Zcash FROST Security Assessment

Zcash Foundation
Version 1.1 – October 19, 2023

Prepared By
Paul Bottinelli
Thomas Pornin
Eli Sohl

Prepared For
Deirdre Connolly
Natalie Eskinazi
Jack Gavigan
Conrado Gouvea
Maria Pilar Guerra-Arias
Chelsea Komlo

1 Executive Summary

Synopsis

In Summer 2023, the Zcash Foundation engaged NCC Group to conduct a security assessment of the Foundation's FROST threshold signature implementation, based on the paper *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*[1]. This project implements v12 of the draft FROST specification[2] in Rust, with a variety of options available for underlying elliptic curve groups. The review was performed by three consultants over 25 person-days of effort, including a retest phase performed a few weeks after the original engagement.

Scope

The scope targeted the project's v0.6.0 release (corresponding to commit `5fa17ed`), and covered the project's main crates:

- `frost-core`
- `frost-ed25519`
- `frost-ed448`
- `frost-p256`
- `frost-secp256k1`
- `frost-ristretto255`

as well as the dependency `reddsa` at tagged version 0.5.1. Parts of `Ed448-Goldilocks` were also in scope, particularly those components used by `frost-ed448`.

Limitations

No noteworthy limitations were encountered during this project. It is noted that this engagement focused on reviewing the given FROST implementation and matching it to the reference implementation and paper, rather than on reviewing these source materials themselves.

Key Findings

No critical or high-severity findings were identified. A number of Medium, Low, and Informational findings were reported; among these, the following are highlighted:

- Finding "Insufficient Participant Commitment List Checks", in which a malicious adversary may perform elaborate attacks against participants, including denial-of-service attacks and potential forgeries by crafting malicious Signing Packages that are undetected by other participants.
- Finding "Missing Length Check in Identifiers List", where the failure to ensure that a custom list of identifiers is consistent with the threshold parameters of the scheme may facilitate denial-of-service attacks and result in a potential loss of security provided by the threshold assumption.
- Finding "Ed448 Base Field Incorrect Negation", where the Ed448 implementation voids the security guarantees of the formal verification of the fiat-crypto primitives through the misuse of the fiat-crypto API.

This report also includes an Engagement Notes section, a semi-structured collection of observations that did not warrant findings, but that may be of independent interest to the Zcash team. Additionally, a FROST Security Requirements section collecting requirements from the latest FROST draft specification was developed during the course of the engagement.

The project concluded with a retest phase that confirmed *all* findings were fixed. Additionally, the Zcash team diligently addressed all but two of the observations in the Engagement Notes section.

1. https://eprint.iacr.org/2020/852
2. https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-12.html

Strategic Recommendations

Overall, the project is well implemented and the code contains extensive comments, making navigation and understanding of these complex cryptographic primitives easier. NCC Group encourages the Zcash Foundation team to maintain this standard of quality as the library matures.

A number of findings discuss issues related to the lack of input validation, particularly in functions accepting potentially untrusted input. Consider performing a pass over the code base and adding parameter validation checks where appropriate, prioritizing functions that are exposed externally. Consider using the section FROST Security Requirements as a companion reference to ensure the necessary checks are in place.

The existing “FROST Book” could be greatly expanded. For instance, it could be augmented with usage examples, description of library structure, discussion of how concepts from the FROST paper and draft specification map onto the implementation, etc. In particular, well-commented usage examples covering the library’s full range of features would significantly reduce the likelihood of API misuse by end users.

Some of this material already exists - for instance, some usage examples are provided within individual backend crates - but could be better publicized and collected for easy discoverability, and could be expanded to cover newer library features such as DKG.

2 Dashboard

Target Data

Name	Zcash FROST
Type	Cryptographic Library
Platforms	Rust

Engagement Data

Type	Cryptography and Implementation Review
Method	Source Code Review
Dates	2023-07-05 to 2023-07-26
Consultants	3
Level of Effort	25 person-days

Targets

`frost-core v0.6.0`	A generic implementation of FROST in Rust
`frost-ed25519 v0.6.0`	A backend for `frost-core` adding support for Ed25519
`frost-ed448 v0.6.0`	A backend for `frost-core` adding support for Ed448
`frost-p256 v0.6.0`	A backend for `frost-core` adding support for P-256
`frost-ristretto255 v0.6.0`	A backend for `frost-core` adding support for Ristretto255
`frost-secp256k1 v0.6.0`	A backend for `frost-core` adding support for Secp256k1
`reddsa v0.5.1`	An implementation of RedDSA used by `frost-core`
`Ed448-Goldilocks`	An implementation of `Ed448-Goldilocks` used by `frost-448`

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	3
Low issues	3
Informational issues	2
Total issues	**8**

Category Breakdown

Cryptography	3
Data Validation	4
Denial of Service	1

Component Breakdown

Ed448-Goldilocks	3
frost-core	5

Critical High Medium Low Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Ed448 Base Field Incorrect Negation	Fixed	72Y	Medium
Insufficient Participant Commitment List Checks	Fixed	AW3	Medium
Missing Length Check in Identifiers List	Fixed	9XW	Medium
Potential Timing Attacks in Ed448 Implementation	Fixed	T3P	Low
Unchecked Accesses to Data Structures	Fixed	4VP	Low
Missing Signing Package Validation May Cause a Panic	Fixed	2WM	Low
Lack of Zeroization in Ed448 Scalar Inversion	Fixed	HGL	Info
Minimum Participant Constraint Enforcement Improvements	Fixed	XLV	Info

4 Finding Details

Medium

Ed448 Base Field Incorrect Negation

Overall Risk	Medium	**Finding ID**	NCC-E008263-72Y
Impact	High	**Component**	Ed448-Goldilocks
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Through misuse of the fiat-crypto API, the Ed448 implementation voids the security guarantees of the formal verification of the fiat-crypto primitives. An actual miscomputation in the context of FROST does not seem possible, though.

Description

The Ed448-Goldilocks crate, used by the Zcash implementation of the FROST(Ed448, SHAKE256) ciphersuite, relies itself on two possible backends for the implementation of operations over the curve base field (integers modulo $q = 2^{448} - 2^{224} - 1$). The `u32` backend is intended for 32-bit architectures, whereas 64-bit architectures should use the `fiat_u64` backend (selected by default), which is a wrapper around the fiat-crypto implementation of computations in specific finite fields. Fiat-crypto consists of automatically generated routines, using a methodology that also outputs mathematical proofs of correctness of the result for all possible inputs. Use of fiat-crypto code is a great step toward ensuring that the implementation operates properly, even on maliciously crafted input; however, this applies only if the fiat-crypto routines are used appropriately.

Within the fiat-crypto implementation of operations modulo *q* on a 64-bit architecture (using the `p448_solinas_64` module), values can use two internal representations. Both split the integer over eight limbs, in base 2^{56}, but they differ on the allowed ranges for the limb values. In *recent* fiat-crypto versions (e.g. version 0.1.20), the two representations have distinct Rust type names:

- `fiat_p448_tight_field_element`: limb values must be between 0 and 2^{56} (inclusive).
- `fiat_p448_loose_field_element`: limb values must be between 0 and 3×2^{56} (inclusive).

“Tight” values can be trivially converted into “loose” values (since the allowed limb range of the latter includes that of the former), but transforming a “loose” representation into a “tight” representation of the same value requires some carry propagation, which is done by the `fiat_p448_carry()` function. Each implementation of arithmetic primitives is typed, e.g. addition (`fiat_p448_add()`) expects “tight” inputs, but produces a “loose” output. The formal verification of the implementation is guaranteed only as long as only “tight” values are provided as parameters to functions that expect such “tight” values.

Links above are to the most recent fiat-crypto crate version at the time of writing, which is 0.1.20. The `Ed448-Goldilocks` crate uses an older version (0.1.4). In that older version, the same arithmetic routines are used, but there are no separate type aliases for tight and loose values; instead, both use the generic `[u64; 8]` type. In any case, even in version 0.1.20, the two types are really type *aliases* on `[u64; 8]`, and are interchangeable with each other; thus, it is up to the caller to ensure that loose values are reduced into tight values where necessary. The lack of really distinct Rust types means that if a necessary reduction is omitted, this will not be detected by the compiler through type analysis.

