero-primitives`, but these edges are not explicitly displayed.

5 Crates Overview

In this section, we describe each crate name, version, purpose, internal dependencies, and a brief description of crate structure. We use a breadth-first, top-down approach following Figure 2. Elements of public APIs (i.e. with the modifier `pub`) have names which are decorated `thusly`†, and elements exposed at the crate level (i.e. with the modifier `pub(crate)`) have names which are decorated `thusly`$^\Delta$. We provide links throughout the document to corresponding glossary entries.

- `monero-wallet (v0.1.0)`
  - Purpose: Handle all wallet functionality.
  - Internal Dependencies:
    - `monero-address (v0.1.0)`†
    - `monero-clsag (v0.1.0)`†
    - `monero-rpc (v0.1.0)`†,
    - `monero-serai (v0.1.4-alpha)`†
  - Structure: A standard library crate, with the corresponding entry point at `/wallet/src/lib.rs`.
  - Tests at `/wallet/src/tests/runner/mod.rs`.
  - The `monero-wallet (v0.1.0)` crate employs the following modules.
    - `/wallet/src/decoys.rs` handles decoy selection with a publicly exposed struct `OutputWithDecoys`.

    * `/wallet/src/extra.rs`† handles the `extra` field of a transaction.
    * `/wallet/src/output.rs`$^\Delta$ handles transaction outputs.
    * `/wallet/src/scan.rs` handles transaction scanning.
    * `/wallet/src/send/mod.rs`† handles sending transactions. This is a directory module, and contains the following file modules:
      · `/wallet/src/send/eventuality.rs` handles Eventualities.
      · `/wallet/src/send/multisig.rs` handles sending threshold transactions.
      · `/wallet/src/send/tx.rs` handles sending transactions.
      · `/wallet/src/send/tx_key.rs` handles keys for sending transactions.
    * `/wallet/src/view_pair.rs` handles pairs of keys, where one is a public spend key, and the other is a private view key.

- `monero-simple-request-rpc (v0.1.0)`
  - Purpose: Default RPC to avoid external dependencies on, e.g. reqwest. Only used in dev dependencies.
  - Internal Dependencies:
    * `monero-rpc (v0.1.0)`
  - Structure: A standard library crate, with the corresponding entry point at `/rpc/simple-request/src/lib.rs`.

- `monero-rpc (v0.1.0)`
  - Purpose: handle RPC calls for interacting on the Monero network.
  - Internal Dependencies:
    * `monero-address (v0.1.0)`
    * `monero-serai (v0.1.4-alpha)`
  - Structure: A standard library crate, with the corresponding entry point at `/rpc/src/lib.rs`, employing no file or directory modules.

- `monero-serai (v0.1.4-alpha)`
  - Purpose: the overall transaction library.
  - Internal dependencies:
    * `monero-borromean (v0.1.0)`
    * `monero-bulletproofs (v0.1.0)`
    * `monero-clsag (v0.1.0)`
    * `monero-generators (v0.4.0)`†
    * `monero-io (v0.1.0)`†
    * `monero-mlsag (v0.1.0)`
    * `monero-primitives (v0.1.0)`†
  - Structure: A standard library crate, with the corresponding entry point at `/src/lib.rs`.

    - File modules:
        - `/src/block.rs`†
        - `/src/merkle.rs`
        - `/src/ring_signatures.rs`†
        - `/src/ringct.rs`†
        - `/src/transaction.rs`†
    - Tests at `/src/tests/mod.rs`

- `monero-address (v0.1.0)`
  - Purpose: handles Monero addresses.
  - Internal dependencies:
    - `monero-io (v0.1.0)`
    - `monero-primitives (v0.1.0)`
  - Structure: A standard library crate, with the corresponding entry point at `/wallet/address/src/lib.rs`.
    - File module: `/wallet/address/src/base58check.rs`.
    - Tests at `/wallet/address/src/tests.rs`.

- `monero-borromean (v0.1.0)`
  - Purpose: Handles Borromean signatures and Borromean range proofs.
  - Internal dependencies:
    - `monero-generators (v0.4.0)`
    - `monero-io (v0.1.0)`
    - `monero-primitives (v0.1.0)`
  - Structure: A standard library crate, with the corresponding entry point at `/ringct/borromean/src/lib.rs`. Employs no modules, and untested.

- `monero-bulletproofs (v0.1.0)`
  - Purpose: Handles original bulletproofs and bulletproofs plus.
  - Internal dependencies:
    - `monero-generators (v0.4.0)`
    - `monero-io (v0.1.0)`
    - `monero-primitives (v0.1.0)`
  - Structure: A standard library crate, with the corresponding entry point at `/ringct/bulletproofs/src/lib.rs`.
    - File modules:
        - `/ringct/bulletproofs/src/batch_verifier.rs`$^\triangle$
        - `/ringct/bulletproofs/src/core.rs`$^\triangle$
        - `/ringct/bulletproofs/src/point_vector.rs`$^\triangle$
        - `/ringct/bulletproofs/src/scalar_vector.rs`$^\triangle$
    - Directory modules:

            - `/ringct/bulletproofs/src/original/mod.rs`$^\Delta$
            - `/ringct/bulletproofs/src/plus/mod.rs`$^\Delta$
        - Tests at `/ringct/bulletproofs/src/plus/mod.rs`.

- `monero-clsag (v0.1.0)`
    - Purpose: Handles CLSAG ring signatures and a FROST-like thresholdization.
    - Internal dependencies:
        - `monero-generators (v0.4.0)`
        - `monero-io (v0.1.0)`
        - `monero-primitives (v0.1.0)`
    - Structure: A standard library crate, with the corresponding entry point at `/ringct/clsag/src/lib.rs`.
        - File module: `/ringct/clsag/src/multisig.rs`.
        - Tests: `/ringct/clsag/src/tests.rs`

- `monero-mlsag (v0.1.0)`
    - Purpose: Handles MLSAG ring signatures.
    - Internal dependencies:
        - `monero-generators (v0.4.0)`
        - `monero-io (v0.1.0)`
        - `monero-primitives (v0.1.0)`
    - Structure: A standard library crate with entry point at `/ringct/mlsag/src/lib.rs`. Employs no modules, and untested.

- `monero-primitives (v0.1.0)`
    - Purpose: Handles Pedersen Commitments and Decoys.
    - Internal dependencies:
        - `monero-io (v0.1.0)`
        - `monero-generators (v0.4.0)`
    - Structure: A standard library crate with entry point at `/primitives/src/lib.rs`.
        - File module at `/primitives/src/unreduced_scalar.rs`.
        - Tests at `/primitives/src/tests.rs`.

- `monero-generators (v0.4.0)`
    - Purpose: Handles hashing data to elliptic curve group elements and all fixed generators used in Monero protocol computations.
    - Internally dependent only on `monero-io (v0.1.0)`.
    - Structure: A standard library crate with entry point at `/generators/src/lib.rs`.
        - File module at `/generators/src/hash_to_point.rs`.

    * Tests at `/generators/src/tests/mod.rs`

- `monero-io (v0.1.0)`
  - Purpose: Handles reading and writing various data structures used in Monero protocol computations (e.g. bytes, scalars, group elements, lists whose entries are the same type).
  - No internal dependencies.
  - Structure: A standard library crate with entry point at `/io/src/lib.rs`. Employs neither modules nor tests.

6 `monero-io` (v0.1.0)

The `monero-io` crate (`networks/monero/io/src/lib.rs`) implements canonical serialization and deserialization routines for various Monero protocol data types. It enforces strict canonicalization where required by validating varint encoding, scalar reduction, point encoding, and (optionally) sub-group membership.

`varint` Encoding (`read_varint`) Implemented at lines 133–151 . Uses a continuation-bit scheme to store a number in 7-bit chunks. The code rejects non-canonical encodings (leading zeros or overflow) and returns an error on malformed inputs.

Scalar Serialization (`read_scalar`) Implemented at lines 158–160. Reads a fixed 32-byte little-endian field element and ensures it is a canonical `curve25519-dalek` scalar, returning an `io::Error` otherwise.

Point Serialization (`read_point` / `read_torsion_free_point`) Implemented at lines 179–185 and 187–193. `read_point` calls `decompress_point` (lines 172–176) to guarantee a unique, canonical encoding, while `read_torsion_free_point` additionally verifies that the point lies in the prime-order subgroup.

Vector Deserialization (`read_array`) At lines 208–214 the vector returned by `read_raw_vec` is guaranteed to have length `N`; if not, `read_raw_vec` already returns an error and the closure is never invoked. The subsequent `vec.try_into().unwrap()` is therefore unreachable in practice and does not introduce a panic path.

Length-Prefixed Vector Deserialization (`read_vec`) At lines 216–219 a length prefix is read via `read_varint`, after which an *empty* vector is created and elements are pushed one-by-one as long as the stream supplies valid items. Although the crate does not pre-allocate the reported size, a maliciously long (or infinite) stream can still lead to memory exhaustion. Implementing an optional caller-supplied maximum length would mitigate this.

