Serai code audit

Cypher Stack*

March 16, 2023

Contents

*`https://cypherstack.com`

1 Overview

1.1 Introduction

Serai is a distributed exchange protocol and Rust implementation. This report represents an audit of specific areas of underlying cryptographic libraries and functionality in the Serai codebase.

Serai and Cypher Stack identified the following goals for this audit:

- Assert that implementations of external specifications or protocols are done correctly
- Determine if malicious or unintended edge cases can be exploited
- Identify cases where the code may panic unexpectedly
- Note situations where documentation is insufficient to convey intent or design
- Ensure that secret data is handled as safely as is feasible with respect to memory clearing and constant-time operations
- Identify areas of the implementation where efficiency gains are possible and reasonable
- Determine the extent to which the implementation contains relevant and comprehensive tests

1.2 Summary of findings

Overall, the implementation is well designed and carefully written with secure deployment in mind. We did not identify any findings of particular immediate concern.

Consequently, the findings described in this report primarily identify recommendations for improvement. These deal with inconsistencies, documentation, tests, or areas of the codebase where fixes or changes could mitigate future issues.

However, we carefully note that we do not assign severity ratings to the issues contained herein. Such ratings are often subjective, and typically depend

on the ease or difficulty of triggering specific behavior, which can be challenging to assess in a consistent way.

2 Scope

Specific directories within the codebase are in scope for this audit. Each listed directory is given relative to the repository tree at a specific commit on the `crypto-audit` branch of the Serai repository[1]. Serai issued fixes via subsequent commits on the `crypto-audit` branch, which we discuss as part of our findings.

Separately, later updates leading to a specific commit on the `crypto-tweaks` branch of the repository[2] were reviewed, after which another fix was issued via a commit on the `crypto-tweaks` branch, as discussed later.

2.1 `crypto/ciphersuite`

Review of this directory should determine if it provides correct wrappers for particular elliptic curves. The `ed448` implementation is out of scope.

Additionally, it should be determined if the hash-to-field implementations match intended designs.

- For `ristretto255` it should match an IETF draft specification[3] instantiated with `SHA2-512`
- For `ed25519` it should match IETF RFC 8032[4]
- For `secp256k1` and `P-256`, it should match an IETF draft specification[5] locked to a single element without DST validation

2.2 `crypto/dalek-ff-group`

Review of this directory should determine if it is a valid `curve25519-dalek` wrapper for `ff/group` bindings.

2.3 `crypto/dkg`

Review of this directory should determine if it is a correct implementation of the key generation protocol defined in the FROST preprint[6].

It should additionally determine if the added encryption functionality and blame design appear to be implementated correctly and safely. Authenticated

[1] `https://github.com/serai-dex/serai/tree/eeca440fa7faf5c6b3c72c225bb18f256e34e0bd`

[2] `https://github.com/serai-dex/serai/tree/62dfc63532f1dbd97ea1273ae9ac9f0761e94ac8`

[3] `https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-decaf448-05.html#name-scalar-field`

[4] `https://datatracker.ietf.org/doc/html/rfc8032`

[5] `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16`

[6] `https://eprint.iacr.org/2020/852`

encryption is specifically not addressed here. There is documentation[7] with informal design intent.

2.4 crypto/dleq

Review of this directory should determine if it is a correct implementation of the discrete logarithm equality proof defined in a paper by Chaum and Pedersen[8].

It should also determine if the included merged-challenge proof for multiple statements is implemented correctly.

The cross-group proving system is out of scope.

2.5 crypto/ff-group-tests

Review of this directory should determine if it represents relevant and correct tests of generic field and group functionality.

2.6 crypto/frost

Review of this directory should determine if it is a correct implementation of version 12 of the FROST IETF draft[9]. It should additionally determine if the extensions described in the documentation[10] are implemented correctly and safely.

2.7 crypto/multiexp

Review of this directory should determine if it is a correct implementation of Straus- and Pippenger-type multiscalar multiplication algorithms. It should additionally determine if the batch verification is implemented correctly and safely. For the Straus-type algorithm, the information at doi:10.2307/2310929[11] may be useful. For the Pippenger-type algorithm, the information in this article[12] may be useful.

2.8 crypto/schnorr

Review of this directory should determine if it is a correct implementation of standard Schnorr signatures, using the approach from IETF RFC 8032[13] with a modular challenge function. It should additionally determine if the batch

[7] https://github.com/serai-dex/serai/blob/develop/docs/cryptography/Distributed%20Key%20Generation.md

[8] https://chaum.com/wp-content/uploads/2021/12/Wallet_Databases.pdf

[9] https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-12.html

[10] https://github.com/serai-dex/serai/blob/develop/docs/cryptography/FROST.md

[11] https://doi.org/10.2307/2310929

[12] https://scholarship.claremont.edu/cgi/viewcontent.cgi?article=1141&context=hmc_fac_pub

[13] https://datatracker.ietf.org/doc/html/rfc8032

verification implementation is correct and secure, and if the aggregation matches the half-aggregation technique from this preprint[14].

2.9 `crypto/transcript`

Review of this directory should determine if it appears to represent safe designs for transcripting. There are no particular specifications for these designs.

More specifically, all designs are intended to represent reasonable behavior for secure transcripts to be uses in Fiat-Shamir applications. The provided `RecommendedTranscript` design is additionally intended to be secure against an attack where an adversary in possession of a transcript challenge is able to produce a valid continuation of the underlying transcript. The more general `DigestTranscript` design is intended to be secure against hash-based attacks to the same degree as the underlying hash function with which it is instantiated.

3 Findings

In the subsequent findings, we include actions taken and responses provided by Serai in response to an initial version of this report. Where listed, commits based on these actions refer to the `crypto-audit` repository branch, and function as hyperlinks if supported by your document reader software.

3.1 `crypto/ciphersuite`

This crate provides wrappers for elliptic curve groups, their associated scalar fields, and hashing operations.

3.1.1 Incomplete documentation for hash-to-field constants

The implementation for the `secp256k1` and `P-256` elliptic curve groups follows an IETF draft specification[15] for hashing uniformly to each curve's scalar field. Doing so requires careful configuration of constants relating to the desired security level and scalar field modulus. While the implementation sets up these constants correctly, it does so partly by the use of "magic numbers" and the use of big-integer structures whose interaction is not documented.