Such a reduction step is missing in the implementation of the `FieldElement56::negate()` function. The `FieldElement56` type, defined in the Ed448-Goldilocks crate, is a simple wrapper around an array of eight 64-bit limbs:

```
#[derive(Copy, Clone, Debug)]
pub struct FieldElement56(pub(crate) [u64; 8]);
```

The actual contents are not documented, but the internal array is passed as-is to the fiat-crypto functions that expect tight values, such as `fiat_p448_add()`; we must therefore assume that `FieldElement56` should contain only tight values. This is consistent with how, for instance, addition on `FieldElement56` values is implemented:

```
impl Add<FieldElement56> for FieldElement56 {
    type Output = FieldElement56;
    fn add(self, rhs: FieldElement56) -> Self::Output {
        let mut inter_res = self.add_no_reduce(&rhs);
        inter_res.strong_reduce();
        inter_res
    }
}
```

`add_no_reduce()` and `strong_reduce()` wrap around `fiat_p448_add()` and `fiat_p448_carry()`, respectively: the former outputs a loose value, which the latter reduces into a tight value; the `inter_res` variable transiently contains a loose value, but upon exiting the `add()` function, it has been normalized to a tight value.

The implementation of negation does not include the normalization step:

```
    /// Negates a field element
    pub(crate) fn negate(&self) -> FieldElement56 {
        let mut result = FieldElement56::zero();
        fiat_p448_opp(&mut result.0, &self.0);
        result
    }
```

The `fiat_p448_opp()` function outputs loose values (with limbs up to 2^{57} - 2). If a negated `FieldElement56` value is used in other arithmetic operations, then this will imply using a loose representation of a field element with fiat-crypto functions that expect tight representations, thereby voiding the security guarantees offered by the formal verification of the fiat-crypto code.

In practice, the following code demonstrates how that issue can lead to an incorrect output:

```
    #[test]
    fn test_negate() {
        let x = FieldElement56::zero();
        let y = x.negate();
        assert_eq!(y.to_bytes(), [0u8; 56]);
    }
```

The negation of zero should still be zero, and its only valid (canonical) encoding is a sequence of 56 bytes of value 0x00. The `test_negate()` function should execute successfully, per the API of `FieldElement56`, but in practice it fails. In the test code above, `y.to_bytes()` does not yield an all-zero output, but instead the little-endian encoding of q (the field modulus). Internally, `negate()` returns the integer $2q$, and `to_bytes()` performs a *single* conditional subtraction of q (to attempt to normalize the value into the 0 to q-1 range), hence the obtained result.

Manual analysis of the fiat-crypto routines indicates that they in fact support a larger range of limb values on input, especially when used only through the `FieldElement56` wrappers, which enforce reduction to a tight representation after each addition and subtraction. It appears that the *only* case of an incorrect output is the one demonstrated here: negation of (exactly) the value zero, and subsequent encoding of that value into bytes, with no arithmetic operation on the value between negation and encoding. This situation cannot be reached through the external API of the Ed448-Goldilocks crate: negation of field elements can happen by calling the `negate()` or `torque()` functions on an `ExtendedPoint`, but encoding into bytes happens only from the `ExtendedPoint::compress()` function, and is preceded by conversion to affine coordinates, which involves multiplication of field elements by the inverse of the internal `Z` coordinate. Multiplication always produces a properly reduced (tight) representation.

The issue presented here is therefore not immediately exploitable. It still implies the loss of the formal verification guarantees, and thus should be fixed.

Recommendation

A call to `fiat_p448_carry()` should immediately follow the call to `fiat_p448_opp()`, to ensure proper reduction.

Location

Ed448-Goldilocks/src/field/fiat_u64/prime_field.rs, lines 164-168

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 29 of the Ed448-Goldilocks crate, and observed that a call to `fiat_p448_carry()` had been introduced following the call to `fiat_p448_opp()`. This is aligned with the recommendation above. As such, this finding has been marked “Fixed”.

Medium

Insufficient Participant Commitment List Checks

Overall Risk	Medium	**Finding ID**	NCC-E008263-AW3
Impact	Medium	**Component**	frost-core
Exploitability	Medium	**Category**	Data Validation
		Status	Fixed

Impact

A malicious adversary may perform elaborate attacks against participants, including denial-of-service attacks and potential forgeries by crafting malicious Signing Packages that are undetected by other participants.

Description

The FROST signature process is split into two rounds: in a first round, participants generate nonces and their corresponding public commitment values (which are sent to the Coordinator, who collates them with a Message in a Signing Package); in a second round, participants “sign” the Message by computing their respective *signature shares* on the Signing Package.

In the implementation, this Signing Package is represented by the structure `SigningPackage` defined in *frost-core/src/frost.rs*, and excerpted below (with some annotations left out for ease of presentation).

```
pub struct SigningPackage<C: Ciphersuite> {
    /// The set of commitments participants published in the first round of the
    /// protocol.
    pub signing_commitments: BTreeMap<Identifier<C>, round1::SigningCommitments<C>>,
    // ...
    /// Message which each participant will sign.
    ///
    /// Each signer should perform protocol-specific verification on the
    /// message.
    message: Vec<u8>,
    // ...
    /// Ciphersuite ID for serialization
    ciphersuite: (),
}
```

The field `signing_commitments` highlighted in the code excerpt above is the data structure mapping the nonce commitments generated by participants during round 1 to their respective identifiers. This list of commitments is used to compute the group commitment during the signature share generation process performed by the `sign()` function, located in the file *frost-core/src/frost/round2.rs*, as can be seen in the highlighted in the excerpt of the function below.

```
pub fn sign<C: Ciphersuite>(
    signing_package: &SigningPackage<C>,
    signer_nonces: &round1::SigningNonces<C>,
    key_package: &frost::keys::KeyPackage<C>,
) -> Result<SignatureShare<C>, Error<C>> {
```

```
    // Encodes the signing commitment list produced in round one as part of generating [`BindingFactor`], the
    // binding factor.
    let binding_factor_list: BindingFactorList<C> =
        compute_binding_factor_list(signing_package, &key_package.group_public, &[]);
    let binding_factor: frost::BindingFactor<C> =
        binding_factor_list[key_package.identifier].clone();

    // Compute the group commitment from signing commitments produced in round one.
    let group_commitment = compute_group_commitment(signing_package, &binding_factor_list)?;

    // Compute Lagrange coefficient.
    let lambda_i = frost::derive_interpolating_value(key_package.identifier(), signing_package)?;

    // Compute the per-message challenge.
    let challenge = challenge::<C>(
        &group_commitment.0,
        &key_package.group_public.element,
        signing_package.message.as_slice(),
    );

    // Compute the Schnorr signature share.
    let signature_share = compute_signature_share(
        signer_nonces,
        binding_factor,
        lambda_i,
        key_package,
        challenge,
    );

    Ok(signature_share)
}
```

In the implementation above, the `signing_commitments` field (a member of the `signing_package` structure passed as a parameter to the `sign()` function) is never checked to be valid and consistent with the participant's view. Finding "Missing Signing Package Validation May Cause a Panic" discusses a potential panic that may occur when a participant's identifier is missing. However, this finding highlights that participants also do not ensure that the commitments associated to their identifiers are the ones they initially sent, nor that the list does not contain unexpected entries, such as duplicate values. The function also does not ensure that the number of participants tracked in the `signing_commitments` list is consistent with the minimum and maximum number of signers specified for this signing round.

This contravenes the FROST specification, which, under Section 5.2. Round Two - Signature Share Generation, states that:

> each participant MUST ensure that its identifier and commitments (from the first round) appear in commitment_list.

In practice, an adversary may be able to perform a number of attacks on participants. A straightforward attack against a target participant consists in the tampering of that participant's commitments, which will go undetected until the aggregation phase, at which point the signature verification process will fail and that participant will be identified as the culprit. However, more complex attacks might be performed by adversaries with larger consequences. The original FROST paper[3] describes, under *Section 2.5 – Attacks on Parallelized Schnorr Multisignatures*, some attacks that can be leveraged by adversaries with control of the commitments, such as a signature forgery using a ROS Solver[4]. Such an attack could potentially be carried out here, given that an attacker would essentially have entire control over the commitment values. Additionally, an adversary also has significant freedom over the inputs to the function `compute_signature_share()` (called on line 211 of the excerpt above) which involves the participant's long-term private key, and selectively providing certain inputs could potentially lead to *some* leak of private information, for example via side-channel attacks. These attacks were not investigated in more depth due to the time-boxed nature of the engagement.