Findings (monero-io v0.1.0):

- **Unbounded Length Acceptance**: `read_vec` (lines 216–219) honours any length reported by `read_varint`. Exposing a configurable maximum element count is recommended.

- **Error Propagation & Documented Panics**: Functions such as `varint_len` and `write_varint` will panic if the supplied value exceeds `u64::MAX`; this is documented behaviour.
- **Cofactor vs. Torsion Checks**: Callers that require prime-order points must use `read_torsion_free_point`.

7 `monero-generators` (v0.4.0)

The `hash_to_point` function implements Monero's `hash_to_ec` operation to deterministically maps 32 bytes to a point on the Ed25519 curve as in:

Input

- Takes exactly 32 bytes.

Processing

1. Performs `Keccak256` hash of input bytes.
2. Squares the hash result and doubles it to get value `v`.
3. Computes intermediate values:
   - $\texttt{w} = \texttt{v + 1}$.
   - $\texttt{x} = \texttt{w}^2 - \texttt{A}^2 v$ where $A = 486662$ (curve parameter).
   - Calculates a partial X-coordinate through a series of field operations.
4. Derives Y-coordinate through:
   - Calculates `y = w - x`.
   - Sets sign bit based on zero checks.
   - Performs field inversions and multiplications to get final Y value.
5. Sets the high bit of byte 31 based on calculated sign.

Output

- Returns an EdwardsPoint in compressed 32-byte format:
  - Bytes 0 through 30: Y coordinate.
  - Byte 31: Sign bit of X coordinate (MSB format).
- Multiplies result by cofactor (8) to ensure point is in the prime-order subgroup.

The function uses the `FieldElement` type from `dalek-ff-group` for constant-time field arithmetic operations over the Ed25519 field of order $2^{255} - 19$. All operations are performed in constant time as a mitigation against timing-based side-channel attacks.

7.1 Operations

The `monero-generators` crate provides generator points for Pedersen commitments and Bulletproofs in Monero as in:

7.2 Core Generator

Generation

The base generator point H is computed as in:

1. Take `ED25519_BASEPOINT_POINT` (the standard Ed25519 base point G).
2. Compress it to 32 bytes.
3. Compute `Keccak256` hash of these bytes.
4. Map the hash to a curve point using `hash_to_point`.
5. Multiply by the cofactor, 8.

Powers Table

Precomputes powers of H for efficient amount commitments as in:

- Creates array of 64 points. The Rust implementation uses additive notation as do most CryptoNote-style protocols, so this is the vector[1] $[H, 2H, 4H, \ldots, 2^{63}H]$.
- Each entry is double the previous entry.
- Used for efficient decomposition of u64 amounts.

Potential Vulnerabilities and Recommendations:

1. **Use of `unwrap`**: Critical operations (e.g., field inversion, decompression) still employ `unwrap()` without explicit error handling.

   Recommendation: Replace `unwrap()` with `expect()` (with descriptive messages) or return a `Result` to mitigate unexpected panics.

2. **Variable Reuse and Code Clarity**: Variables like `x` and `X`, `y` and `Y` may cause confusion in the function's local logic.

   Recommendation: Consider inline comments clarifying each variable's purpose to minimize confusion.

7.3 Bulletproofs Generator Vector Creation

Constants

- `MAX_COMMITMENTS`: 16 (maximum provable commitments per range proof).
- `COMMITMENT_BITS`: 64 (bits per commitment value).
- `MAX_MN`: `MAX_COMMITMENTS` * `COMMITMENT_BITS` (total bits in proof).

Generator Creation

Takes a domain separation tag (`dst`) and produces two vectors of points:

1. Create `preimage` $= H$`.compress()`$||$`dst`.
2. For $i = 0, \ldots,$ `MAX_MN` $- 1$:
   - Let $j = 2 * i$.

[1]In the multiplicative notation more traditionally used in group theory, this is the vector $[H, H^2, H^4, ..., H^{2^{63}}]$.

- $[i] =$ `hash_to_point`(`Keccak256`(`preimage`||`varint`(j))).
- $G[i] =$ `hash_to_point`(`Keccak256`(`preimage`||`varint`($j+1$))).

Output

Returns `Generators` struct containing:

- G: Vector of `MAX_MN` points for value commitments.
- H: Vector of `MAX_MN` points for mask commitments.

All operations maintain constant-time properties through:

- Use of `LazyLock` for thread-safe lazy initialization.
- Constant-time point operations from `curve25519-dalek`.
- Constant-time hash-to-point mapping.

8 `monero-primitives` (v0.1.0)

8.1 Overview

The `monero-primitives` crate provides the core cryptographic operations required by Monero's protocol. The crate operates in both `std` and `no-std` environments, with certain optimizations available when `std` is enabled.

8.2 Core Types

8.2.1 UnreducedScalar

A structure representing an unreduced scalar value, primarily used to handle legacy Monero code sections where scalar reduction was not enforced:

```
pub struct UnreducedScalar(pub [u8; 32]);
```

Key operations:

- `recover_monero_slide_scalar`: Recovers the scalar that would have been incorrectly interpreted by Monero's `slide` function due to lack of reduction checks in Borromean range proofs.
- `non_adjacent_form`: Computes the width-5 non-adjacent form (NAF) representation, intentionally matching Monero's potentially incorrect behavior.

8.2.2 Commitment

Represents a transparent Pedersen commitment with the following structure:

```
pub struct Commitment {
    pub mask: Scalar,
    pub amount: u64
}
```

Operations:

- `calculate`: Computes the Pedersen commitment point as $amount \cdot G + mask \cdot H$.
- `zero`: Creates a commitment to zero using a mask value of 1.
- Serialization and deserialization functionality.

8.2.3 Decoys

Manages ring signature data with the structure:

```
pub struct Decoys {
    offsets: Vec<u64>,
    signer_index: u8,
    ring: Vec<[EdwardsPoint; 2]>
}
```

Features:

- Stores ring member positions as offsets from previous positions.
- Maintains key-commitment pairs for each ring member.
- Provides access to the signer's position and ring members.

8.3 Core Operations

8.3.1 Hash Operations

- `keccak256`: Computes `Keccak256` hash of input data.
- `keccak256_to_scalar`: Maps input to a scalar via `Keccak256`(data) mod ℓ, where ℓ is Ed25519's prime order.

8.3.2 Cached Computations

When the `std` feature is enabled:

- `INV_EIGHT`: Caches 8^{-1} mod ℓ.
- `G_PRECOMP`: Maintains precomputed tables for the Ed25519 base point.

8.4 Implementation Details

8.4.1 NAF Implementation

- Non-constant time operation, explicitly matching Monero's implementation.
- Window size is fixed at 5 bits.
- Edge cases:
  - Handles overflow beyond 256 bits by truncating.

  - Uses special carry propagation when the sum exceeds 15 or the difference goes below -15.

The NAF computation in `non_adjacent_form` follows these steps:

1. Convert input to a bit array.
2. Process bits sequentially.
3. For each non-zero bit:
   - Examine the next 5 bits (window).
   - Combine bits when sum ≤ 15 or difference ≥ -15.
   - Handle carry propagation.

This intentionally matches Monero's implementation, including specific edge cases which may or may not be "correct" in contexts other than matching the Monero default wallet implementation.

8.4.2 Scalar Recovery

The `recover_monero_slide_scalar` function:

- Returns direct reduction if the high bit is not set.
- Otherwise reconstructs the scalar from its NAF representation.
- Uses precomputed odd scalars for efficiency.

8.5 Security Considerations

8.5.1 Critical Assumptions

- A zero hash result in `keccak256_to_scalar` causes a panic.
- `UnreducedScalar` operations do not guarantee constant-time execution.
- Commitment masking relies on the discrete logarithm assumption in the Ed25519 group.

8.5.2 Implementation Notes

- **Serialization**:
  - Uses custom formats, distinct from Monero protocol standards.
  - Provides no backwards compatibility guarantees.
- **Memory Safety**:
  - Implements `Zeroize` and `ZeroizeOnDrop` for sensitive data.
  - Performs careful bounds checking when handling rings (anonymity sets/tuples).
- **Compatibility Choices**:

    - Matches Monero behavior even when suboptimal.
    - Preserves specific edge cases for protocol compatibility.
    - Retains variable-time operations where Monero uses them.

- Implements `Zeroize` and `ZeroizeOnDrop` for sensitive data.
- Matches Monero's behavior for compatibility, even if suboptimal.
- Employs crate-specific serialization formats, not the Monero protocol standard.

8.6 Critical Findings

1. **Non-Adjacent Form (NAF) Vulnerability:**
   The `non_adjacent_form` function executes in variable time and is vulnerable to timing side-channel attacks. This is documented.

   Recommendation: Consider offering a constant-time alternative if used in secret-dependent contexts.