Recommended fix is to document the derivation of these constants and their relationship to each other for improved clarity.

Action: Addressed in commit `18ac806`.

3.1.2 Duplicated domain separation tag overflow handling

When hashing to the scalar field of the `secp256k1` and `P-256` elliptic curve groups, the implementation first hashes a domain separation tag with an IETF-specified prefix, and then uses this hash output if the tag's length exceeds 255

[14] `https://eprint.iacr.org/2021/350`

[15] `https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-16.html`

bytes. In addition to performing this initial hash even if the tag's length does not overflow, the functionality is unnecessary to implement. The `elliptic_curve` library, which is used for internal hashing operations, includes functionality via `Domain::xmd` that performs this hashing, and does so only when needed.

Recommended fix is to refactor the implementation to use this library's functionality for domain separation tag overflow handling.

Action: Addressed in commit `ac0f5e9`.

3.1.3 Unnecessary wrapping addition

When computing the modulus for reduction of the output used for hashing to the scalar field of the `secp256k1` and `P-256` elliptic curve groups, the implementation performs a wrapping addition within a `U386` big-integer instantiation. Because of the padding and length limits used in this instantiation, it is not possible for the wrapping addition to overflow and wrap; as a result, using a checked addition may suffice and be more efficient.

Recommended fix is to consider the use of a checked addition for this purpose. However, its use should be carefully documented to avoid unsafe behavior that may be caused by future changes.

Action: Addressed in commit `cb4ce5e`.

3.1.4 Hashing to scalar may collide

The implementation's hash-to-scalar functionality for the `ristretto255` and `ed25519` elliptic curve groups uses a modular reduction of domain-separated `SHA-512` output. However, the domain separation tag and input data are directly concatenated when hashed; the lack of tag length prepending (or equivalent functionality) means that distinct combinations of tag and input data can collide. This is required for compliance with some signature standards, but may be unsafe in other protocols.

Recommended fix is to carefully document this behavior to avoid misuse.

Action: Addressed in commit `686a5ee`.

3.2 `crypto/dalek-ff-group`

This crate implements relevant group and field traits from the `group` and `ff` crates for the `ristretto255` and `ed25519` elliptic curve groups (and their associated scalar fields).

3.2.1 Constants are unsourced and untested

The implementation defines several constants relevant to its underlying group and field structures. Most of these are effectively ported directly from the `curve25519-dalek` library, albeit with type or format differences. However, these constants do not have documented references to their library equivalents, making it more challenging to identify and confirm their accuracy.

Further, there are no tests present in the crate that assert the correctness of these ported constants. Instead, the implementation appears to assume that its broader test crate, which contains general tests for the `ff` and `group` traits across several elliptic curve groups, will identify any issues with the constants. This is not guaranteeed.

A minimum recommended fix is to document the specific source of group and field constants. A more comprehensive fix is to add tests that directly assert the correctness of ported constants against their `curve25519-dalek` library equivalents. Finally, an even more comprehensive is to generate these constants at compile time directly from their library equivalents.

Action: Addressed in commit `40a6672`.

3.2.2 Non-standard group element randomization

The implementation uses a single generic design for producing random group elements in both the `ristretto255` and (prime-order subgroup of the) `ed25519` elliiptic curve groups. It uses a non-standard random byte manipulation approach and attempts to decompress it as a group element byte representation, additionally checking for torsion. It is unclear if this approach is guaranteed to yield a uniform group element distribution.

Further, while the general approach of using rejection sampling on a compressed Edwards point via its y-coordinate is standard, it is not standard for `ristretto255`. The underlying `curve25519-dalek` library already contains functionality for transforming uniform byte data to group elements via a dual application of an `Elligator` mapping.

Recommended fix is to separately implement group element randomization: use standard rejection sampling without byte manipulation for `ed25519`, and use the underlying library functionality for `ristretto255`.

Action: Addressed in commit `d929a8d` using a hashing method for consistent implementation.

3.2.3 Group element randomization can yield identity

The implementation's design for group element randomization can yield the identity element for both `ed25519` and `ristretto255`. However, the `Group` trait specification for the `random` function requires that the identity element not be produced.

Recommended fix is to check for and reject the identity element in each group's randomization functionality.

Action: Addressed in commit `74647b1`.

3.2.4 Unused and public field constants

The field implementation contains a public constant `EDWARDS_D` that is unused. It may have been intended for use by an internal test that was not added, or intended for use by implementers.

On a related note, the constants `MOD_3_8` and `MOD_5_8`, which are used internally, are also public.

If the constants are not intended for external use outside the trait requirements, recommended fixes are to remove the unused constant and make the others private.

Action: Addressed in commit `40a6672`.

3.2.5 Missing documentation

The field implementation's `sqrt_ratio_i` function is not documented, and is not a trait requirement.

Recommended fix is to add documentation.

Action: Addressed in commit `40a6672`.

3.3 `crypto/dkg`

This crate provides a framework for distributed key generation intended for use in threshold signing operations. It also provides an instantiation of the framework for FROST key generation.

3.3.1 Unnecessary fallible type conversion in Lagrange coefficients

When computing Lagrange coefficients, the implementation uses a `try_from` and `unwrap` approach to convert `u16` indexes to `u64` before being converted to scalar field elements as part of the coefficient computation. This is unnecessary, as the operation is infallible.

Recommended fix is to use `from` to perform the conversion instead.

Action: Addressed in commit `d72c4ca`.

3.3.2 Unnecessary stream cipher nonce

When encrypting data for other participants, the implementation uses a transcript as a key derivation function to generate a `ChaCha20` key and nonce. While the key and nonce are uniformly distributed due to the transcript design, they are deterministic from an underlying Diffie-Hellman shared secret bound to the transcript. In particular, this means that any transcript using the same initial binding will produce the same key and nonce, negating the effect of the nonce altogether. It is equally safe in this case to use a fixed nonce (for example, zero).

Additionally, it should be carefully documented that with either a deterministic or fixed nonce, a Diffie-Hellman shared secret must never be reused. The implementation uses an ephemeral key to derive a new shared secret on each call to the `encrypt` function, and is therefore safe; however, updating the documentation for the underlying `cipher` function call may avoid unsafe use arising from future updates.