Recommendation

Add checks to validate that the `signing_commitments` field contains the participant's identifier and that the commitments listed for that identifier correspond to the commitments sent to the Coordinator during phase 1. Additionally, for the purpose of defence in depth, consider whether some additional checks could be performed to provide assurance of the validity of the commitments of other participants. For example, if the signer had access to the number of participants or the expected threshold, they could check whether the length of the map is consistent with the known participant number.

Location

frost-core/src/frost/round2.rs

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 480, and observed that a new `min_signers` field had been added to the `KeyPackage` structure, which is now used to check that the number of Signing Commitments in a Signing Package is sufficient prior to a signing operation (see updates to the file *frost-core/src/frost/round2.rs*). Together with the changes introduced in pull request 452 to address finding "Missing Signing Package Validation May Cause a Panic", this finding is now appropriately mitigated and has been marked "Fixed" as a result.

Client Response

1. Several of the defense in depth recommendations can easily be circumvented by an adversary. For example, checking if the set of commitments is equal to the assumed number of signers can easily be circumvented by an adversary that adds random group elements to the set of commitments. As such, the performance overhead of performing these checks do not seem to outweigh the benefits.
2. It is unclear to us how the participant's long-lived secret key could leak even if the adversary had complete control over the inputs to determine the binding factor and the challenge. It is clear that ROS attacks are viable if the participant does not ensure that their commitments are represented in the commitment set.
3. The function `generate_secret_shares` is assumed to be performed by a trusted dealer. If the dealer is not trusted, then all security is lost. If the dealer is untrusted, then a DKG should be used, to generate key material in such a way that no single entity is trusted.

3. https://eprint.iacr.org/2020/852.pdf
4. https://eprint.iacr.org/2020/945.pdf

Medium

Missing Length Check in Identifiers List

Overall Risk	Medium	**Finding ID**	NCC-E008263-9XW
Impact	High	**Component**	frost-core
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Failure to ensure that a custom list of identifiers is consistent with the threshold parameters of the scheme may facilitate denial-of-service attacks and result in a potential loss of security provided by the threshold assumption.

Description

A recent commit in the FROST repository under review introduced support for deriving identifiers from arbitrary strings, in order to create participant identifiers from personal data such as email addresses.

The function `split()` in *frost-core/src/frost/keys.rs* is the entry point for a Dealer to split an existing private signing key into FROST shares to be distributed to the various participants. The relevant arguments of that function are two unsigned 16-bit integer values representing the maximum and the minimum number of signers in order to generate the secret polynomial, as well as a list of identifiers. This identifiers list can either be of type `Default`, in which case default identifier values will be assigned to participants (namely “1 to max_signers, inclusive”), or of type `Custom`, which represents a “user-provided list of identifiers” (see definition of the `IdentifierList` enum in *keys.rs*).

Presumably, the size of the provided identifiers list should be consistent with the `max_signers` and `min_signers` parameters. However, these bounds on the identifiers list size are not enforced within the code base. The excerpt of the `split()` function below shows how the execution proceeds to generate the secret shares without ever ensuring the consistency of the identifiers list with the `max_signers` and `min_signers` parameters.

```
pub fn split<C: Ciphersuite, R: RngCore + CryptoRng>(
    key: &SigningKey<C>,
    max_signers: u16,
    min_signers: u16,
    identifiers: IdentifierList<C>,
    rng: &mut R,
) -> Result<(HashMap<Identifier<C>, SecretShare<C>>, PublicKeyPackage<C>), Error<C>> {
    let group_public = VerifyingKey::from(key);

    let coefficients = generate_coefficients::<C, R>(min_signers as usize - 1, rng);

    let default_identifiers = default_identifiers(max_signers);
    let identifiers = match identifiers {
        IdentifierList::Custom(identifiers) => identifiers,
        IdentifierList::Default => &default_identifiers,
    };

    let secret_shares =
        generate_secret_shares(key, max_signers, min_signers, coefficients, identifiers)?;
```

Highlighted in the `split()` function above, the execution then proceeds into the function `generate_secret_shares()` in *frost-core/src/frost/keys.rs*, which is provided below, for reference.

```rust
pub(crate) fn generate_secret_shares<C: Ciphersuite>(
    secret: &SigningKey<C>,
    max_signers: u16,
    min_signers: u16,
    coefficients: Vec<Scalar<C>>,
    identifiers: &[Identifier<C>],
) -> Result<Vec<SecretShare<C>>, Error<C>> {
    let mut secret_shares: Vec<SecretShare<C>> = Vec::with_capacity(max_signers as usize);

    let (coefficients, commitment) =
        generate_secret_polynomial(secret, max_signers, min_signers, coefficients)?;

    let identifiers_set: HashSet<_> = identifiers.iter().collect();
    if identifiers_set.len() != identifiers.len() {
        return Err(Error::DuplicatedIdentifier);
    }

    for id in identifiers {
        let value = evaluate_polynomial(*id, &coefficients);

        secret_shares.push(SecretShare {
            identifier: *id,
            value: SigningShare(value),
            commitment: commitment.clone(),
            ciphersuite: (),
        });
    }

    Ok(secret_shares)
}
```

The function above first generates the secret polynomial based on the `min_signers` and `max_signers` parameters. Then, the function iterates over *all* identifiers in the highlighted loop, evaluating the polynomial at that particular value and creating the corresponding secret share.

Since the size of the `identifiers` list can be smaller than `min_signers` or larger than `max_signers`, it can lead to a few potential issues:

- If the `identifiers` list is larger than `max_signers`, the Dealer will evaluate the secret polynomial more times than there are potential participants, which could result in unexpected private key disclosure – *if these extra shares were to be distributed*. Indeed, the reconstruction portion of the secret sharing scheme is based on polynomial interpolation, and computing additional shares damages the threshold property of the secret sharing scheme.
- An arbitrarily large `identifiers` list may also result in potential denial-of-service attacks, since providing a large list will take a long time to process (due to the numerous polynomial evaluations required) and will require large memory allocations.
- If the `identifiers` list is smaller than `min_signers`, the participants would not be able to reconstruct the secret, which also constitute a form of denial-of-service.

Recommendation

Add a check in the `split()` function (and possibly in the `generate_secret_shares()` function) to ensure that the size of the identifiers list is within the [`min_signers`, `max_signers`] range. It seems reasonable to expect the size of the identifiers list to be equal to `max_signers`, in which case it would be recommended to ensure strict equality.

Location

- *frost-core/src/frost/keys.rs*
- *frost-core/src/frost/keys.rs*

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 481, and observed that the `split()` function in *frost-core/src/frost/keys.rs* now ensures that the length of the identifier list is equal to the maximum number of signers, and returns an error otherwise. This is aligned with the recommendation above. As such, this finding has been marked "Fixed".

Low

Potential Timing Attacks in Ed448 Implementation

Overall Risk	Low	**Finding ID**	NCC-E008263-T3P
Impact	High	**Component**	Ed448-Goldilocks
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

Non constant-time code in operations on Ed448 scalars and based field elements might be leveraged by attackers observing the timing characteristics of the execution of code using secret values, so as to obtain some information on these values.

Description

The Ed448 scalar field (integers modulo the subgroup 446-bit prime order p) is implemented by Ed448-Goldilocks with the custom `Scalar` type. Internally, values are represented over 14 limbs in base 2^{32} (always reduced to the canonical range 0 to p-1). Montgomery multiplication is used: for inputs x and y, Montgomery multiplication computes the integer $xy + kp$ for some non-negative integer k (lower than 2^{448}), such that the result is a multiple of 2^{448}. A simple shift can then divide that value by 2^{448}, thus yielding a result which is necessarily less than $2p$. By conditionally subtracting p (i.e. subtracting p, but adding it back if the subtraction makes the value negative), one obtains a properly reduced representation of xy/R modulo p, where $R = 2^{448}$.