8.7 Moderate Findings

1. **Inconsistent Error Handling:**
   The code uses a mix of `assert!`, `Option<T>`, and `io::Result<T>`, which can lead to unclear error propagation and unexpected termination.

   Recommendation: Standardize on a uniform, `Result`-based error handling approach to provide consistent and contextual error reporting.

8.8 Neutral Findings

1. **Reachable Panic in Scalar Conversion:**
   The `keccak256_to_scalar` function uses an `assert!` that can panic if the resulting scalar equals zero. This is a documented design decision because the probability of this occurring is $\approx \frac{1}{2^{-252}} =\approx 2^{252}$.

    2. **Recommendation:** Replace the assertion with proper error propagation to safely handle unexpected inputs.

9 `monero-rpc` (v0.1.0)

The `monero-rpc` crate provides abstractions for communicating with a Monero daemon, retrieving various data (blocks, transactions, fee estimates, *etc.*), and publishing transactions. The crate relies on data structures from `monero-serai`, using them to parse and serialize Monero primitives (such as transactions and blocks). It offers a specialized trait for decoy selection.

9.1 Overview

The `monero-rpc` crate defines:

1. **`RpcError`**: An enumeration capturing errors that may arise when performing RPC calls (see §9.2).
2. **`Rpc` trait**: The primary abstraction for interacting with a Monero daemon. It defines a set of asynchronous methods for retrieving blocks, transactions, and other chain data, as well as publishing transactions (see §9.3).
3. **`DecoyRpc` trait**: A higher-level trait extending the concept of retrieving outputs for constructing ring signatures (see §9.10).
4. **Various helpers and supporting items**, such as:
    - `ScannableBlock`, bundling a *Monero* block together with its non-miner transactions (in pruned form), plus metadata used when scanning outputs.
    - `FeeRate` and `FeePriority`, providing abstractions for Monero fee estimation.
    - Utility functions to parse certain specialized binary responses from a node, such as `get_o_indexes.bin`.

Implementors of `Rpc` are expected to supply the low-level HTTP or transport logic, including authentication if applicable. The actual requests and responses are shaped to match Monero's JSON-RPC and specialized "binary" endpoints. This allows various backends (e.g., over Tor/i2p, local node, dedicated HTTP crates) to be slotted in.

9.2 `RpcError` Enumeration

The `RpcError` enum defines all error conditions encountered by the RPC layer:

- **`InternalError(String)`** — Typically signals some logical problem unrelated to standard node replies (for instance, constructing a request with out-of-range parameters).
- **`ConnectionError(String)`** — Indicates an inability to reach or properly communicate with the daemon (e.g. timeouts, malformed responses, or connectivity disruptions).
- **`InvalidNode(String)`** — The remote node returned data not conforming to the Monero protocol, or otherwise gave impossible or contradictory information (suspecting a malicious or misconfigured node).
- **`TransactionsNotFound(Vec<[u8; 32]>)`** — One or more requested transactions could not be retrieved by the node.
- **`PrunedTransaction`** — A transaction was retrieved in a pruned form when unpruned data was required. In the current code, pruned transactions are not considered usable for certain operations.
- **`InvalidTransaction([u8; 32])`** — The node claims a transaction is valid, but it fails local parsing or verification.

- **InvalidFee** — The fee returned by the node was nonsensical or out-of-range for safe usage.
- **InvalidPriority** — The given fee priority could not be honored or mapped (e.g. out of the valid `1-4` range for Monero's known fee multipliers).

Such typed error handling allows the upper layers or other consumers of the `monero-rpc` crate to correctly distinguish user-facing issues (like `TransactionsNotFound`) from transport or malicious-node issues (`ConnectionError` or `InvalidNode`).

9.3 `Rpc` Trait

The `Rpc` trait is the centerpiece of the crate, defining the primary asynchronous calls for interacting with a Monero daemon. Its methods can be broken into three categories: *primitive calls*, *block/transaction retrieval*, and *transaction publishing*.

9.4 `post`

```
fn post(
    &self,
    route: &str,
    body: Vec<u8>
) -> impl Future<Output = Result<Vec<u8>, RpcError>> + Send;
```

At the lowest level, `post` takes an HTTP route string (e.g.: `get_transactions`) and a raw byte vector containing the request body. It returns a future resolving to a `Result` with the raw byte vector for the response (or an error). Implementors must handle:

- HTTP or HTTPS connections.
- Digest or basic authentication.
- Connection pooling or caching.

Nothing in `monero-rpc` enforces a particular method of connection; `post` abstracts this detail away.

9.5 `rpc_call` and `json_rpc_call`

```
fn rpc_call<Params: Serialize + Debug,
            Response: DeserializeOwned + Debug>(
    &self,
    route: &str,
    params: Option<Params>
) -> impl Future<Output = Result<Response, RpcError>> + Send;

fn json_rpc_call<Response: DeserializeOwned + Debug>(
    &self,
    method: &str,
    params: Option<Value>
) -> impl Future<Output = Result<Response, RpcError>> + Send;
```

These two methods use `post` internally, providing:

1. `rpc_call`: For "raw" RPC calls, typically when a daemon endpoint does not follow the JSON-RPC 2.0 standard but instead uses POST-based JSON data (for example, `get_transactions`).

2. `json_rpc_call`: For endpoints explicitly requiring `json_rpc` protocol usage (e.g. `get_block_header_by_height`). The parameters and method name are placed into the JSON-RPC structure, sent, and the result is deserialized back to a typed `Response`.

Both automatically handle JSON deserialization into strongly typed Rust structures. If the response is malformed or the node returns an error-like structure, `RpcError::InvalidNode` or `RpcError::ConnectionError` may be produced.

9.6 `get_height`, `get_transactions`, and Block-Related Methods

The crate includes convenience methods for:

- `get_height()`: Returns the current chain height (the number of blocks, with the genesis block counted as height 1).

- `get_block`, `get_block_by_number`, and `get_block_hash`: Retrieve and verify specific blocks, either by zero-indexed block number or by 32-byte hash. The code ensures the block's serialized hash matches the node's reported identifier, guarding against untrusted or corrupted data.

- `get_transactions` and `get_pruned_transactions`: Fetch zero or more transactions by their 32-byte hash, verifying they match local expectations (checking the computed transaction hash). Pruned or unpruned forms are selectively parsed via the relevant entry point.

- `get_transaction` (unpruned) and `get_pruned_transaction`: Single-transaction convenience wrappers.

Block retrieval can optionally return a `ScannableBlock`, packaging a `Block` and its pruned non-coinbase transactions, allowing scanning for RingCT outputs.

9.7 `publish_transaction`

```
fn publish_transaction(
    &self,
    tx: &Transaction
) -> impl Future<Output = Result<(), RpcError>> + Send;
```

Publishes a `Transaction` to the network. It leverages the `send_raw_transaction` route (or equivalent) on the Monero daemon. Should the daemon reject the transaction (for instance, if it conflicts with a recently seen double spend), an appropriate `RpcError` is returned.

9.8 get_fee_rate

```
fn get_fee_rate(
    &self,
    priority: FeePriority
) -> impl Future<Output = Result<FeeRate, RpcError>> + Send;
```

Requests fee estimates for a particular `FeePriority` (one of `Unimportant`, `Normal`, `Elevated`, `Priority`, or `Custom`). Internally, it interprets the JSON response to obtain the fee in "per-weight" units and a quantization mask. The resulting `FeeRate` structure:

- `per_weight`: The base fee per weight unit.
- `mask`: A mask used to round the final transaction fee upward.

The consumer can compute the final fee for a transaction based on the transaction weight and this `FeeRate`. If an unrecognized or invalid priority is requested, `RpcError::InvalidPriority` is raised.

9.9 generate_blocks

While not part of a typical production Monero daemon (where block generation is not done via RPC in mainnet environments), `generate_blocks` is useful in a local testing context (e.g. in a private regtest setup). It creates blocks, awarding the block reward to the specified `Address`.

```
fn generate_blocks<const ADDR_BYTES: u128>(
    &self,
    address: &Address<ADDR_BYTES>,
    block_count: usize
) -> impl Future<Output = Result<(Vec<[u8; 32]>, usize), RpcError>> + Send;
```

This returns a list of the newly mined blocks' hashes and the resulting height.

9.10 DecoyRpc Trait

The `DecoyRpc` trait extends the base RPC for selecting decoy outputs. When constructing Monero transactions, ring signatures typically reference random "decoy" outputs on-chain, enabling privacy. `DecoyRpc` adds specialized queries for zero-amount RingCT outputs:

- `get_output_distribution_end_height()`: Reports the maximum block index used in the distribution. Typically this equals the chain height.
- `get_output_distribution(range: Range<usize>)`: Returns a vector of cumulative output counts for that block range. By focusing on zero-amount outputs, it leverages the `get_output_distribution` endpoint.
- `get_outs(indexes: &[u64])`: Fetches details for a batch of RingCT outputs, including the block height, unlocking status, compressed key ($\mathtt{C}_i$), and the commitment (`mask`).