Recommended changes are to use a fixed nonce, and update the `cipher` function documentation.

Action: Addressed in commit `8bee626`.

3.3.3 Lack of low-level guards against polynomial evaluation at zero

When performing Shamir-based secret distribution where a (share of a) secret value is encoded as the evaluation of a polynomial with random coefficients at zero, it is essential never to reveal this evaluation unintentionally. Violations of this principle have occurred in high-profile open implementations[16] in various ways.

While the implementation of FROST distributed key generation, which uses this kind of secret encoding, is careful to ensure that higher-level functionality never uses zero as a participant index for polynomial evaluation, there is an engineering risk of mistakenly doing so in future updates. Indeed, participant indexing for evaluation is done via range looping, and it is well-known programmer lore that incorrect loop indexing bounds can be easy to miss. Further, it could be possible that a misbehaving participant is able to trick an honest player into performing such an evaluation if a future update made this exploitable (to be clear, we have not identified such a risk in the implementation).

One mitigation may be to ensure that polynomial evaluation, which is done using Horner's method in the `polynomial` function, provides an assertion that the provided partipant index be nonzero. Since this is the lowest level at which this index could be checked, such a check would provide a reliable guard against zero indexing.

A more robust, but intensive, guard would be to use Rust's typing system for enforcement of valid participant indexes. Defining a new participant index type could ensure that a zero index is not representable, and ensure that polynomial evaluation is only possible on valid input values without lower-level checks. This approach may more quickly catch indexing errors by the compiler, and could catch other such errors during runtime without needing to propagate errors from the polynomial evaluation itself.

Action: Addressed in commits `87dea5e` and `2d56d24`.

3.3.4 Incomplete FROST session-specific domain separation

Domain separation is used in the implementation to differentiate proofs and keys, and to bind data to key generation sessions to prevent unwanted replay. However, not all data is optimally bound to a FROST key generation session.

When a participant prepares its initial commitment message to be shared with its peers, it binds a caller-provided domain separator (recommended to be specific to the generation session) into the Fiat-Shamir challenge used in the included proof of knowledge.

Later, the participant sends to each peer an encrypted message containing share data. The key used for this encryption is derived from a transcript with a fixed domain separator, which is not explicitly bound to a session. Further, the encrypted message is accompanied by a proof of knowledge of the discrete

[16] `https://blog.trailofbits.com/2021/12/21/disclosing-shamirs-secret-sharing-vulnerabilities-and-announcing-zkdocs/`

logarithm of an included ephemeral public key, but the Fiat-Shamir challenge used for this proof also uses a fixed domain separator without session binding.

Finally, when decrypting a received encrypted message from a peer, the participant generates a particular discrete logarithm equality proof that can be used to identify a malicious peer. This proof's Fiat-Shamir challenge similarly is not bound to a session.

While we did not identify exploits arising from this due to defined abort points in FROST key generation, it may be useful to include session binding for each of these cases as a matter of good practice.

Action: Addressed in commit `4d6a0bb`.

3.3.5 Incorrect documentation

When discussing key promotion, which migrates keys between group generators in a verifiable manner, the implementation's documentation is incorrect. The documentation rather confusingly implies that this process is intended to migrate keys between elliptic curve groups, which is incorrect. The implementation initially provides a generic definition that could allow for different groups, but later applies a type restriction that requires a common group. While it may be possible to securely migrate keys in this manner using cross-group discrete logarithm equality proofs (which the repository provides, albeit experimentally), this is out of scope and does not apply.

Recommendation is to update the documentation to clarify that key promotion is currently implemented only between generators within the same group, although it may later be more fully generalized to separate groups.

Action: Addressed in commit `1a6497f`.

3.4 `crypto/dleq`

This crate implements discrete logarithm equality proofs. One such implementation is that of a generalization of a proof by Chaum and Pedersen[17] showing that sets of group elements share common discrete logarithms relative to given generators. Another such implementation, which is out of scope, is a more complex construction that operates between distinct groups.

While the implementation does not provide or cite a security proof that its construction is a sigma protocol (that is, with properties of completeness, special soundness, and special honest-verifier zero knowledge (SHVZK)), such a proof is straightforward using modern techniques.

3.4.1 Security proof

Let $\mathbb{G}$ be a cyclic group where the discrete logarithm problem is hard, and let $\mathbb{F}$ be its scalar field. Suppose we wish to make m separate discrete logarithm equality assertions, where each such assertion $j \in [1, m]$ proves that n_j group elements share a common discrete logarithm with respect to a given set of

[17] `https://chaum.com/wp-content/uploads/2021/12/Wallet_Databases.pdf`

generators. We do not require that the set of generators be the same across assertions.

The construction in the implementation is (effectively) intended to be a sigma protocol for the following relation:

$$\{(G_{j,i})_{j,i=1}^{m,n_j}, (P_{j,i})_{j,i=1}^{m,n_j} \subset \mathbb{G}; (x_j)_{j=1}^{m} | \forall j \in [1,m] \forall i \in [1,n_j] : P_{j,i} = x_j G_{j,i}\}$$

To execute the protocol, the prover does the following:

- For $j \in [1,m]$, samples $x'_j \in \mathbb{F}$ uniformly at random.
- For $j \in [1,m]$ and $i \in [1,n_j]$, sets $P'_{j,i} = x_j G_{j,i}$.
- Sends $(P_{j,i})_{j,i=1}^{m,n_j}$ to the verifier.
- Receives a uniformly-sampled challenge $c \in \mathbb{F}$ from the verifier.
- For $j \in [1,m]$, sets $y_j = x'_j + c x_j$.
- Sends $(y_j)_{j=1}^{m}$ to the verifier.

The verifier accepts the proof if and only if the equality

$$y_j G_{j,i} = P'_{j,i} + c P_{j,i}$$

holds for $j \in [1,m]$ and $i \in [1,n_j]$.

The protocol is complete by inspection.

To show the protocol is 2-special sound, fix the statement values $(G_{j,i})_{j,i=1}^{m,n_j}$ and $(P_{j,i})_{j,i=1}^{m,n_j}$. We construct an extractor that, given two accepting transcripts rewound with distinct challenges, extracts a valid witness.