The conditional subtraction is performed by the `sub_extra()` function:

```
fn sub_extra(a: &Scalar, b: &Scalar, carry: u32) -> Scalar {
    let mut result = Scalar::zero();

    // a - b
    let mut chain = 0i64;
    for i in 0..14 {
        chain += a[i] as i64 - b[i] as i64;
        // Low 32 bits are the results
        result[i] = chain as u32;
        // 33rd bit is the borrow
        chain >>= 32
    }

    // if the result of a-b was negative and carry was zero
    // then borrow will be 0xfff..fff and the modulus will be added conditionally to the result
    // If the carry was 1 and a-b was not negative, then the borrow will be 0x00000...001 ( this should not happen)
    // Since the borrow should never be more than 0, the carry should never be more than 1;
    // XXX: Explain why the case of borrow == 1 should never happen
    let borrow = chain + (carry as i64);
    assert!(borrow == -1 || borrow == 0);

    chain = 0i64;
    for i in 0..14 {
        chain += (result[i] as i64) + ((MODULUS[i] as i64) & borrow);
```

```
392         // Low 32 bits are the results
393         result[i] = chain as u32;
394         // 33rd bit is the carry
395         chain >>= 32;
396     }
397
398     result
399 }
```

On lines 371-379, *p* is subtracted from the value. The subtraction yields a final “borrow” value which can only be 0 (result is non-negative) or -1 (result is negative), since the input value was less than *2p*. Code on lines 389-396 adds back the modulus *p* if the borrow is -1 (i.e. the bit pattern 0xffffffffffffffff), but adds zero if the borrow is 0.

On line 387, an `assert!` clause verifies that `borrow` is indeed equal to 0 or -1. In Rust, such assertions are retained in release builds (conversely, the `debug_assert!` macro would make an assertion in debug builds only). The logical “or” operation (`||` operator) *may* be converted by the compiler into a conditional jump; in that case, the execution time and memory access pattern of the code would depend on whether `borrow` was 0 or -1 at that point (a jump misprediction typically induces a pipeline flush and a delay of a dozen clock cycles; loading of the instructions from memory may induce cache misses that can increase that delay to hundreds of cycles). Such variance is potentially detectable by attackers who are in position of observing the timing behaviour of the implementation, e.g. if the code executes in a security enclave (such as Intel SGX) or if the attacker can control a virtual machine co-hosted on the same hardware as the target system. Each information leak can thus be about one bit of information on the involved scalar values. In particular, the secret signing shares of FROST members are used repeatedly in multiplications with other changing values, and a one-bit leak per protocol execution could lead to private share extraction in as little as a few hundreds of observations.

Another similar leak is in the 32-bit backend for operations on the base field of curve Ed448 (in *Ed448-Goldilocks/src/field/u32/prime_field.rs*, function `strong_reduce()`, line 331):

```
324         // There are two cases to consider; either the value was >= p or it was <less than
            ↳ p
325         // Case 1:
326         // If the value was more than p, then the final borrow will be zero. This is
            ↳ scarry.
327         // Case 2:
328         // If the  value was less than p, the final borrow will be -1.
329
330         // The only two possibilities for the borrow bit is -1 or 0.
331         assert!(scarry == 0 || scarry + 1 == 0);
```

In this case, the `strong_reduce()` function is called only when encoding a field element into bytes, or when converting a curve point to affine coordinates. The borrow (`scarry`) will almost always be -1, because in that field implementation, values are “weakly reduced” and almost never exceed the modulus value. Moreover, the `u32` backend is used only when selecting it explicitly in the compilation process, presumably to better support 32-bit architectures. Thus, this leak is less likely to be a practical issue than the first one presented above.

Recommendation

The two `assert!` clauses should be either converted to a single constant-time test, or simply removed.

Location

- *Ed448-Goldilocks/src/field/scalar.rs*, line 387
- *Ed448-Goldilocks/src/field/u32/prime_field.rs*, line 331

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 31 of the Ed448-Goldilocks crate, and observed that the two offending `assert` calls had been removed, as suggested in the recommendation above. As such, this finding has been marked “Fixed”.

Low

Unchecked Accesses to Data Structures

Overall Risk	Low	**Finding ID**	NCC-E008263-4VP
Impact	Medium	**Component**	frost-core
Exploitability	Low	**Category**	Denial of Service
		Status	Fixed

Impact

Unchecked accesses to different data structures in the code base may lead to unhandled panics, eventually crashing the application.

Description

This finding lists a few instances where data structures are accessed at indices that may not exist, which would result in unhandled panics. These instances relate to the arguments of the `aggregate()` function, which does not ensure its three inputs represent a valid, consistent set of data.

Public Key Package

The structure `PublicKeyPackage` defined in *frost-core/src/frost/keys.rs* contains the public keys of all signers, as well as the group public key data.

```
pub struct PublicKeyPackage<C: Ciphersuite> {
    /// When performing signing, the coordinator must ensure that they have the
    /// correct view of participants' public keys to perform verification before
    /// publishing a signature. `signer_pubkeys` represents all signers for a
    /// signing operation.
    pub(crate) signer_pubkeys: HashMap<Identifier<C>, VerifyingShare<C>>,
    /// The joint public key for the entire group.
    pub(crate) group_public: VerifyingKey<C>,
```

The `signer_pubkeys` map is used during the aggregation process performed by the Coordinator in the `aggregate()` function, in *frost-core/src/frost.rs*. The signature of that function is provided below.

```
pub fn aggregate<C>(
    signing_package: &SigningPackage<C>,
    signature_shares: &HashMap<Identifier<C>, round2::SignatureShare<C>>,
    pubkeys: &keys::PublicKeyPackage<C>,
) -> Result<Signature<C>, Error<C>>
```

The `aggregate()` function accesses the `signer_pubkeys` member of the `pubkeys` parameter at an index coming from the `signature_shares` parameter, as can be seen in the excerpt below.

```
// Verify the signature shares.
for (signature_share_identifier, signature_share) in signature_shares {
    // Look up the public key for this signer, where `signer_pubkey` =
    ↪ _G.ScalarBaseMult(s[i])_,
    // and where s[i] is a secret share of the constant term of _f_, the secret polynomial.
    let signer_pubkey = pubkeys
        .signer_pubkeys
        .get(signature_share_identifier)
        .unwrap();
```

While it seems unlikely to occur in practice, there is a possibility that the `signature_share_identifier` is not contained in the `signer_pubkeys` map, leading to a panic due to the `unwrap()` call on line 424, which the code does not gracefully handle. The principle of defense in depth could be followed by checking that all the identifiers in the `signature_shares` are present in the `pubkeys`.

Signing Package

The structure `SigningPackage` defined in *frost-core/src/frost.rs* and excerpted below, keeps track of the commitments issued by the different participants during the first round of the signature generation protocol. This structure maintains a `BTreeMap`, the `signing_commitments`, mapping participant's identifiers to their commitments.

```
pub struct SigningPackage<C: Ciphersuite> {
    /// The set of commitments participants published in the first round of the
    /// protocol.
    pub signing_commitments: BTreeMap<Identifier<C>, round1::SigningCommitments<C>>,
```

Accessing a specific participant's commitments is performed by calling the `signing_commitment()` function in *frost-core/src/frost.rs*, which essentially acts as a wrapper returning the signing commitments of the provided `identifier` in the underlying `BTreeMap`, see below.

```
/// Get a signing commitment by its participant identifier.
pub fn signing_commitment(&self, identifier: &Identifier<C>) ->
↳ round1::SigningCommitments<C> {
    self.signing_commitments[identifier]
}
```

This function does not ensure that the `identifier` is present in the map before accessing it. Looking up an identifier which is not present in the `signing_commitments` would result in an unhandled panic. The `signing_commitment` function is called from the `aggregate()` function, and is used to verify the individual shares in case the aggregated signature is invalid, see snippet below.

```
// Verify the signature shares.
for (signature_share_identifier, signature_share) in signature_shares {

    // ...

    // Compute the commitment share.
    let R_share = signing_package
        .signing_commitment(signature_share_identifier)
        .to_group_commitment_share(&binding_factor);
```

This constitutes another instance where a look-up index (the `signature_share_identifier`) is taken from a data structure different than the one being accessed (the `signing_package`), which could result in an unhandled panic.