- `get_unlocked_outputs(indexes, height, fingerprintable)`: Returns the subset of outputs that are actually unlocked by the node's current chain state, or by a purely deterministic check (when `fingerprintable_deterministic` is true). This helps ensure decoys represent valid, spendable outputs from the node's perspective.

In typical usage, one might maintain a local database of zero-amount outputs (built incrementally), but `DecoyRpc` allows direct on-demand retrieval from a node if required.

9.11 Supporting Types and Structures

9.11.1 `ScannableBlock`

```
pub struct ScannableBlock {
    pub block: Block,
    pub transactions: Vec<Transaction<Pruned>>,
    pub output_index_for_first_ringct_output: Option<u64>,
}
```

`ScannableBlock` encapsulates a fully read, unpruned coinbase (*miner*) transaction plus the pruned forms of other transactions, along with an optional `output_index_for_first_ringct_output`. This index helps to avoid repeated queries for the output index of every single RingCT output in the block, which can be inefficient.

9.11.2 `FeeRate`

```
pub struct FeeRate {
    per_weight: u64,
    mask: u64,
}
```

This data structure is an interpreted result from the node's fee estimate. The `FeeRate::calculate_fee_from_weight` method performs final rounding:

$$\text{fee} = \left\lceil \frac{\texttt{per\_weight} \times \texttt{tx\_weight}}{\texttt{mask}} \right\rceil \times \texttt{mask}.$$

9.11.3 `FeePriority`

```
pub enum FeePriority {
    Unimportant,
    Normal,
    Elevated,
    Priority,
    Custom { priority: u32 },
}
```

Monero typically recognizes four distinct priority levels: "unimportant," "normal," "elevated," and "priority." Each internally maps to a numeric multiplier for the base fee. The `Custom` variant allows direct numeric control, though the node might reject unknown or extreme values.

9.12 Security Considerations

1. **Node Trust and Validation**: Since `monero-rpc` can operate on untrusted nodes, it attempts some validation (for example, verifying that a transaction's hash matches the requested hash). However, in general, the node is relied upon for essential data about chain state. The `InvalidNode` error signals a mismatch from protocol expectations.

2. **Confidentiality of Queries**: Repeatedly querying the node about certain output indexes or unconfirmed transactions can leak usage patterns. Future or external solutions may integrate local caches or batch requests to reduce fingerprinting.

3. **Pruned Transactions**: If the node can only supply pruned data for certain legacy or low-mixin transactions, the library raises `RpcError::PrunedTransaction` if unpruned data is strictly required. Users must ensure they only proceed when they can meaningfully handle a pruned transaction (e.g. scanning coinbase outputs).

9.13 Findings

1. **Unchecked Conversions and Potential Panics**: The codebase contains instances of unchecked type conversions and unwrapped operations (e.g., converting between `usize` and `u64` or `u32`) that can lead to runtime panics if given malicious or otherwise unexpected data.

   > **Recommendation:** Replace `unwrap()` calls with explicit error handling or use `expect()` with descriptive messages, ensuring domain assumptions are enforced.

2. **Limited Node Response Verification**: Verification of responses (e.g., in `get_transactions`) primarily relies on hash comparisons with minimal structural checks. While this helps catch some invalid data, additional validation of response fields would improve security, such as:

   > **Recommendation:** Implement further integrity checks and stricter validation of node-supplied data, such as:
   >
   > - Verifying that each returned transaction's *weight* matches the byte-length supplied by the daemon.
   > - Ensuring every input's key-image is 32 B and—in pruned mode—that the corresponding pseudo-output commitment is present.
   > - Rejecting blocks whose timestamp lies more than 2 ahead/behind the local clock.
   > - Confirming that `extra` fields obey the length limits imposed by consensus rules (e.g. bulletproof sizes, tx-public-key length).

3. **Timelock and Arithmetic Safety**: Unchecked arithmetic operations—especially those involving timelock values—could lead to overflows or logical errors. A `// TODO: ...` comment (e.g., referencing `github.com/serai-dex/serai/issues/104`) highlights the need for more robust handling of unusual timelock scenarios.

   > **Recommendation:** Use checked arithmetic operations and validate timelocks in boundary cases.

4. **Cryptographic Timing Considerations** (Informational)
   The RPC crate itself does not handle wallet private keys or other high-value secrets; equality checks currently compare only public data (hashes, commitments, points). While side-channel risk is therefore low in today's code, future extensions that compare secret-dependent values *in place* must adopt constant-time routines (e.g. `subtle::ConstantTimeEq`).

   Recommendation: Document this assumption clearly and integrate constant-time helpers if secret material is ever introduced.

10 `monero-bulletproofs` (v0.1.0)

The `monero-bulletproofs` crate implements both the original Bulletproofs range proof scheme and the newer Bulletproofs+ scheme for the Monero protocol. The library is located within the Serai repository under `networks/monero/ringct/bulletproofs`. Its purpose is to generate and verify range proofs over amounts in confidential transactions, ensuring that the amounts are in a valid range (specifically $[0, 2^{64})$) without revealing the amounts themselves. Below is an overview of its structure, methods, and verification flow, referencing the relevant source files.

10.1 Structure and Entry Points

The crate exposes its main functionality via:

- `Bulletproof`: An enum representing either the original Bulletproof or a Bulletproof+ proof.
- `prove` and `prove_plus`: Functions that construct the respective Bulletproof or Bulletproof+ proofs for a list of commitments.
- `verify` and `batch_verify`: Functions that verify these proofs, either in a standalone or batched manner.

Internally, the crate is separated into several modules:

- `core`: Provides low-level operations such as multi-exponentiation routines and challenge product computation.
- `batch_verifier`: Holds accumulators that enable batching multiple Bulletproof or Bulletproof+ verifications into a single multi-exponentiation.
- `original`: Implements the original Bulletproofs approach, including the inner-product proof (`IpProof`) used by `AggregateRangeProof`.
- `plus`: Implements Bulletproofs+ as described by the Monero project. It includes the improved weighted inner product proof (`WipProof`) and the aggregated range proof mechanism that uses it.
- `scalar_vector` and `point_vector`: Utility types for vectors of `Scalar` and `EdwardsPoint`, offering safe indexing and algebraic operations.
- `tests`: Internal tests that validate correctness across both schemes, employing batched and single verifications.

10.2 Range Proof Representation

The top-level enum `Bulletproof` is:

- `Original(OriginalProof)`: The classic Bulletproof structure using the original protocol design.
- `Plus(PlusProof)`: The Bulletproof+ structure, notably smaller and with a slightly different internal proving mechanism.

Each structure internally contains the group elements and scalars required for proof verification. For instance, `OriginalProof` holds:

- `(A, S, T1, T2)`: Points used in the range proof polynomial commitment.
- `tau_x, mu, t_hat`: Challenge-related scalars for the polynomial identity check.
- `ip`: An `IpProof` implementing the inner product argument.

Likewise, `PlusProof` holds:

- `A`: The aggregated vector commitment.
- `wip (WipProof)`: A specialized weighted inner product proof, the core cryptographic novelty of Bulletproofs+.

10.3 Aggregated Range Proof Generation

Both Bulletproof and Bulletproof+ provide *aggregated* proofs that combine multiple range proofs for multiple outputs into a single proof whose size grows only logarithmically in the number of outputs. Each scheme follows a "commit-and-challenge" approach similar to standard Bulletproof protocols, but they differ in their choice of transcript hashing and the particular steps for the inner product argument.

Original Bulletproof (`original` module) The original code is in `original/mod.rs` and `original/inner_product.rs`, closely mirroring the *Bulletproofs* paper. Key steps:

1. *Precomputation:* Generators `G` and `H` are allocated, one for each bit of each commitment. The maximum number of commitments is specified by `MAX_COMMITMENTS`.
2. *Polynomial Commitments:* The protocol encodes bits of the amounts in vectors (a_L, a_R), from which it forms commitments (A, S).
3. *Challenges:* The code uses `Keccak256` (converted to a `Scalar` via `keccak256_to_scalar`) to derive random challenges y, z, x.
4. *Constrained Polynomial Identity:* The aggregator constructs (T_1, T_2) to bind the polynomial terms, and merges everything with the challenge x to produce $\hat{t}$, τ_x, and μ.
5. *Inner Product Proof:* The `IpProof` is generated by recursively splitting vectors in half and committing to partial inner products, storing the commitments (L_i, R_i).