Sample uniformly-random challenges $\overline{c} \neq c$, and let $(P'_{j,i})_{j,i=1}^{m,n_j}, (y_j)_{j=1}^{m}$ and $(P'_{j,i})_{j,i=1}^{m,n_j}, (\overline{y}_j)_{j=1}^{m}$ be corresponding transcripts for valid proofs. This means that for $j \in [1,m]$ and $i \in [1,n_j]$, the equalities

$$y_j G_{j,i} = P'_{j,i} + c P_{j,i}$$

and

$$\overline{y}_j G_{j,i} = P'_{j,i} + \overline{c} P_{j,i}$$

must hold Subtracting, we obtain that

$$(y_j - \overline{y}_j) G_{j,i} = P'_{j,i} + (c - \overline{c}) P_{j,i}$$

for each pair. This means that

$$P_{j,i} = \frac{y_j - \overline{y}_j}{c - \overline{c}} G_{j,i}$$

and we have an extracted witness of the correct form that satisfies the proof relation.

To show the protocol is SHVZK, fix the statement values $(G_{j,i})_{j,i=1}^{m,n_j}$ and $(P_{j,i})_{j,i=1}^{m,n_j}$. We construct a simulator that, given a uniformly-sampled challenge, can construct a valid proof transcript that is distributed identically to that of a real proof.

Sample a challenge c uniformly at random. For $j \in [1, m]$, sample $y_j \in \mathbb{F}$ uniformly at random. For $j \in [1, m]$ and $i \in [1, n_j]$, set $P'_{j,i} = y_j G_{j,i} + c P_{j,i}$.

Then $(P'_{j,i})_{j,i=1}^{m,n_j}, (y_j)_{j=1}^{m}$ is trivially an accepting transcript by construction. Further, since each y_j in a real proof is distributed uniformly at random, the simulated transcript is distributed identically.

Action: Not applicable.

3.4.2 Incomplete documentation

Several functions in the implementation lack documentation.

Recommended fix is to add documentation.

Action: Addressed in commit `6104d60`.

3.4.3 Proving system functionality may be combined

The implementation contains separate structs and functions to handle proofs for both single and multiple discrete logarithm equality assertions. It is the case that the former is algebraically a special case of the latter, with only minor transcript generalizations. Because of this, the code can be simplified significantly by treating the single-assertion case as a wrapper to the multiple-assertion functionality. Doing so may reduce future technical debt.

Action: Addressed in commit `65376e9` for verification only.

3.4.4 Prover does not check input consistency

When producing a proof for the general case of multiple discrete logarithm assertions, the implementation does not check that the number of generators and scalars provided is consistent. This can result in a proof that is invalid and will be rejected by the verifier. This issue was identified by Serai.

This behavior is not a security risk, as an adversary can always write its own malicious prover, and an invalid proof constructed by the implementation due to an input size mismatch will be correctly rejected by an honest verifier. However, returning an invalid proof to the caller may produce unexpected behavior.

One fix is to have the prover return a `Result` that can indicate this error, and let the caller decide how to proceed. Another fix is to rely on an `assert` against an input consistency check; however, this implies that the caller should perform its own consistency check to avoid an unexpected application panic. A further fix is to modify the prover function signature to accept either a slice of generator/scalar tuples, or to accept a custom input type that can handle input consistency prior to being provided to the prover. Which approach to choose depends on the implementation complexity and desired caller behavior.

Action: Addressed in commit `c1435a2`.

3.5 crypto/ff-group-tests

This crate provides a set of tests for generic fields, prime-order fields, and groups compatible with the `group` and `ff` crates' traits. The tests are then run against the groups and fields corresponding to several implemented curves used in ciphersuites, which must implement these traits. The crate does not introduce any new functionality.

3.5.1 Cyclic test structure

The codebase defines a collection of ciphersuites elsewhere, comprising the following elliptic curves:

- `ristretto255`
- `ed25519`
- `ed448`
- `secp256k1`
- `P-256`

Each ciphersuite implements the custom `Ciphersuite` trait, which in turn requires that the underlying elliptic curve and scalar field implement `PrimeGroup` and `PrimeField`, respectively. Due to library availability and dependencies for these curves, they are implemented in different ways.

Because of these differences, the tests are conducted on the supported ciphersuites in a manner that is somewhat inconsistent. The `ristretto255` and `ed25519` ciphersuites are tested as part of the `crypto/dalek-ff-group` wrapper. The `ed448` ciphersuite (which is out of scope) is tested as part of its `crypto/ed448` implementation; we note that this implementation is documented as being primarily for testing purposes. However, the `secp256k1` and `P-256` ciphersuites are tested directly by this crate, as they are defined using existing library dependencies.

The inclusion of tests for `secp256k1` and `P-256` via development dependencies serves a somewhat cyclic purpose: it allows for the use of established curve libraries to ensure this crate's tests pass, and provides additional testing for those libraries. In theory, this could introduce a failure where an implementation flaw in one of these curve libraries coincides with an incorrect test in this crate (though this may be unlikely).

Recommended change is to document the presence and intent of the curve library development dependencies.

Action: Addressed in commit `32c18ca`.

3.5.2 Tests are incomplete

Because the tests in this crate are intended for use against generic finite cyclic groups, they variously require trait bounds from the `group` and `ff` crates. Each

of these trait bounds introduces or requires various operations (generally by additional trait bounds) relevant to the group or scalar field structure under test, with the intent that a compliant curve implementation in fact have a valid group structure making it suitable for use as a ciphersuite elsewhere in the codebase. These operations include group addition and inversion, field addition and inversion, field multiplication and inversion, identity properties, equality, element representation, and interaction between group and scalar field operations.

It is complex and somewhat infeasible to properly exercise every possible operational case introduced by each underlying trait bound. However, as one intent of the crate is to be used with new or custom curve implementations, it is important to exercise as many cases as is feasible.