Binding Factor List

The structure `BindingFactorList` is used to store the participants binding factors in a `BTreeMap`, indexed by their identifiers, see the excerpt provided below from *frost-core/src/frost.rs*.

```
/// A list of binding factors and their associated identifiers.
#[derive(Clone)]
pub struct BindingFactorList<C: Ciphersuite>(BTreeMap<Identifier<C>, BindingFactor<C>>);
```

A few lines below that structure definition, an `index` function is implemented to facilitate accessing the data stored in the underlying map.

```
impl<C> Index<Identifier<C>> for BindingFactorList<C>
where
    C: Ciphersuite,
{
    type Output = BindingFactor<C>;

    // Get the binding factor of a participant in the list.
    //
    // [`binding_factor_for_participant`] in the spec
    //
    // [`binding_factor_for_participant`]: https://www.ietf.org/archive/id/draft-irtf-cfrg-
    ↳ frost-11.html#section-4.3
    fn index(&self, identifier: Identifier<C>) -> &Self::Output {
        &self.0[&identifier]
    }
}
```

Once more, this function does not check that the `identifier` provided as parameter is in the `BindingFactorList`, and would panic if it weren't. Furthermore, in this specific instance, the FROST specification explicitly mandates an error be returned in case the participant is unknown, under algorithm `binding_factor_for_participant()` in Section 4.3. List Operations.

```
Inputs:
...

Outputs:
...

Errors:
- "invalid participant", when the designated participant is
  not known.

def binding_factor_for_participant(binding_factor_list, identifier):
  for (i, binding_factor) in binding_factor_list:
    if identifier == i:
      return binding_factor
  raise "invalid participant"
```

This function is currently used in *frost-core/src/frost.rs* on line 429:

```
429 let binding_factor = binding_factor_list[*signature_share_identifier].clone();
```

And in *frost-core/src/frost/round2.rs* on line 195:

```
194 let binding_factor: frost::BindingFactor<C> =
195     binding_factor_list[key_package.identifier].clone();
```

Recommendation

Remediation of this finding could be performed by first checking that the keys are present in their respective data structure before accessing them. Additionally, consider adding logic ensuring the three inputs to the `aggregate()` function are consistent with each other, namely that the expected identifiers are present in all three data structures.

Modify the behavior around accessing the `binding_factor_list` such that it returns an error when the participant was unknown, as also mandated in the FROST specification.

Location

- *frost-core/src/frost.rs* on line 423
- *frost-core/src/frost.rs* on line 235
- *frost-core/src/frost.rs* on line 106

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 477, and observed that a number of measures had been put in place to address the issues listed in this finding:

- The `Index` implementation of a `BindingFactorList` was replaced by a `get()` function, which now has an `Option` return type, and returns `None` in case the given identifier was not found. This addresses the issue described under the subheading "Binding Factor List".
- The `signing_commitment()` function was updated to return an `Option` type, and returns `None` in case the given identifier was not present in the underlying data structure. This addresses the issue described under the subheading "Signing Package".
- The `aggregate()` function has been augmented with a check ensuring that the Signing Commitments and the Signature Shares have the same set of identifiers, and that they all are present in the Signer Pubkeys. This addresses the issue described under the subheading "Public Key Package".

In addition, a few other improvements related to unchecked accesses were introduced as part of this PR. This finding has been marked "Fixed" as a result.

Low

Missing Signing Package Validation May Cause a Panic

Overall Risk	Low	**Finding ID**	NCC-E008263-2WM
Impact	Medium	**Component**	frost-core
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

A potentially malicious Coordinator may perform a denial-of-service against a participant by inducing a crash triggered by the reception of a Signing Package which is missing that participant's commitment.

Description

From the participants' point of view, the FROST signature process is split into two rounds: in a first round, participants generate nonces and their corresponding public commitment values (which are sent to the Coordinator, who collates them with a Message in a Signing Package); in a second round, participants "sign" the Message by computing their respective *signature shares* on the Signing Package.

Under Section 5.2. Round Two - Signature Share Generation, the FROST draft specification states that participants must validate the `commitment_list` list received from the Coordinator:

> Moreover, each participant MUST ensure that its identifier and commitments (from the first round) appear in commitment_list.

In the implementation, that *commitment list* is a field (`signing_commitments`) of the structure `SigningPackage`, which is one of the parameters to the *signature share generation* process (and received by participants from the Coordinator). This signature share generation is performed by participants by calling the function `sign()`, located in the file *round2.rs*, and partially excerpted below.

```
pub fn sign<C: Ciphersuite>(
    signing_package: &SigningPackage<C>,
    signer_nonces: &round1::SigningNonces<C>,
    key_package: &frost::keys::KeyPackage<C>,
) -> Result<SignatureShare<C>, Error<C>> {
    // Encodes the signing commitment list produced in round one as part of generating [`BindingFactor`], the
    // binding factor.
    let binding_factor_list: BindingFactorList<C> =
        compute_binding_factor_list(signing_package, &key_package.group_public, &[]);
    let binding_factor: frost::BindingFactor<C> =
        binding_factor_list[key_package.identifier].clone();

    // Compute the group commitment from signing commitments produced in round one.
    let group_commitment = compute_group_commitment(signing_package, &binding_factor_list)?;
```

The `sign()` function above does not ensure that the participant's identifier is present in the *commitment list*, in what appears to be a contradiction to the FROST specification. As a result, a participant receiving a `SigningPackage` missing their commitment entry will build a `binding_factor_list` (see line 192 above) that does not include their entry. A panic will then be triggered when trying to access the `binding_factor_list` at a non-existent index (i.e., `thread 'check_sign_with_dealer' panicked at 'no entry found for key'`) in the line highlighted above.

While a malicious coordinator is not explicitly covered by the FROST threat model[5], performing thorough input validation is recommended. These considerations may become particularly more crucial if the role of the coordinator were to be removed and distributed among the participants themselves, as described in Section 7.5. Removing the Coordinator Role.

Recommendation

Ensure that the `signing_package` received as argument in the `sign()` function contains the participant's identifier (i.e., `key_package.identifier`) before proceeding further into the signature share generation process.

Location

frost-core/src/frost/round2.rs

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 452, and observed that the `sign()` function in *frost-core/src/frost/round2.rs* now ensures that the signer's commitment is present in the signing package, and that the signing commitment received as parameter corresponds to the expected one. This is aligned with the recommendation above. As such, this finding has been marked "Fixed".

5. https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-15.html#name-security-considerations

Info

Lack of Zeroization in Ed448 Scalar Inversion

Overall Risk	Informational	**Finding ID**	NCC-E008263-HGL
Impact	High	**Component**	Ed448-Goldilocks
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Non-zeroized values in heap-allocated buffers might be harvested as a consequence of other attacks. In the FROST context, scalar inversion is used only on non-secret values, and therefore cannot leak any secret.

Description

Memory zeroization is about ensuring that secret values do not linger in the system RAM long after they ceased to be used; indeed, some specific attack scenarios (e.g. cold-boot attacks) may allow attackers to observe the state of the system memory after sensitive information has been processed. Since memory zeroization is a second line of defence, and can be expensive and/or cumbersome to apply systematically on all values, it is customary to reserve it for heap-allocated buffers: it is expected that stack buffers are “wiped” promptly after deallocation, since all functions use the same stack space repeatedly.