Bulletproof+ (`plus` module) Bulletproof+ is provided by the `plus/` folder, primarily in `weighted_inner_product.rs` and `aggregate_range_proof.rs`. The official Monero code slightly alters the structure of the proof to reduce proof size, while still maintaining logarithmic size. Notable elements:

1. The *Weighted Inner Product* (`WipStatement`) generalizes the classic inner product. It incorporates certain transformations by weighting vectors with powers of y and rearranging final group elements.
2. The *Generators* are re-labeled for Monero's usage: `g_bold` and `h_bold` in code, though the actual roles are reversed (Monero uses H as the value basepoint and G for mask derivation).
3. Aggregation is performed similarly with powers of z and y, but the final step uses a more efficient `WipProof`.
4. The function `prove_plus` sets up the aggregated statement, computes an A commitment, then calls `WipStatement::prove` to produce the final weighted inner product proof.

At verification time, `aggregate_range_proof.rs` re-derives the same hashed challenges from the set of commitments and checks consistency with the `WipProof`. As with the original proof, a batched approach merges all terms into a single multi-scalar multiplication check.

10.4 Core Routines and Utilities

`core/mod.rs` Contains low-level primitives:

- `multiexp` and `multiexp_vartime`: Perform (variable-time) multi-exponentiation to accumulate the proof checks.
- `challenge_products`: Generates the partial products of the challenges (e_i, e_i^{-1}) for the recursive proof steps, optimizing repeated multiplications.

These functions are the primary means of combining scalar-point pairs for cryptographic checks throughout the library.

`batch_verifier.rs` Implements the `BatchVerifier` structure, storing:

- `original`: An accumulator for the original Bulletproof scheme.
- `plus`: An accumulator for Bulletproof+.

Each accumulator collects scalar multipliers for fixed and variable basepoints (respectively `G`, `H`, per-commitment bases, etc.). Finally, `verify` executes a *single* call to `vartime_multiscalar_mul` to confirm all queued proofs in batch.

`scalar_vector.rs` and `point_vector.rs` Define lightweight wrappers around `Vec<Scalar>` and `Vec<EdwardsPoint>` to simplify:

- Indexing with bounds checks (in debug mode).
- Element-wise operations: addition, subtraction, or multiplication on each vector element.
- Inner product computation: a straightforward $(\mathbf{a}, \mathbf{b}) \mapsto \sum a_i b_i$.

These wrappers help maintain correctness by preventing mistakes in indexing or length mismatches between vectors.

10.5 Transcript and Fiat–Shamir Challenges

Throughout the protocol, challenges $(y, z, x, \dots)$ are computed by hashing various group elements and scalars with `Keccak256`. This is done via:

- `keccak256_to_scalar`(...): Converts a `Keccak256` hash into a `Scalar`.
- Combined with domain separators and partial transcripts (e.g. `A`, `S`, `T1`, `T2` in the original scheme).

In `plus/transcript.rs`, Bulletproof+ uses an additional domain separation constant `bulletproof_plus_transcript`, plus Monero's `hash_to_point` trick, though its necessity is not entirely clear from the code.

10.6 Security Observations

- *Cofactor clearing*: By multiplying all external inputs (`EdwardsPoint` commitments) by `INV_EIGHT` then re-multiplying by 8, the code ensures those points lie in the primary subgroup. This is critical to avoid torsion-based forgeries.
- *Batch verification correctness*: Accumulated scalars and points are carefully updated to ensure the final check equals the identity only if all aggregated proofs are valid. If any proof is malformed, the final multi-exponentiation will yield a non-identity point.
- *Randomness and transcript:* The crate depends on externally provided randomness (via `rand_core` or through `OsRng` in tests). The Fiat–Shamir heuristic is used for non-interactive challenge derivation.

10.7 Findings

1. **Transcript Construction and Domain Separation:**
   The code (e.g. `plus/transcript.rs` lines `8-17` and `transcript_A_B` in `weighted_inner_product.rs` lines `247-269`) employs a static domain-separation constant derived via `hash_to_point`. Although functionally correct, the inline documentation does not explicitly clarify how this constant binds challenges to specific protocol parameters or commit to all relevant data (e.g. original basepoints).

Recommendation: Expand inline comments or code documentation to explain each parameter included in the transcript, referencing Monero's recommended domain separation. Where possible, incorporate additional references (like basepoints or block-specific tags) to reduce the risk of cross-protocol collisions.

2. **Error Handling and Unchecked Conversions:**
Several functions still rely on `unwrap()`, but the uses in `lib.rs` lines 108–146 (`prove`) come *after* the witness has already been validated (lines 108–113). Because the two validation stages are wholly internal to the crate, these panics should be unreachable in practice. The current structure is therefore sound, though slightly duplicitous.

    Recommendation: Consolidate the two validation paths (witness checks and the subsequent assertions) or replace the remaining `unwrap()`s with explicit error-propagation helpers such as `expect("invariant violated")`. Either approach would preserve today's safety guarantee while making the intended invariant clearer to future maintainers.

3. **Cofactor Clearing and Documentation Consistency:**
The code correctly multiplies external inputs by `INV_EIGHT` and compensates by multiplying by `8` later (e.g. `original/mod.rs` lines `61-62` and `batch_verifier.rs` lines `23-53`). However, the comments explaining why this step eliminates torsion-based forgeries vary in detail across modules.

    Recommendation: Standardize these comments to note that cofactor clearing ensures points lie in the primary subgroup. Reference lines `61-62` in `original/mod.rs` from the vantage of best practices recommended in cryptographic literature (e.g. the rationale behind multiplying by the cofactor).

4. **Variable-Time Operations:**
The crate uses `multiexp_vartime` for public data, which is acceptable as long as it is never called with secret-dependent scalars. For instance, see `core.rs` lines `50-74` and references in `plus/weighted_inner_product.rs` lines `416-435`.

    Recommendation: Audit each call to `multiexp_vartime` to ensure it cannot be reached with secret values. If any scalar is secret, switch to a constant-time alternative or document precisely why variable time is safe (e.g. purely public aggregator data).

11 `monero-address` (v0.1.0)

11.1 Purpose

The `monero-address` crate provides functionality for handling Monero addresses, including standard, subaddress, integrated, and featured addresses. It also supports encoding and decoding Monero addresses in `base58Check` format.

11.2 Internal Dependencies

- `monero-io (v0.1.0)`.
- `monero-primitives (v0.1.0)`.

11.3 Structure

A standard library crate, with the corresponding entry point at `/wallet/address/src/lib.rs`.

- `base58Check` module.
- Tests at `/wallet/address/src/tests.rs`.

11.4 Functionality

The `monero-address` crate defines several types of Monero addresses and provides encoding, decoding, and validation functionality.

Address Types The `monero-address` crate defines the following address types.

- **Legacy:** A standard Monero address (public spend and view keys).
- **Legacy Integrated:** A legacy address with an embedded 8-byte payment ID.
- **Subaddress:** A derived address that allows the receiver to differentiate sources of funds.
- **Featured Address:** An extended address format supporting subaddresses, embedded payment IDs, and a guarantee against the burning bug.

Each address type is represented using the `AddressType` enum, which provides methods for checking whether an address is a subaddress, retrieving its embedded payment ID, and determining if it is guaranteed.

Network Identification The `Network` enum defines three possible Monero network types.

- **Mainnet:** The primary Monero blockchain.
- **Stagenet:** A staging network with the same rules as mainnet, used for testing.
- **Testnet:** A network used to test new features before they are deployed.

Each network is associated with specific address prefix bytes to distinguish them from each other.

`base58Check` Encoding and Decoding The `base58Check` module implements `base58Check` encoding and decoding, ensuring that Monero addresses include a checksum for error detection.

- `encode`: Encodes a byte array into `base58Check` format.
- `decode`: Decodes a `base58Check` string into raw bytes.
- `encode_check`: Computes a checksum and encodes a byte array into `base58Check`.
- `decode_check`: Decodes a `base58Check` string and verifies its checksum.

Address Parsing and Generation Monero addresses can be created from public spend and view keys, or parsed from strings.

- `MoneroAddress::new`: Creates a new Monero address given a network, address type, and key pair.
- `MoneroAddress::from_str`: Parses an address string for a given network.
- `MoneroAddress::from_str_with_unchecked_network`: Parses an address string without verifying the network.
- `MoneroAddress::to_string`: Serializes an address to a `base58Check`-encoded string.

Address Validation The crate ensures that parsed addresses meet the expected format.

- **Length validation:** Ensures addresses are the correct length based on their type.
- **Checksum verification:** Ensures that decoded addresses pass the `base58Check` validation.
- **Key validation:** Ensures that the spend and view keys are valid Edwards25519 points.

11.5 Serialization Details

The `monero-address` crate serializes addresses into a compact binary format before encoding them into `base58Check`. The serialization structure is as follows.

1. **Network Byte:** A single byte indicating the network and address type.
2. **Spend Key:** A 32-byte compressed Edwards25519 public key.
3. **View Key:** A 32-byte compressed Edwards25519 public key.
4. **Optional Fields:**
    - **Integrated Payment ID:** An 8-byte payment ID (only for integrated addresses).
    - **Feature Flags:** A variable-length integer encoding subaddress, payment ID, and guarantee status.
5. **Checksum:** A 4-byte `Keccak256` hash of the serialized data.