The following additional `Group`-related tests are recommended:

- Successive calls to `random` with a suitable random number generator yield non-identical values, in order to detect obvious randomization failure[18]
- Calling `sum` on an iterator with non-identical values produces the expected result, in order to assert that each element in the iterator is included exactly once in the resulting sum
- Calling `from_bytes` on an invalid group element representation produces the expected `None`-like result, to assert that deserialization can fail
- Calling `from_bytes_unchecked` on an invalid group element representation produces whatever result is deemed appropriate

The following additional `Field`-related tests are recommended:

- Successive calls to `random` with a suitable random number generator yield non-identical values, in order to detect obvious randomization failure
- Calling `is_odd` on a doubled even element returns the expected value
- Calling `is_odd` on a doubled odd element returns the expected value
- Calling `from_repr` on an invalid or non-canonical scalar representation produces the expected `None`-like result, to assert that deserialization can fail
- Calling `from_repr` on a valid and canonical scalar encoding, and then calling `to_repr` on the resulting scalar, should result in the same encoding; while it is not clear if this is a required trait behavior, it may be unsafe for an implementation to behave otherwise

The following existing `Field`-related tests are recommended to be changed:

- The tests for the validity of the `S` constant with respect to the given root of unity and multiplicative generator should assert that the underlying computed t value is odd, as specified by the trait documentation

Action: Addressed in commits `93f7afe` and `ed056cc` as feasible.

[18] `https://xkcd.com/221/`

3.6 `crypto/frost`

This crate provides a generic implementation of FROST signatures, with the intent of compatibility with version 12 of the corresponding draft standard.

It also extends this functionality. In particular, it is designed to accommodate other Schnorr-like signature applications by permitting more general transcripting, nonce handling, and key offsets.

3.6.1 Ambiguous handling of invalid nonce generation

When generating nonces for the first round of FROST signing, the implementation performs a check that the output from the H_3 hash function is not zero. If it is, the generation process repeats with new randomness until the output is nonzero.

We note that the IETF draft specification is poorly written as it pertains to this. According to the specification, a nonce is initially permitted to be zero. However, when the corresponding nonce commitments are serialized, if any such commitment is the group identity, the serialization produces an error that is intended to result in a protocol abort. Unfortunately, the specification is ambiguous about if (or when) this nonce commitment serialization occurs by the player generating nonces. It is already the case that deserialization of such an invalid nonce commitment by another player will correctly result in a protocol abort.

Despite these specification issues, the implementation's handling is safe and does not result in any incompatibility. The FROST preprint indicates that nonces must be uniformly sampled so as to be nonzero; since the implementation effectively performs rejection sampling, it is an accurate representation of this. A careful reading of this aspect of the preprint shows that a player sampling in this manner will never initiate such a protocol abort. So while the implementation differs from a strict reading of the specification, it does not introduce any incompatibility or risk safety in any way.

Recommended action is to note this behavior in the code for clarity.

Action: Addressed in commit `969a5d9`.

3.6.2 Nonce generation is not checked in test vectors

The draft specification includes test vectors for each supported ciphersuite. For each, the implementation generates a signature using the provided keys and message, and asserts that it is valid and matches the expected encoded value.

In addition to including each player's key share and nonce commitments, each test vector includes randomness data intended to be used to assert that compliant implementations produce suitable nonces using the standard's hash-based approach in an expected way. However, the implementation does not use these randomness values to test its nonce generation functionality.

The design of the nonce generation functionality is not currently suited to supporting the test vectors, as it accepts a reference to a cryptographically-secure random number generator used to produce the necessary randomness

(a common approach in Rust-based cryptographic protocol implementations). This would need to be refactored to instead accept byte data that is either produced by a random number generator or provided as part of a test vector.

While such a refactoring of may introduce additional engineering risk, a recommended action is to do so, and to test it against each test vector to assert compliance with the standard.

Action: Addressed in commit `7a05466`.

3.6.3 Nonce commitment encoding is not checked in test vectors

The draft specification test vectors include nonces and nonce commitments for each player in each test vector. The implementation reads and uses the nonces, but does not specifically check that the corresponding nonce commitments (which it computes itself) from the test vectors match.

While the draft standard does not specify communication details between players for each signing round, the encodings used for nonce commitments are. It may be useful to assert that the nonce commitments produced by the implementation from the nonces encode as expected.

Action: Addressed in commit `39b3452`.

3.6.4 Inconsistent use of RFC 8032 test vectors

The implementation includes a single `ed448` signature test vector, the function name of which indicates it is taken from RFC 8032. While the test vector is from RFC 8032, it should be more clearly documented as such.

Further, it is unclear why only the particular test vector was chosen for inclusion, as RFC 8032 provides additional `ed448` test vectors. Additionally, it is unclear why none of the RFC 8032 test vectors for `ed25519` are included.

Action: Addressed in commit `6a15b21`.

3.6.5 Signature test vectors are unsourced

The implementation uses JSON-encoded FROST test vectors as part of its test harness. While the vectors are taken directly from the draft specification repository[19], they are not documented with this source.

Recommend documenting the test vector source for clarity.

Action: Addressed in commit `62b3036`.

3.6.6 Incomplete documentation and terminology for nonces

Several structures and functions, particularly those relating to nonces and nonce commitments, lack documentation. This is particularly relevant since the implementation is extremely generalized relative to its handling of these constructions. Standard FROST nonce commitments are formed by taking a pair of

[19] `https://github.com/cfrg/draft-irtf-cfrg-frost`

randomly-generated scalar nonces and multiplying each by the same group generator; these are later summed using a deterministic binding factor. However, the implementation allows for much more flexibility. In the most general case, a participant may use the same nonce pair across multiple group generators, and in fact perform this operation repeatedly with multiple collections of nonce scalar pairs and group generators. Although standard-compliant FROST signatures do not use this generality, the broader design adds complexity to the overall design that warrants careful documentation.

Related to this, the documentation for the `SignMachine::from_cache` function appears to be incomplete and truncated.

Finally, the terminology used to refer to aspects of the nonce and nonce commitment functionality is somewhat relaxed, as terms like "nonce" and "generator" and "commitment" are sometimes reused or combined in subtle ways that are not particularly standardized throughout.

Recommend adding proper documentation for clarity, and considering a refactoring of nonce and nonce commitment terminology for consistency and clarity.

Action: Addressed in commit `5a3406b`.