In the Ed448-Goldilocks crate, inversion of a scalar value is done through exponentiation (using Fermat’s little theorem: the inverse of x modulo p is equal to x^{p-2}). To speed up the inversion, the `Scalar::invert()` function uses a square-and-multiply algorithm with wNAF recoding of the exponent, and a precomputed window of low (odd) powers of the input:

```
    pub fn invert(&self) -> Self {
        let mut pre_comp: Vec<Scalar> = vec![Scalar::zero(); 8];
        let mut result = Scalar::zero();

        let scalar_window_bits = 3;
        let last = (1 << scalar_window_bits) - 1;

        // precompute [a^1, a^3,,..]
        pre_comp[0] = montgomery_multiply(self, &R2);

        if last > 0 {
            pre_comp[last] = montgomery_multiply(&pre_comp[0], &pre_comp[0]);
        }

        for i in 1..=last {
            pre_comp[i] = montgomery_multiply(&pre_comp[i - 1], &pre_comp[last])
        }
```

In Fermat’s little theorem, the exponent is not secret (it’s p-2, and p is public), but the value to invert *may* be a secret scalar. The precomputed values are stored in the `pre_comp` vector, which is heap-allocated, and is not zeroized before release. Therefore, *in case inversion is called on a secret scalar*, the implementation allows secret values to remain indefinitely in the heap (until the buffer is reused, which may take a long time).

Within the FROST context, scalar inversion is used only on denominators in Lagrange polynomials; these scalars depend only upon the share *identifiers*, which are public information (contrary to the share *values*). This potential lack of zeroization has thus no impact in the Zcash FROST implementation.

Recommendation

Since the precomputed window has a known, fixed length (8 elements), it should be allocated on the stack, as a simple `[Scalar; 8]` array. This would avoid leaking secret values to the heap, and may also be slightly faster in practice.

Location

Ed448-Goldilocks/src/field/scalar.rs, line 182

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 33 of the Ed448-Goldilocks crate, and observed that the precomputed window was now defined as a `[Scalar; 8]` array, as suggested in the recommendation above. As such, this finding has been marked “Fixed”.

Info

Minimum Participant Constraint Enforcement Improvements

Overall Risk	Informational	**Finding ID**	NCC-E008263-XLV
Impact	Medium	**Component**	frost-core
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Some invalid participant parameters are not detected early enough in the execution, and may result in an unhandled panic.

Description

The FROST protocol is run with a group of participants from which a sufficient threshold is required in order to produce a valid signature. Section 5. Two-Round FROST Signing Protocol of the specification provides constraints on these parameters:

> The protocol is configured to run with a selection of NUM_PARTICIPANTS signer participants and a Coordinator. NUM_PARTICIPANTS is a positive integer at least MIN_PARTICIPANTS but no larger than MAX_PARTICIPANTS, where MIN_PARTICIPANTS <= MAX_PARTICIPANTS, MIN_PARTICIPANTS is a positive non-zero integer and MAX_PARTICIPANTS is a positive integer less than the group order.

Note that the language here provides explicit constraints on these values, but does not formally specify requirements (e.g. using a MUST statement), and as such may easily be missed by implementers.

The minimum and maximum numbers of participants are required during the key generation procedure, during which the group signing key is split into multiple shares. In the implementation, the process by which the private key is split into shares is performed in the function `split()` in *keys.rs*, an excerpt of which is provided below.

```
pub fn split<C: Ciphersuite, R: RngCore + CryptoRng>(
    key: &SigningKey<C>,
    max_signers: u16,
    min_signers: u16,
    identifiers: IdentifierList<C>,
    rng: &mut R,
) -> Result<(HashMap<Identifier<C>, SecretShare<C>>, PublicKeyPackage<C>), Error<C>> {
    let group_public = VerifyingKey::from(key);

    let coefficients = generate_coefficients::<C, R>(min_signers as usize - 1, rng);
```

In the code excerpt above, an unhandled panic may occur in debug mode when providing a minimum number of signers equal to 0. Since `min_signers` is of unsigned type, subtracting `1` results in an `attempt to subtract with overflow` panic. Note that this happens in debug mode only. In release mode, the computation will wrap around, and will result in a `min_signers` value much larger than the maximum number of signers. This inconsistency would later be caught by the function `generate_secret_polynomial()` in *keys.rs*, which is called later in the execution and ensures that `min_signers` is larger than 1 and not greater than `max_signers`, as can be seen in the code excerpt below.

```rust
pub(crate) fn generate_secret_polynomial<C: Ciphersuite>(
    secret: &SigningKey<C>,
    max_signers: u16,
    min_signers: u16,
    mut coefficients: Vec<Scalar<C>>,
) -> Result<(Vec<Scalar<C>>, VerifiableSecretSharingCommitment<C>), Error<C>> {
    if min_signers < 2 {
        return Err(Error::InvalidMinSigners);
    }

    if max_signers < 2 {
        return Err(Error::InvalidMaxSigners);
    }

    if min_signers > max_signers {
        return Err(Error::InvalidMinSigners);
    }
```

However, the `split()` function being the main entry point for key generation, validity checks should arguably be performed upon calling that function.

Note that the unhandled panic described above may also happen in two other places within the code base

1. In *frost-core/src/frost/keys/dkg.rs*, in the function `part1()`:

```rust
let coefficients = generate_coefficients::<C, R>(min_signers as usize - 1, &mut rng);
```

2. In *frost-core/src/frost/keys/repairable.rs*, in the function `repair_share_step_1()`:

```rust
let rand_val: Vec<Scalar<C>> = generate_coefficients::<C, R>(helpers.len() - 1, rng);
```

Recommendation

Consider adding validity checks on the minimum and maximum number of signers in the `split()` function itself (and in the functions `part1()` and `repair_share_step_1()`).

Location

- *frost-core/src/frost/keys.rs*
- *frost-core/src/frost/keys/dkg.rs*
- *frost-core/src/frost/keys/repairable.rs*

Retest Results

2023-09-20 – Fixed

NCC Group reviewed changes introduced in pull request 453, and observed that a new function, called `validate_num_of_signers`, had been introduced. This function performs appropriate checks on the number of signers, and is now called where appropriate, notably as the first instruction in the `split()` function. This is aligned with the recommendation above. As such, this finding has been marked “Fixed”.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 Engagement Notes

This section includes various remarks and minor observations that are not considered security vulnerabilities, but that the NCC Group team deemed worth reporting.

After the initial engagement, the Zcash team diligently addressed all but two of the below observations. Brief explanations and links to the relevant pull requests have been added in the various notes below, on paragraphs starting with "**Update**".

General Notes on frost-core

- The function `reconstruct()` defined in *frost-core/src/frost/keys.rs* does not ensure that the minimum amount of signers' shares is provided, as also described in that function's documentation, see below.

```
/// The caller is responsible for providing at least `min_signers` shares;
/// if less than that is provided, a different key will be returned.
pub fn reconstruct<C: Ciphersuite>(
    secret_shares: &[SecretShare<C>],
) -> Result<SigningKey<C>, Error<C>> {
    if secret_shares.is_empty() {
        return Err(Error::IncorrectNumberOfShares);
    }
```

  This constitutes a slight deviation from the FROST specification, which states that an `invalid parameters` error should be returned in that case. The relevant excerpt from Section D.1. Shamir Secret Sharing is highlighted below.

```
Errors:
- "invalid parameters", if fewer than MIN_PARTICIPANTS input shares
  are provided.

def secret_share_combine(shares):
  if len(shares) < MIN_PARTICIPANTS:
    raise "invalid parameters"
  s = polynomial_interpolate_constant(shares)
  return s
```

  Consider updating the `reconstruct()` function to return an error if too few input shares are provided.

  Update: pull request 482 addresses the above observation.

- The documentation preceding the generic `deserialize()` function in *frost-core/src/lib.rs* states that it may fail if the deserialization process results in a zero scalar, as highlighted in the code excerpt below.

```
/// A member function of a [`Field`] that attempts to map a byte array `buf` to a [`Scalar`].
///
/// Fails if the input is not a valid byte representation of an [`Scalar`] of the
/// [`Field`]. This function can raise an [`Error`] if deserialization fails or if the
/// resulting [`Scalar`] is zero
///
/// <https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-11.html#section-3.1-3.9>
fn deserialize(buf: &Self::Serialization) -> Result<Self::Scalar, FieldError>;
```

However, the specialized implementations all seem to allow deserialization of a zero scalar. Consider for example the implementation of `deserialize()` for Ed25519 located in *frost-ed25519/src/lib.rs* and provided below, for reference:

```
fn deserialize(buf: &Self::Serialization) -> Result<Self::Scalar, FieldError> {
    match Scalar::from_canonical_bytes(*buf).into() {
        Some(s) => Ok(s),
        None => Err(FieldError::MalformedScalar),
    }
}
```

Note that the implementation is actually consistent with the FROST draft reference, which specifically allows, in Section 3.1. Prime-Order Group, the deserialization of the zero scalar. The generic `DeserializeScalar()` function is defined as follows.

```
DeserializeScalar(buf): Attempts to map a byte array buf to a Scalar s. This function raises an error if deserialization fails; see Section 6 for group-specific input validation steps.
```

In Section 6.1. FROST(Ed25519, SHA-512), an example of the specialized version of that function for Ed25519 is defined, which allows deserializing zero, as highlighted in the excerpt below.

```
DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 32-byte string. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1]. Note that this means the top three bits of the input MUST be zero.
```

Consider updating the documentation of the `deserialize()` function and drop the “or if the resulting [`Scalar`] is zero” part.