11.6 Testing

The `monero-address` crate includes comprehensive tests to validate the correctness of its encoding, decoding, and parsing functions.

- **Base58Check Encoding Tests:** Ensures that encoded addresses correctly round-trip through decoding.
- **Standard Address Tests:** Validates correct parsing and serialization of standard Monero addresses.
- **Integrated Address Tests:** Ensures proper handling of embedded payment IDs.
- **Subaddress Tests:** Validates the ability to generate and recognize subaddresses.
- **Featured Address Tests:** Ensures correct parsing and serialization of extended features.
- **Vector-Based Tests:** Uses predefined address vectors to ensure consistency with expected values.

The test suite ensures that all address types are handled correctly and provides coverage for both expected and edge cases.

11.7 Findings

1. **Use of `expect` in Internal Helpers (Informational)**
   Both `encoded_len_for_bytes` and the `encode`/`decode` loops employ `expect()` on conversions that are *provably* in-range once Rust's minimum pointer width (≥ 16 bits) is assumed. Because the value being indexed is reduced modulo 58 on every iteration, an out-of-bounds index is unreachable. Consequently, no adversarial input can trigger a panic on supported platforms. No code change recommended.

12 `monero-clsag` (v0.1.0)

The `monero-clsag` crate implements Compact Linkable Spontaneous Anonymous Group (CLSAG) signatures and provides a FROST-inspired threshold signing mechanism. The implementation consists of two main components:

12.1 Core CLSAG Implementation

`ClsagContext`
Holds the context needed for signing:

- A commitment opening (mask and amount)
- Selected decoy positions and ring

`Clsag` Signature
A signature consisting of:

- D: Difference of commitment randomness scaling the key image generator

- s: Vector of responses for each ring member
- c1: First challenge in the ring

Core Algorithm

The core signing/verification algorithm:

1. Takes ring members, key image I, pseudo-out P, and message m
2. For signing:
   - Generates random nonces for the real signer
   - Computes the key image generator from the public key
   - Calculates initial commitments A and AH
3. For both signing and verifying:
   - Computes challenges using `Keccak256`
   - Performs ring calculations over the Ed25519 group
   - Validates signature components

12.2 FROST-Inspired Threshold Signing

`ClsagMultisig`

Implements threshold signing for CLSAG:

- Uses FROST key generation and coordination
- Extends CLSAG to support threshold key images
- Preserves CLSAG's linkability property

Key Components

- **`ClsagMultisigMaskSender`**: A channel for communicating commitment masks.
- **`ClsagAddendum`**: Key image shares produced during signing.
- **`Interim`**: Stores partial signature data during the protocol.

Protocol Flow

1. Initialize with a transcript and CLSAG context
2. Share and aggregate key image contributions
3. Generate and share nonces via FROST
4. Produce partial signatures
5. Verify shares and construct the final signature

Security Properties

- Maintains unforgeability of CLSAG
- Preserves one-time key image property
- Requires a threshold of signers to complete
- Uses transcript-based challenge generation

The implementation leverages the `curve25519-dalek` library for Ed25519 operations and includes comprehensive test vectors covering both single-signer and threshold signing scenarios. All operations maintain constant-time properties to prevent timing side-channels.

12.3 Findings

1. **Transcript and Challenge Derivation Deviations (Medium Severity):** The transcript is constructed with fixed prefixes and ring-member data (see `core()` in `lib.rs`, lines 96–123), but it omits global output indexes or other transaction identifiers. Cryptographically, missing parameters can reduce domain separation, potentially allowing replays or confusion across different transactions.

   Recommendation: Include relevant transaction parameters in the transcript and clearly document which fields are hashed. This mitigates potential malleability by tying each challenge more tightly to unique context data.

2. **Inefficient and Fragile Channel Implementation (Medium Severity):** The mask channel uses an `Arc<Mutex<Option<Scalar>>>`, which introduces synchronization overhead. A code comment explains that a oneshot channel was not an option in `std`, revealing a design trade-off.

   Recommendation: Refactor to use a oneshot-like mechanism from an external crate for more efficient, lock-free communication. If `Arc<Mutex<...>>` remains, add clarifying comments about potential concurrency pitfalls.

3. **Mask-channel `unwrap()` can panic on protocol misuse (Medium Severity):** In `ClsagMultisig::process_addendum` the line `self.mask = Some(self.mask_recv.take().unwrap().recv());` assumes the mask receiver is present and the underlying mutex is unpoisoned. In the normal single-use flow this is safe, but if `process_addendum` is called twice or after a prior panic, the `unwrap()` will panic.

   Recommendation: Guard the call with a descriptive `expect(...)` or return a `Result` so that out-of-order invocation cannot cause a panic.

13 `monero-simple-request-rpc` (v0.1.0)

Purpose The `monero-simple-request-rpc` crate provides an HTTP(S)-based transport layer for performing RPC calls to a Monero daemon. This crate is designed to be minimal and efficient, avoiding dependencies on larger libraries like `reqwest`, while supporting both authenticated and unauthenticated connections. The crate implements the `Rpc` trait from the `monero-rpc` crate.

Internal Dependencies

- `monero-rpc (v0.1.0)`: Provides the `Rpc` trait and associated error types.
- `simple-request`: Handles HTTP(S) request and response processing.

- `digest-auth`: Implements HTTP Digest Authentication for authenticated connections.
- `tokio`: Used for asynchronous operations and synchronization primitives.
- `hex`: Used for encoding and decoding hexadecimal data.

Structure The crate is a standard library crate with its entry point at `/rpc/simple-request/src/lib.rs`. The main component is the `SimpleRequestRpc` struct, which encapsulates the connection logic, authentication handling, and request processing. The implementation includes:

- **Authentication Handling**: Supports both authenticated and unauthenticated RPC connections. Authentication uses HTTP Digest Authentication when credentials are provided in the URL.
- **Thread Safety**: Uses `Arc<Mutex>` for safe concurrent access to authentication state and nonce management.
- **Request Processing**: Sends RPC requests and processes responses, including retries for stale authentication challenges.
- **Timeout Management**: Allows for customizable request timeouts, defaulting to 30 seconds.

Detailed Functionality

`SimpleRequestRpc::new` Creates a new `SimpleRequestRpc` instance with a default timeout of 30 seconds. If the provided URL contains credentials, they are parsed and used for Digest Authentication.

`SimpleRequestRpc::with_custom_timeout` Similar to `new`, but allows specifying a custom timeout duration. It parses the URL for authentication details and initializes the appropriate client.

`SimpleRequestRpc::post` Implements the `post` method from the `Rpc` trait. It sends a POST request to the specified RPC route with the provided payload, handling retries for authentication failures due to stale challenges.

`SimpleRequestRpc::inner_post` An internal method that performs the actual request logic. It handles both authenticated and unauthenticated requests and processes the response to extract the body or detect errors.

`SimpleRequestRpc::digest_auth_challenge` Extracts and parses the `WWW-Authenticate` header from the server response to initialize or update the Digest Authentication state.

`Authentication Retry Logic` Implements a two-attempt retry mechanism for authentication failures, with specific handling for stale nonces and connection errors. This retry logic is important for maintaining connection stability during authentication state changes.

Authentication State Management The authentication state is managed through an internal `Authentication` enum with two variants:

- `Unauthenticated`: Contains a single `Client` instance for all requests.
- `Authenticated`: Contains a username, password, and a thread-safe connection state managed through `Arc<Mutex<(Option<(WwwAuthenticateHeader, u64)>, Client)>>`.

Testing Integration tests are located in `/rpc/simple-request/tests/tests.rs`. These tests validate:

- RPC functionality for retrieving blockchain height, blocks, and hardfork versions.
- Block generation with different amounts of blocks (1 and 5).
- Authentication handling for valid and stale challenges.
- Decoy RPC functionality and output distribution queries.
- Sequential test execution using `LazyLock<Mutex<()>>` to prevent test interference.
- Error handling for various failure scenarios.

Error Handling The implementation provides comprehensive error handling for:

- Connection failures with detailed error messages.
- Authentication failures, including stale nonce detection.
- Invalid or malformed responses.
- Timeout errors with a configurable duration.
- URL parsing and validation errors.

Summary The `monero-simple-request-rpc` crate is a lightweight, efficient solution for connecting to a Monero daemon via HTTP(S). It adheres to the `Rpc` trait interface, enabling seamless integration with the broader `monero-rpc` ecosystem. Its design prioritizes thread safety, robust authentication handling, and comprehensive error management, making it suitable for both development and production use cases.

13.1 Findings

1. **Secure Credential Storage:**
   Authentication credentials are parsed directly from the URL and stored as plain strings. There is no mechanism to ensure that these sensitive values are securely zeroized when no longer needed.

   > **Recommendation:** Adopt secure types such as `Zeroizing<String>` to manage sensitive credential data.