3.6.7 Unsafe transcript is public

The implementation includes `IetfTranscript`, a public transcript construction that implements the `Transcript` trait and is intended for use in generating compliant FROST signatures. This construction exists to allow a generalization of transcripting for FROST signatures to enable safer handling of signatures within other protocols, and is designed such that the more general signature functionality reduces cleanly to the draft specification in the event that no such protocol embedding is required.

To meet its intended use case, `IetfTranscript` naively handles transcript operations in a manner that would be unsafe in other situations. For example, it ignores transcript labeling and domain separation, does not label messages, and produces empty challenges.

We note that the documentation for this construction clearly indicates that it should not be "used within larger protocols" because of its design. However, the fact that it is public seems to introduce unncessary risk of unsafe use in other contexts.

If it is not possible to restrict the visibility of `IetfTranscript`, it may be useful to introduce a marker trait indicating `Transcript` implementations that might be considered safe for general use.

Action: Addressed in commit `a42a84e`.

3.6.8 Offset splitting can be simplified

The implementation supports, as an optional feature, the ability for a group to offset its signing key by a secret value, such that verification applies a corresponding offset to the group public key. This approach, while not part of

the draft specification, is included in draft ZIP 312[20] We note that this re-randomization is still undergoing formal analysis[21], but is straightforward.

The approach used in the implementation is to have each player apply a share of the offset when producing their signing shares, and to verify the signature against an offset public group key. While the existing API may not support doing so, there are two approaches that may be used if considered both feasible and simpler.

In the first, a designated player applies the entire offset when producing its signature share. The selection of this player may be done deterministically by establishing a rule like using the player with the lowest index.

In the second, the method of ZIP 312 is used. No player applies the offset (in whole or in part) during either round of signing; however, each player includes the public key offset when producing nonce binding factors. During the aggregation phase, each player applies the offset when summing signature shares.

Recommended fix is to switch to one of these methods for simplicity if feasible (given likely required API changes). If particular signer interoperability is desired, use of the ZIP 312 method may be preferred.

Action: Addressed in commit `c6284b8`.

3.6.9 Insufficient tests

The implementation includes a variety of tests for supported cipheruites. It first tests randomized FROST threshold Schnorr signature correctness. It next creates invalid randomized FROST threshold Schnorr signatures that are produced by deterministically invalidating the share of a fixed player, and asserts that the resulting signature fails verification. Finally, it checks compliance with the draft specification by asserting that test vectors produce expected valid signatures.

While these are useful tests, they fail to capture failure modes of interest, and additionally fail to exercise the broader generality of the design beyond that relating to specification compliance.

There are several failure modes that might arise, and many important checks throughout the implementation to assert validity and other security properties. Currrently, the only failure mode testing is to invalidate a single signature share; while this is done on tests using randomized keys, the share invalidation method (both the player index and method of malleating an originally-valid share) uses fixed values. The test is therefore fairly limited.

Part of the generalization allows for the use of key offsets intended for re-randomizing signatures. Because this is not directly supported as part of the specification, it is untested as part of the library. However, it is used elsewhere in the repository, which is out of scope.

Further, the implementation supports the use of more complex nonce commitment constructions with common discrete logarithms, and even multiple sets

[20] `https://github.com/ZcashFoundation/zips/blob/8836e22610c4575b7c46f8a31e3819de1ac7efbf/zip-0312.rst#re-randomizable-frost`

[21] `https://zfnd.org/frost-performance/`

of these. This functionality involves the careful use of additional binding factors and discrete logarithm equality proofs, but is not tested as FROST Schnorr signatures do not use these features.

Recommended fix is to add tests that exercise failure modes more flexibly, and that exercise the more generalized functionality that the implementation provides.

Action: Addressed in commit `2fd5cd8`.

3.7 `crypto/multiexp`

This crate contains implementations of Straus- and Pippenger-type multiscalar multiplication algorithms. In addition, it provides a generic batching system that smartly handles the use case of asserting that each of a set of group element linear combinations evaluates to zero, and optionally identifying a failing linear combination within a failing set.

3.7.1 Code duplication

The implementation applies a random weight to each linear combination when added to a batch queue, and then flattens this data when evaluating the resulting multiscalar multiplication. However, it does so inconsistently and in several places, in part to handle zeroization differently for constant- and variable-time use cases. For example, constant-time verification performs the flattening in the `verify_core` function (which is called by the `verify` wrapper), variable-time verifications performs it directly in the `verify_vartime` function, and blaming performs it in the `blame_vartime` function. It may be simpler and more clear to refactor these functions to reduce duplication and risk of error.

A recommended action, if such simplification is desired, is to perform such a refactoring.

Action: Addressed in commit `15d6be1`.

3.7.2 Denial-of-service risk in blame

The implementation offers verification modes that, on failure of a batch to evaluate to zero, performs a binary search to identify a failing linear combination within the batch. The binary search always performs a split at the center of the underlying data vector as it performs its evaluations. Because of this, it may be possible to maliciously order input data within a batch in order to maximize the computation effort and time required for blame identification. This is likely not a practical concern due to the logarithmic complexity of binary search, and the risk of this occurring depends entirely on how the caller arranges its batch data.

Recommended action is to either shuffle the input data prior to the blame operation, or select the binary search split points randomly. Note that in the latter action, it is not necessary to use a cryptographically-secure random num-

ber generator; any reasonable pseudorandom number generator would suffice and likely yield improved performance.

Action: Not addressed. This ensures that blame is deterministic, and takes advantage of the complexity scaling as a mitigation.

3.7.3 Incomplete tests

Tests are incomplete. While benchmarking code for the Straus and Pippenger algorithms exists, there are no specific test functions that directly check both algorithms separately; instead, correctness tests for both algorithms use variable input sizes to trigger different algorithm selections. Further, there are no checks (via benchmarks or tests) against edge cases for either algorithm. A more robust approach would be to test both algorithms and the selection algorithm directly.

Further, testing of batch verification is offloaded to other crates that use this functionality.

Recommended action is to add these tests.

Action: Addressed in commit `8661111`.

3.7.4 Variable-time input data is assumed not to be secret

The implementation generally assumes that variable-time operations are not secret, and does not internally zeroize input data. While it is often the case that higher-level protocol implementers will use constant-time operations for secret input data and variable-time operations for public input data, this may not always be the case. In particular, it may be the case that a protocol implementer does not deem the risks of operation time leakage and in-memory copies of secrets to be linked in any particular way.