Update: pull request 483 fixes the documentation discrepancy.

- There seems to be a minor optimization potential in *frost-core/src/frost/keys.rs*, where a default list of identifiers is allocated even if there already exists a custom list of identifiers. However, the relatively small size of the parameters make this optimization likely futile.

```
let default_identifiers = default_identifiers(max_signers);
let identifiers = match identifiers {
    IdentifierList::Custom(identifiers) => identifiers,
    IdentifierList::Default => &default_identifiers,
};
```

Update: pull request 481 performs the optimization suggested above.

- The multi-scalar multiplication functions (i.e., the function `optional_multiscalar_mul()` in *frost-core/src/scalar_mul.rs* and also reproduced in *reddsa/src/scalar_mul.rs*) could check that the iterators of base points and scalars have the same length (after having iterated over them). The excerpt below shows how the function iterates over `scalars` to obtain the `nafs` vector, and over `elements` to obtain `lookup_tables`.

```
fn optional_multiscalar_mul<I, J>(scalars: I, elements: J) -> Option<Element<C>>
where
    I: IntoIterator,
    I::Item: Borrow<Scalar<C>>,
    J: IntoIterator<Item = Option<Element<C>>>,
{
    let nafs: Vec<_> = scalars
```

```
            .into_iter()
            .map(|c| NonAdjacentForm::<C>::non_adjacent_form(c.borrow(), 5))
            .collect();

        let lookup_tables = elements
            .into_iter()
            .map(|P_opt| P_opt.map(|P| LookupTable5::<C, Element<C>>::from(&P)))
            .collect::<Option<Vec<_>>>()?;
```

It is not uncommon to encounter fairly trivial multi-signature verification bypasses when function execution iterates over lists of different lengths. This does not seem possible in the implementation under review, as signatures and public verification keys are added as a tuple to a verification batch. In the spirit of defense in depth, consider adding consistency checks for these two vectors.

Update: pull request 494 ensures the respective vectors are of equal lengths.

- In the distributed key generation process, implemented in *frost-core/src/frost/keys/dkg.rs*, a small discrepancy with the FROST paper[6] exists in the computation of the challenge `c_i`. In addition to the context string Φ being dropped (which was explicitly called out by the Zcash team) the verification key and the commitment are swapped, as can be seen in the two lines highlighted below.

```
    // Round 1, Step 2
    //
    // > Every P_i computes a proof of knowledge to the corresponding secret
    // > a_{i0} by calculating σ_i = (R_i, μ_i), such that k ← Z_q, R_i = g^k,
    // > c_i = H(i, Φ, g^{a_{i0}} , R_i), μ_i = k + a_{i0} · c_i, with Φ being
    // > a context string to prevent replay attacks.

    let k = <<C::Group as Group>::Field>::random(&mut rng);
    let R_i = <C::Group>::generator() * k;
    let c_i = challenge::<C>(identifier, &R_i, &commitment.0[0].0).ok_or(Error::DKGNotSupported)?;
```

While this does not seem to pose a security vulnerability, it may lead to interoperability issues.

Update: pull request 484 addresses the above discrepancy.

- In general, there is a lack of clarity around the maximum number of signers supported by the FROST implementation. Documentation preceding the `generate_with_dealer()` function in *frost-core/src/frost/keys.rs* indicates that

> [The] number of signers is limited to 255.

The `max_signers` variable used throughout the code case is set as a `u16`, whereas a `u8` would be enough to store that value. Additionally, it is unclear what prevents values larger than 255 to be used, as the implementation does not seem to enforce this upper bound. In comparison, the FROST specification first states (under Section 5. Two-Round FROST Signing Protocol) that:

> MAX_PARTICIPANTS is a positive integer less than the group order.

However, the specification also states under Appendix D.1. Shamir Secret Sharing on that:

6. https://eprint.iacr.org/2020/852.pdf

> MAX_PARTICIPANTS, the number of shares to generate, an integer less than 2^16.

Further clarifications on that topic would be welcome.

Update: pull request 485 removes some ambiguities listed in the above note.

- The documentation of the structure `PublicKeyPackage` in *frost-core/src/frost/keys.rs* is slightly imprecise in that it states that the map `signer_pubkeys` "represents all signers for a signing operation". This map actually tracks all the participants , even if they don't partake in the signing process.

```
pub struct PublicKeyPackage<C: Ciphersuite> {
    /// When performing signing, the coordinator must ensure that they have the
    /// correct view of participants' public keys to perform verification before
    /// publishing a signature. `signer_pubkeys` represents all signers for a
    /// signing operation.
    pub(crate) signer_pubkeys: HashMap<Identifier<C>, VerifyingShare<C>>,
    /// The joint public key for the entire group.
    pub(crate) group_public: VerifyingKey<C>,
```

Update: pull request 485 also clarifies this ambiguity.

- The FROST specification, under Section 3.1. Prime-Order Group is very explicit in its definition of the `DeserializeElement(buf)` function, stating that:

> `This function raises an error if deserialization fails or if A is the identity element of the group`

The implementation fulfills this requirement; trying to deserialize the point at infinity results in a `GroupError::InvalidIdentityElement` error. Specifically, this prevents users of the library to deserialize the group identity as a `VerifyingKey`, defined in *frost-core/src/verifying_key.rs*. However, the NCC Group team noticed that it is still possible to instantiate a `VerifyingKey` from the identify point, as can be seen in the following example.

```
let inf = <C::Group as Group>::identity();
let vk2: VerifyingKey<C> = VerifyingKey::<C>::new(inf);
```

Alternatively, a `VerifyingKey` corresponding to the group identity can also be instantiated from a `SigningKey` set to the zero scalar, which itself can be deserialized as follows.

```
let encoded_zero = <<<Ed25519Sha512 as Ciphersuite>::Group as
↳ Group>::Field>::zero().to_bytes();
let sk = SigningKey::<Ed25519Sha512>::deserialize(encoded_zero).unwrap();;
let r = VerifyingKey::<Ed25519Sha512>::from(&sk);
```

It is important to note that the verification of a trivial Schnorr signature (namely σ = *(R, z) = (point-at-infinity, 0)*) with a key equal to the point-at-infinity will successfully pass for any message, which may be an unwanted behavior. While this observation may not pose any meaningful risk, it does allow adversaries to arbitrarily inflate batches that will still verify.

Update: pull request 496 added a guard ensuring the zero scalar couldn't be deserialized as a `SigningKey`.

Notes on RedDSA

The different hash function definitions for *reddsa/src/frost/redjubjub.rs* (and similarly for *reddsa/src/frost/redpallas.rs*) slightly differ, in spirit, from the specifications in the FROST draft specification. Consider the definition of the hash function `H1` defined for the Jubjub curve:

```
impl Ciphersuite for JubjubBlake2b512 {
    const ID: &'static str = "FROST(Jubjub, BLAKE2b-512)";

    type Group = JubjubGroup;

    type HashOutput = [u8; 64];

    type SignatureSerialization = [u8; 64];

    /// H1 for FROST(Jubjub, BLAKE2b-512)
    fn H1(m: &[u8]) -> <<Self::Group as Group>::Field as Field>::Scalar {
        HStar::<sapling::SpendAuth>::new(b"FROST_RedJubjubR")
            .update(m)
            .finalize()
    }
```

In comparison, `H1` is defined as follows, for the ciphersuite FROST(Ed25519, SHA-512):

```
H1(m): Implemented by computing H(contextString || "rho" || m), interpreting the 64-byte
↳digest as a little-endian integer
```

where `contextString` is set to `"FROST-ED25519-SHA512-v1"`.