2. **TLS Configuration and Response Content Validation:**
   TLS settings are not explicitly configured, and the response processing does not verify the `Content-Type` header. This might allow processing of unintended or malformed responses.

   > **Recommendation:** Explicitly configure TLS verification and validate the `Content-Type` header in responses.

3. **Legacy Code and Test Coverage:** There is legacy, commented-out code within the response processing routine and test coverage for error scenarios is limited. Maintaining such code can obscure the intended functionality and complicate maintenance.

   > **Recommendation:** Remove any legacy code segments and extend test coverage to include edge cases and error conditions.

14 `monero-borromean` (v0.1.0)

This crate provides a Borromean ring signature–based approach to a 64-bit range proof within the Monero protocol. It defines two primary types:

- `BorromeanSignatures`
- `BorromeanRange`

Both types are specialized to operate with 64-bit range proofs and rely on Curve25519-based group operations to construct and verify Borromean ring signatures.

The crate is generally well-structured and follows best practices such as zeroizing sensitive data. However, the following sections detail both positive observations and several actionable findings, including two bugs uncovered during review (one reachable panic condition and a subtle consensus-related edge case).

Data Structures

- `BorromeanSignatures`
  - Stores exactly 64 Borromean ring signatures in two arrays of scalars, `s0` and `s1`, each of length 64.
  - Holds the final challenge scalar `ee`.
  - Implements custom `read` and `write` methods using the `monero-io` crate to produce or consume canonical byte encodings.
  - Its `verify` function checks correctness by:
    (a) Performing an iterative double-scalar multiplication on each signature component.
    (b) Accumulating hash outputs in a transcript.
    (c) Ensuring the final hash matches `ee`.
- `BorromeanRange`

- Contains a `BorromeanSignatures` field and 64 `bit_commitments`, each an `EdwardsPoint`.
- `read` and `write` functions available for serialized I/O.
- Its `verify` method checks that:
  (a) The sum of all `bit_commitments` equals the provided `commitment`.
  (b) For each bit, subtracting `H_pow_2`[i] from `bit_commitments`[i] produces a second set of points.
  (c) The embedded `BorromeanSignatures` correctly verifies under those two sets of points.

Purpose and Scope This crate provides an older, yet still relevant, Monero-based range proof system. While newer systems like Bulletproofs have largely supplanted Borromean proofs in many contexts, the `monero-borromean` crate is crucial for maintaining backwards compatibility with legacy proofs and for verifying archived transactions that used Borromean range proofs. It accepts both `std` and `no_std` environments, and depends on a few of the core Monero crates:

- `monero-io` for reading and writing typed data.
- `monero-generators` for fixed-base scalar multiplication with H^2.
- `monero-primitives` for cryptographic primitives such as `UnreducedScalar`.

Verification Logic

(i) `BorromeanSignatures::verify` loops over 64 "bits" (i.e., the separate ring signatures). For each bit:
  - It checks the challenge scalar by computing intermediate elliptic-curve multiplications.
  - It accumulates partial results into a 2048-byte transcript.

(ii) Finally, if the scalar derived from hashing the transcript matches the stored scalar `ee`, the signature is valid

14.1 Findings

In reviewing `monero-borromean (v0.1.0)`, we note that it is generally well-structured and clear in its focus on Borromean-based range proofs. However, several issues emerged during the audit:

1. **Transcript Construction Fragility:**
   The fixed-size (2048-byte) transcript is adequate for the legacy Borromean format, but makes future extension awkward. Consider a length-prefixed or streaming transcript scheme.

2. **Insufficient Rationale for Custom Scalar Handling:**
   A short design note explaining why `UnreducedScalar` is needed would help future maintainers.

15 `monero-mlsag` (v0.1.0)

The `monero-mlsag` crate provides functionality for Multilayered Linkable Spontaneous Anonymous Group (MLSAG) signatures as used in the Monero protocol. The crate is organized around two primary data structures: `RingMatrix` and `Mlsag`.

15.1 Ring Matrix

Structure

The `RingMatrix` type encapsulates a matrix of Edwards points used for MLSAG verification:

- Internal representation: `Vec<Vec<EdwardsPoint>>`.
- Zeroizes on drop for security.
- Must contain at least 2 ring members.
- All members must have equal length.

Construction Methods

1. `new`: Creates a ring matrix from a pre-formatted vector of vectors.
   - Validates matrix dimensions.
   - Ensures minimum ring size of 2.
   - Ensures consistent member lengths.
2. `individual`: Constructs a ring matrix for single output verification.
   - Takes a ring of `[EdwardsPoint; 2]` arrays.
   - Takes a pseudo-output point.
   - Subtracts the pseudo-output from the second column.

Utilities

- `members()`: Returns the count of ring members.
- `member_len()`: Returns the length of each member vector.
- `iter()`: Provides an iterator over matrix members as slices.

15.2 MLSAG Signature

Structure

The `Mlsag` type represents a complete MLSAG signature:

- `ss`: A matrix of response scalars (`Vec<Vec<Scalar>>`).
- `cc`: A challenge scalar.
- Implements zeroization for security.

Serialization

Provides binary serialization methods.

- `write`: Serializes to a writer.

  - Writes the `ss` matrix elements.
  - Writes the `cc` challenge.
- `read`: Deserializes from a reader.
  - Takes the expected mixin count.
  - Takes the expected width of the `ss` matrix.
  - Reconstructs the signature structure.

Verification
The `verify` method validates an MLSAG signature.

1. Input validation.
   - Validates that the key image count matches the ring member length minus 1.
   - Ensures consistent matrix dimensions.
   - Validates key image properties (non-identity, torsion-free).
2. Challenge reconstruction.
   - Maintains a message buffer for hash computation.
   - Iterates through ring members and key images.
   - Computes $L = sG + c_i P$ for each entry.
   - For linkable layers, computes $R = s\mathcal{H}_p(P) + c_i I$.
   - Updates the challenge using `Keccak256`.
3. Final verification.
   - Checks that the reconstructed challenge matches the signature.
   - Returns a `Result` indicating validity.

15.3 Aggregate Ring Matrix Builder

Purpose
The `AggregateRingMatrixBuilder` facilitates the construction of ring matrices for aggregate signatures.

- Manages key ring vectors.
- Tracks amount commitments.
- Handles pseudo-output calculations.

Construction
Created with transaction outputs and a fee.

- Takes a slice of output commitment points.
- Takes the fee amount as a `u64`.
- Computes the initial sum of outputs.

Ring Addition
The `push_ring` method builds the matrix incrementally.

- Validates ring dimensions.

- Separates key and amount components.
- Updates running sums.

Finalization

The `build` method produces the final `RingMatrix`.

- Combines key and amount components.
- Validates the final matrix structure.
- Returns the complete ring matrix.

15.4 Error Handling

The crate defines the `MlsagError` enum for various failure modes.

- `InvalidRing`: Ring size or structure issues.
- `InvalidAmountOfKeyImages`: Incorrect key image count.
- `InvalidSs`: Response matrix dimension mismatch.
- `InvalidKeyImage`: Invalid key image properties.
- `InvalidCi`: Challenge verification failure.

16 `monero-serai` (v0.1.4-alpha)

Overview

The `monero-serai` crate (v0.1.4-alpha) is a *protocol-level* library: it defines and validates the core Monero data structures (transactions, RingCT proofs, blocks, merkle trees, etc.) and the accompanying cryptographic routines. It *does not* perform wallet duties such as key storage, balance tracking, or multisig coordination; higher-level crates build on it for those capabilities. Internally it relies only on its own sub-modules (e.g. `/src/transaction.rs`, `/src/ringct.rs`, `/src/ring_signatures.rs`) and may be paired with `monero-rpc (v0.1.0)` when chain I/O is required.

Structure

A single `lib.rs` entry point exposes several modules:

- **`/src/transaction.rs`**: parsing, serialization and hashing of `Transaction::V1` and `Transaction::V2`.
- **`/src/ringct.rs`**: RingCT data types, Bulletproof/Bulletproof+ handling, CLSAG/MLSAG/Borromean helpers.
- **`/src/ring_signatures.rs`**: legacy ring-signature definitions and verification.
- **`/src/block.rs`**: block header parsing, merkle-root calculation and block hashing.
- **`/src/merkle.rs`**: internal helper for merkle-root computation.

All modules are `no_std`–friendly and guarded by exhaustive tests under `networks/monero/src/tests`.

Functionality

Scanning and Balance Tracking Uses `/wallet/src/scan.rs` to process newly discovered blocks or transactions, checking each output's one-time address against the wallet's view-spend key pair. Any matching outputs are recorded as unspent, with RCT amounts decrypted via the view key.