Recommended actions are either to unify the treatment of input data with respect to zeroization, or carefully document the existing behavior to reduce risk.

Action: Addressed in commit `15d6be1`.

3.7.5 Incomplete documentation

Several functions in the implementation lack documentation.

Recommended fix is to add documentation.

Action: Addressed in commit `e5329b4`.

3.7.6 Inefficiency in constant-time algorithms

When using variable-time multiscalar multiplication evaluation in both the Straus and Pippenger algorithms, the implementation performs an index check that avoids an unnecessary doubling; however, the corresponding constant-time functions do not perform it. Because this does not rely on input data, it does not affect constant-time security.

Recommended fix is to add the check to the constant-time functions for both algorithms.

Action: Addressed in commit `1d2ebdc`.

3.7.7 Inefficiency in variable-time Pippenger algorithm

When using variable-time multiscalar multiplication evaluation in the Pippenger algorithm, the implementation iterates over buckets of group elements and adds their contents to an accumulator. In the case where the underlying curve group implementation uses constant-time addition, this means that inclusion of an empty bucket is done inefficiently.

Recommended fix is to add a check for empty buckets to short-circuit their addition to the accumulator in the variable-time Pippenger evaluation function.

Action: Addressed in commit `1d2ebdc`.

3.8 `crypto/schnorr`

This crate provides generic functionality for Schnorr-type signatures, with the intent of compatibility with RFC 8032 `Ed25519` signatures. Signatures are produced in a manner that is agnostic to the challenge (and, therefore, challenge hashing algorithm) used, and to the underlying elliptic curve group. Batch verification of signatures is supported, and there is an implementation of half-aggregated signatures[22].

Work on `Ed25519` signature verification[23] notes undesired flexibility for signature verification, such that it is possible for different implementations to maintain compatibility with RFC 8032 but have functionally different verification criteria[24]. For example, it is possible to provide non-canonical point encodings as part of a signature whose validity is subsequently undefined. At a minimum, a compliant implementation should pass the appropriate test vectors provided in RFC 8032.

3.8.1 Incomplete cross-compatibility

Because of the poor verification criteria in RFC 8032, `Ed25519` (and related) implementations differ significantly in handling of certain edge cases. While this crate's implementation likely maintains strict compatibility with RFC 8032 in the sense that it passes randomized tests, it can reject signatures provided by other implementations.

In the case of the crate's `ed25519` group dependency provided elsewhere in the repository, verification keys and signature R group elements are checked to assert they have no torsion component and are therefore elements of the curve group's prime-order subgroup. As this is not a specific requirement of RFC 8032, it is possible to present a verification key or signature failing this assertion, which results in signature verification failure; other implementations,

[22] `https://eprint.iacr.org/2021/350`

[23] `https://github.com/penumbra-zone/ed25519-consensus`

[24] `https://hdevalence.ca/blog/2020-10-04-its-25519am`

like those following the ZIP 215[25] specification, would not reject signatures for this reason.

If more strict compatibility with other particular implementations is desired, a recommended fix is to make changes to verification criteria to account for this.

Action: Addressed in commit `35043d2`.

3.8.2 Incomplete tests

The crate does not include RFC 8032 test vectors, which should be specifically checked for strict RFC 8032 compatibility.

Recommended fix is to add tests using such test vectors.

Action: Addressed in commit `08f9287`.

3.8.3 Risk of unsafe challenges

The implementation is intentionally agnostic to challenge generation, and assumes that the caller computes the appropriate challenge for a verification key and signature passed to the signer and verifier. This extends to aggregated signatures as well. Challenge agnosticism introduces nontrivial risk, as failure to properly compute challenges in Fiat-Shamir constructions occurs in practice[26]. Details of challenge computation depend on specific signature instantiations (some of which themselves may imply risk).

Recommended fix is to carefully document this risk.

Action: Addressed in commit `053f07a`.

3.8.4 Incorrect half-aggregated signature implementation

Half-aggregated signatures are implemented. While the source preprint[27] for the technique is academic in nature and not a precise specification, it does not match the implementation architecture.

While the security analysis of the half-aggregation method described in Algorithm 7 of the preprint assumes that aggregation coefficients are produced using all verification keys, nonces, and messages for the signatures being aggregated as hash inputs to the H_1 hash function, the preprint later notes a possible optimization. In this optimization, the existing H_0 outputs from each individual signature are used as H_1 inputs instead. However, the implementation modifies this by passing to H_1 both the verification key and (presumably H_0-derived) challenge from each signature. Because the correctness of individual signatures challenges is (as noted elsewhere) not enforced or checked, the correctness of aggregation H_1-based challenges cannot be asserted.

We further note that the preprint does not appear to discuss the specific implications, if any, of the optimization on the security arguments, which relies

[25] `https://zips.z.cash/zip-0215`

[26] `https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/`

[27] `https://eprint.iacr.org/2021/350`

on a forking lemma structure. While we do not see any particular risks incurred from the optimization, there may be a theoretical loss of reduction tightness.

Although the inclusion of verification keys in the optimized aggregation coefficient computation does not match the construction in the preprint, it does not pose any security issues of concern, and it is not strictly necessary to issue a recommended fix. However, the verification keys may be removed if there is sufficient confidence in the correctness of the individual signature challenges passed to the verifier.

Action: Addressed in commit `8b7e7b1`.

3.8.5 Possible non-uniform sampling of aggregated signature hash functions

As noted, the overall aggregated signature design requires cryptographic hash functions H_0 and H_1 modeling random oracles: H_0 is used to compute individual signature challenges, and H_1 is used to compute aggregation coefficients.

In order to properly model distinct random oracles, it is necessary that H_0 and H_1 be sampled uniformly. We note that this is not specifically indicated by the source preprint, but appears to be implied and required by the security arguments. A standard approach in protocol implementations to ensure uniform hash function sampling is to use careful domain separation of a single cryptographic hash function.

As already noted, individual signature challenges (which in the preprint are constructed using H_0) are not controlled or checked by the implementation in order to be agnostic about specific design choices. However, the aggregation coefficients (which in the preprint are constructed using H_1) in the implementation are directly constructed using domain separation of a generic hash function selected by the caller.

This means that it is possible for the caller to arrange its H_0 challenge and H_1 hash function instantiations such that the required uniform sampling is not maintained. We note that provided the H_0 challenges are sanely computed with non-colliding domain separation, and provided a modern cryptographic hash function is selected for H_1 aggregation coefficients, it is very unlikely that exploitable non-uniform sampling could arise in practice.

One possible mitigation is to offload H_1 hash function domain separation to the caller. While this does introduce some risk by requiring the caller to perform the domain separation safely and correctly, it provides flexibility that may make it easier for the caller to ensure H_0 and H_1 are effectively sampled uniformly as required.

Action: Addressed in commit `5306717`.

3.8.6 Unclear aggregation coefficient security target

The implementation documentation indicates that to target a 128-bit security level, it is sufficient to use a 128-bit hash function for use in computing ag-

gregation coefficients. Interestingly, the citation[28] given for this conclusion is a preprint discussing batch verification, not half-aggregated signatures of the type used in the implementation, and it is unclear why its conclusions apply in this circumstance. The analysis in the design source preprint is somewhat unclear on the nature of the relationship between the security target and aggregation hash function output size, as it presents several different constructions with different reduction tightness. From the analysis given, it appears that the construction used in the implementation still requires 256-bit hash function outputs in order to reach a 128-bit security target. However, the preprint notes that similar Schnorr reductions do not yield particular exploitable weaknesses in other analyses.

Recommended fix is to require and use a hash function with 256-bit output for aggregation coefficients. While this incurs a performance reduction for a variable-time verifier, we note that the implementation may be modified to support batch verification, which can somewhat offset this penalty.

Action: Addressed in commit `97374a3`.

3.9 `crypto/transcript`

This crate provides a generic `Transcript` trait intended to be used in Fiat-Shamir applications, where data is added to a domain-separated transcript, and challenges or RNG seeds may be sampled. It then provides a generic `DigestTranscript` struct that implements `Transcript` and can take advantage of any cryptographic hash function (enforced using a custom `SecureDigest` trait bound). It also provides `MerlinTranscript`, a `Transcript` implementation that uses the Merlin transcripting system[29]. Finally, it provides an instantiation `RecommendedTranscript` of `DigestTranscript` that uses 512-bit `Blake2b` as its hash function.

3.9.1 Malfunctioning digest trait bound

The `SecureDigest` trait includes a bound that the underlying digest output size be at least 256 bytes. This is an error, as the 128-bit security target implies an output size of 256 *bits*, or 32 bytes. The associated docstring also contains this error in its description.

However, there is an associated error with unknown cause. Namely, the `RecommendedTranscript` type alias, which implements `SecureDigest`, uses the `Blake2b` hash function with 512-bit output. As this is only 64 bytes, it should fail the incorrect `SecureDigest` trait bound, but does not.

This is not a security issue, since `RecommendedTranscript` does meet the security target with a safe cryptographic hash function; however, it suggests an upstream implementation error.

[28] `https://cr.yp.to/badbatch/badbatch-20120919.pdf`
[29] `https://merlin.cool/`

Recommended fixes are to correct the docstring and trait bound, and to identify and account for (to the extent possible) any upstream issue that results in the digest output size mismatch.

Action: Addressed in commits `2f4f1de` and `7efedb9`.

3.9.2 Possibly confusing randomization seed function

The `Transcript` trait includes separate functionality for challenge and random number generator seed generation. While these are distinct functions, they use the same underlying `DigestTranscriptMember` implementation for inclusion into the transcript. This means that generation of a challenge or seed with the same label from the same transcript state can (depending on instantiation) result in identical data. It is possible that the user may not expect this behavior, which could result in an unintended collision.

Recommended fixes are either a docstring improvement, or the addition of a separate `DigestTranscriptMember` element for random number generator seeds.

Action: Addressed in commit `79124b9`.

3.9.3 Possible domain separation conflict in Merlin wrapper

The `Transcript` trait includes functionality for transcript domain separation, which can be used for protocol composition. In the `DigestTranscript` struct, the domain separator is distinguished from other transcript elements via the use of a separate `DigestTranscriptMember` to prevent collisions.

The Merlin transcript wrapper fails to explicitly enforce this differentiation, and relies on the Merlin library's message appending functionality. Domain separation is implemented by adding a message to the underlying Merlin transcript with a label `dom-sep`. This means it is possible to induce a transcript collision if the user adds a transcript message with this label.

A recommended fix is to document this behavior to avoid collisions.

Action: Addressed in commit `20a3307`.

3.9.4 No tests exist

This crate contains no tests.

Recommended fix is to add tests.

Action: Addressed in commit `a053454`.

3.10 `crypto-tweaks`

Commits to this repository branch include minor changes like adding debugging trait implementations, updating dependencies, improving constant-time operations, and refactoring for naming clarity. A more substantive update separates Schnorr signature functionality based on transcript type, such that

`IetfTranscript` is handled separately from generic transcripts. Another significant change is to modify the handling of `DigestTranscript` challenges to be safer when used with hash functions susceptible to length extension.

3.10.1 Incomplete handling of intended constant-time operations

The implementation uses Rust's `black_box` functionality to mitigate the risk of variable-time operations relating to boolean conversions to `u8` prior to the use of `Choice`, which accepts only a `u8` value. Further, it aims to zeroize values during this process that may be secret. However, it misses several instances of this behavior.

Recommended fix is to expand the use of `black_box` and zeroization to all such conversions.

Action: Addressed in commit `ad470bc`.

3.10.2 Indirect transcript testing

The implementation includes tests of two `Transcript` instantiations: Merlin transcripts defined via `MerlinTranscript`, and the `Blake2b`-based transcript used in the `RecommendedTranscript` type alias for a `DigestTranscript` instantiation.

However, the test does not directly use the `RecommendedTranscript` type alias, instead using a direct `DigestTranscript` instantiation that happens to match the type alias. As a result, any change to this type alias will not be automatically reflected in testing.

Recommended fix is to use `RecommendedTranscript` directly in testing.

Action: Addressed in commit `669d2db`, which also tests `DigestTranscript` with `SHA-256`, a hash function susceptible to length extension.