Building Transactions

1. *Select Inputs*: Gather unspent outputs sufficient to cover the send amount plus fees.
2. *Fetch Decoys*: Leverage `/wallet/src/decoys.rs` to sample ring members, ensuring ring sizes meet protocol requirements.
3. *Form Outputs*: Create `Output` structures for each recipient or subaddress, and optionally embed data in the `extra` field.
4. *Assemble & Sign*: Construct a `Transaction::V2` with RCT proofs (bulletproof or CLSAG) for confidentiality, then sign using either:
    - Single-signer mode with a local spend key.
    - Multisig mode via `/wallet/src/send/multisig.rs` if threshold signing is required.
5. *Broadcast*: Serialize and send the transaction to the network through `monero-rpc (v0.1.0)`, then wait for confirmation.

Threshold (Multisig) Workflow If configured for multisig, the crate coordinates partial signatures among participants, each owning a share of the private spend key. Using a FROST-like protocol from `monero-clsag (v0.1.0)`, it securely combines these partial signatures into a final valid CLSAG ring signature, preserving linkability and unforgeability.

Transaction Locking and `Timelock`

Monero transactions have a standard (consensus-enforced) lock window of 10 blocks, ensuring outputs cannot be spent immediately. Additionally, transactions may specify an extra *timelock* via the `Timelock` enum, imposing an additional constraint:

- **`None`**: No extra lock beyond the default 10-block lock.
- **`Block(usize)`**: Locked until a specified block height.
- **`Time(u64)`**: Locked until a specific timestamp (in seconds since epoch).

When constructing a transaction, `monero-serai` (v0.1.4-alpha) can optionally set this extra timelock. The wallet must ensure that any unlock time reflected in the `unlock_time` field matches the intended timelock policy.

View Tag for Accelerated Scanning

Recent Monero protocol versions introduce a *view tag* for outputs, stored in `Output::view_tag` (an optional `u8`). This view tag helps accelerate wallet scanning by letting wallets skip outputs that clearly do not belong to them. The `monero-serai` crate uses this when scanning blocks/transactions—if a view tag is present, it can quickly check whether the wallet's private view key is relevant before performing more expensive operations.

Legacy (v1) and Modern (v2) Transactions

While `Transaction::V2` (RingCT) transactions predominate on the Monero network, some legacy blocks contain `Transaction::V1`. The `monero-serai` crate must:

- Correctly parse older `V1` transactions (with legacy ring signatures) during chain scanning.
- Always create modern `V2` transactions with RingCT (currently CLSAG/Bulletproofs) when building new transactions.

Internally, `monero-serai` relies on version checks and the underlying `monero-serai` primitives to ensure compatibility with both formats, even though new outgoing transactions will follow `V2`.

Handling `Block` Data Internally

Although `monero-serai (v0.1.4-alpha)` focuses on wallet-level functionality, it necessarily processes parts of the block structure for scanning. Key points:

- Reading the `BlockHeader` (e.g. `timestamp`, `previous` hash) to track height or handle reorgs.
- Extracting the `miner_transaction` (coinbase) and `transactions` field to iterate over relevant `Transaction` data.
- Computing or verifying a block's hash if needed for chain validation (typically offloaded to `monero-rpc`, but the code structures allow local checks).

This is done under the hood in the `scan-module` or related scanning routines whenever direct block data is obtained from a node or test fixture.

Security Considerations

- **Zeroization**: Sensitive key material, partial signatures, and ephemeral masks are zeroized upon drop to mitigate leakage.
- **Decoy Selection Integrity**: When used by a wallet, selection routines respect consensus rules (e.g. locked-output windows) and sample decoys via the standard gamma distribution, mitigating age-based heuristics.
- **Compatibility**: Relies on `monero-serai (v0.1.4-alpha)` transaction structures, ensuring correct signature generation under Monero's consensus rules.

- **Multisig Safety**: The FROST-like approach for CLSAG is validated by multiple test vectors and reviews, mitigating partial signature forgery.

16.1 Findings

1. **Documented Protocol Exception (Block 202 612)**

   The constant hash substitution for block 202 612 reflects Monero consensus behaviour. No functional change is suggested, but adding a short code comment citing the upstream reference will help future maintainers.

   > **Recommendation:** Reexamine this exception. Either document it comprehensively (including rationale and potential consequences) or provide a configurable mechanism that allows operators to disable or adjust this behavior.

16.2 `monero-wallet (v0.1.0)`

This crate provides high-level Monero wallet functionality built on top of the `monero-serai` and `monero-rpc` crates. Its entry point is at `/wallet/src/lib.rs`, with tests under `/wallet/src/tests/`. It encompasses scanning, transaction creation, decoy selection, and optional threshold multisignature (FROST-like) signing. The main modules are:

- **`scan.rs`**: Detects and records wallet outputs from blocks or transactions. Uses a `Scanner` or a `GuaranteedScanner` (the latter referencing a `GuaranteedViewPair`) to identify owned outputs and respect potential additional `timelock` constraints. Also decrypts payment IDs and subaddress details.

- **`view_pair.rs`**: Holds the `ViewPair` type (public spend key + private view key), plus a `GuaranteedViewPair` for advanced "burning-bug" safety. Includes subaddress derivation and address construction (e.g. integrated addresses). Sensitive key material is zeroized when dropped.

- **`decoys.rs`**: Handles decoy selection for ring signatures by querying a node for zero-amount outputs, filtering for unlocked outputs, and combining them with real inputs to form rings. Provides `OutputWithDecoys` to bundle each spendable output with its chosen decoys.

- **`output.rs`**: Manages discovered wallet outputs, defining `WalletOutput` and associated metadata (e.g. `payment_id`, `subaddress_index`, `additional_timelock`). Allows the wallet to track each output's spendability status.

- **`extra.rs`**: Defines how data is embedded in a transaction's `extra` field, such as payment IDs or serialized wallet-specific information.

- **`send/`**: Organizes transaction creation and signing. Key submodules include:

  - `tx.rs`: Orchestrates assembling inputs (with decoys), outputs (with Bulletproof or Bulletproof+ range proofs), fees, and final signatures. Builds `V2` RingCT transactions and ensures valid CLSAG signatures.

  – `multisig.rs`: Implements a threshold-signing protocol using FROST-like coordination for CLSAG-based transactions. Partially signed transactions can be combined securely among multiple participants.
  – `tx_keys.rs`: Manages ephemeral transaction keys. Ensures each input's key image is deterministically tied to the real spend key and ephemeral randomness. Zeroizes sensitive data on drop.
  – `eventuality.rs`: Defines an `Eventuality` that can describe the state or "intended outcome" of a partially or fully built transaction. Facilitates checking whether an on-chain transaction matches the wallet's intended outputs.

Key Features and Workflow

1. **Scanning and Balance Tracking.** The wallet periodically scans blocks/ transactions to locate outputs whose one-time addresses match its view key. Identified amounts (including RingCT data) are decrypted, tracked, and updated as spendable or locked depending on any `timelock`.
2. **Decoy Selection.** When forming a transaction, the wallet selects decoys from the chain to ensure a valid ring signature with the desired ring size. It relies on RPC calls (via `monero-rpc`) to fetch output distributions and confirm decoys remain unlocked.
3. **Transaction Building.** Inputs are chosen among available outputs, Bulletproof or Bulletproof+ commitments are created to hide amounts, and CLSAG ring signatures are generated for each input reference. The code supports either single-signer or a FROST-based threshold approach.
4. **Optional Threshold Signing.** If configured, multiple partial signatures are coordinated among participants. Final combination yields a valid CLSAG signature without exposing individual private spend keys.
5. **Broadcast and Confirmation.** Completed transactions are serialized and published through the `monero-rpc` crate. The wallet can then monitor subsequent blocks to detect confirmation and refresh spend statuses.

17 Findings

17.1 Panic Conditions and `unwrap`

Description In multiple areas of code, `unwrap` is used in production. If unexpected input is processed (e.g., from an adversarial node response or a malformed block), `unwrap` could cause a runtime panic.

Locations

- `send/tx.rs`: In the function `weight_and_necessary_fee` (approximately lines 248–264), the fee computation and related conversion (e.g. calls to `varint_len(...).unwrap()` and subsequent unwrapping within the fee–selection loop) are performed without error propagation.

Recommendation: Refactor `unwrap` calls in `weight_and_necessary_fee` and `AbsoluteId::read` into structured error handling (e.g., using `?` or `expect`) so that errors are properly returned to the caller.

17.2 Test Coverage and Edge Cases

Description The test suite covers standard usage scenarios; however, it does not fully account for adversarial or heavily customized node responses. For instance, simulating conditions where `get_output_distribution` returns an almost empty list or where block times are unexpected might reveal weaknesses in the decoy selection or scanning code.

Locations

- `tests/decoys.rs`: Standard tests for decoy selection are implemented (approximately lines 45–90), but they only simulate typical node behavior.
- `tests/scan.rs`: Scanning tests (around lines 30–75) verify normal output retrieval but do not simulate adversarial block or RPC responses.

Recommendation: Expand integration tests with simulated or mocked adversarial node data. This will help reveal potential issues in how the library responds when decoy ring sizes are not satisfiable or when distribution data is incomplete.