While the definition in the *reddsa* crate do not seem to contravene the requirements listed in the specification, it may be advisable to update the instantiation of the function above with something along the lines of `HStar::<sapling::SpendAuth>::new(b"FROST-Jubjub-BLAKE2b-512-v1")`. This comment applies equally to the other hash functions, `H2`, `H3`, `H4`, `H5` and to some extent to `HDKG` as well, as well as the corresponding functions for the Pallas curve.

Observations on Batch Verification

An implementation of batch signature verification is defined in *frost-core/src/batch.rs*, and follows Appendix B.1 RedDSA batch validation of the Zcash Protocol Specification.

- The sampling range for the blinding factor does not strictly follow the protocol specification. In the implementation, sampling is performed by calling the `random()` function on line 119 of batch.rs:

  ```
  119 let blind = <<C::Group as Group>::Field>::random(&mut rng);
  ```

  This function is defined generically in frost-core/src/lib.rs and generates a random value in the `[0, l-1]` range, with `l` the prime order of the group, as can be seen in the excerpt below.

  ```
  /// Generate a random scalar from the entire space [0, l-1]
  ///
  /// <https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-11.html#section-3.1-3.3>
  fn random<R: RngCore + CryptoRng>(rng: &mut R) -> Self::Scalar;
  ```

  This constitutes a slight divergence from the protocol specification, where the sampling range is explicitly set to $\{1 .. 2^{128} - 1\}$, see below.

Choose random $z_j : \mathbb{F}^*_{r_\mathbb{G}} \xleftarrow{R} \{1 \,..\, 2^{128} - 1\}$.

Note that this small discrepancy mostly affects the upper bound, since a non-normative note allows the sampling range to include zero.

Non-normative note: It is also acceptable to sample each z_j from $\{0 \,..\, 2^{128} - 1\}$, since the probability of obtaining zero for any z_j is negligible.

Update: Tracked under issue 444, the Zcash team decided not to address this for now.

- The project team noticed that an empty batch successfully passes batch signature verification. For example, the following test in the context of the test suite in *frost-core/src/tests/batch.rs* successfully passes.

```
/// Test batch verification with a Ciphersuite.
pub fn empty_batch_verify<C: Ciphersuite, R: RngCore + CryptoRng>(mut rng: R) {
    let batch = batch::Verifier::<C>::new();
    assert!(batch.verify(rng).is_ok());
}
```

  Note that this does not constitute a forgery; however, this behavior does not strictly follow the *fail-safe default* principle. Technically, it can be argued that in a batch containing no signatures, all signatures are valid. Thus, the batch does not contain any invalid signature, and as such a returned value of *true* can be considered as the correct one. However, this behavior is currently not documented in the API.

  Update: pull request 487 updated the `verify()` function to return an error if the batch size is 0.

- In the batch verification function defined in frost-core/src/batch.rs, consider replacing the highlighted instances of `self.signatures.len()` with `n` in the following code excerpt:

```
pub fn verify<R: RngCore + CryptoRng>(self, mut rng: R) -> Result<(), Error<C>> {
    let n = self.signatures.len();

    let mut VK_coeffs = Vec::with_capacity(n);
    let mut VKs = Vec::with_capacity(n);
    let mut R_coeffs = Vec::with_capacity(self.signatures.len());
    let mut Rs = Vec::with_capacity(self.signatures.len());
    let mut P_coeff_acc = <<C::Group as Group>::Field>::zero();
```

  Update: pull request 487 also addresses this.

Notes on the Different FROST Versions

The latest FROST draft is currently at version 14[7] and was published on July 10, 2023, while the security review was ongoing. Even though most of the code base under review seems to

7. https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-14.html

implement the draft version 11, the project also follows a few of the most recent updates, notably the inclusion of the group public key into the binding computation.

- The code base references several different versions of the FROST draft. The version most frequently referenced is v11, for which there are over fifty direct links. However, the code base also references version 10 a total of three times, see below for examples.

```
/// [`compute_binding_factors`]: https://www.ietf.org/archive/id/draft-irtf-cfrg-
↳ frost-10.html#section-4.4
```

Figure 1: frost-core/src/frost.rs

```
/// [`compute_group_commitment`]: https://www.ietf.org/archive/id/draft-irtf-cfrg-
↳ frost-10.html#section-4.5
```

Figure 2: frost-core/src/frost.rs

The third instance is in *frost-rerandomized/src/lib.rs* which also includes a reference to version 12 on line 129, see below.

```
// [`aggregate`]: https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-12.html#section-5.3
```

Update: pull request 488 updates the relevant links.

- One other important change introduced by this latest version is a modification of the ciphersuite-specific *Context Strings* used in the different hash functions. Interestingly, the FROST specification maintained the "v11" version component throughout versions 11, 12 and 13. For example, consider the following excerpt from Section 6.1. FROST(Ed25519, SHA-512) version 13:

> The value of the contextString parameter is "FROST-ED25519-SHA512-v11".

In version 14, the version component of the context string has been updated to *v1*, as can be seen in the excerpt of the same section for the latest version of the draft specification.

> The value of the contextString parameter is "FROST-ED25519-SHA512-v1".

The implementation uses the *v11* context string, as can be seen in frost-ed25519/src/lib.rs:

```
/// Context string 'FROST-ED25519-SHA512-v11' from the ciphersuite in the [spec]
///
/// [spec]: https://www.ietf.org/archive/id/draft-irtf-cfrg-
↳ frost-11.html#section-6.1-1
const CONTEXT_STRING: &str = "FROST-ED25519-SHA512-v11";
```

The different context strings will have to be updated to adhere to the latest specification.

Update: pull request 438 updates the different context strings and the test vectors, as highlighted above.

Notes on the IETF Draft

- Under section 5.3. Signature Share Aggregation, an incorrect list is provided as argument to the call to `derive_interpolating_value()`, see below.

  ```
  # Compute the interpolating value
  participant_list = participants_from_commitment_list(
      commitment_list)
  lambda_i = derive_interpolating_value(x_list, identifier)
  ```

  The variable `x_list` should be replaced with `participant_list`.

  Update: pull request 448 of the draft performs the suggested update.

- Under section 7.2. Optimizations, there is a typo in the spelling of `RECOMENDED`, which should be spelled with two *M*s; `RECOMMENDED`.

  ```
  As such, the optimization is NOT RECOMENDED, and it is not covered in this document.
  ```

  Update: pull request 447 of the draft fixes the typo above.

Minor Documentation Notes on the Source Code

This section lists a number of relatively minor observations pertaining to the code base documentation.

- The function `derive_interpolating_value()` defined in the file *frost-core/src/frost.rs* does not directly reference the FROST specification, contrary to other functions in the code base that are direct implementations of functions defined in the FROST specification.

  ```
  /// Generates the lagrange coefficient for the i'th participant.
  #[cfg_attr(feature = "internals", visibility::make(pub))]
  fn derive_interpolating_value<C: Ciphersuite>(
  ```

- The documentation of the function `new()` for the `BindingFactorList` in *frost-core/src/frost.rs* states that it takes a *vector of binding factors* while it actually requires a `BTreeMap` of `Identifier`s and `BindingFactor`s.

  ```
  impl<C> BindingFactorList<C>
  where
      C: Ciphersuite,
  {
      /// Create a new [`BindingFactorList`] from a vector of binding factors.
      #[cfg(feature = "internals")]
      pub fn new(binding_factors: BTreeMap<Identifier<C>, BindingFactor<C>>) -> Self {
          Self(binding_factors)
      }
  ```

- A comment preceding the definition of a `NonceCommitment` in *frost-core/src/frost/round1.rs* refers to a Ristretto point. This seems to be an outdated comment since the *frost-core* code base is now ciphersuite-agnostic.

  ```
  /// A Ristretto point that is a commitment to a signing nonce share.
  #[derive(Clone, Copy, PartialEq, Eq)]
  #[cfg_attr(feature = "serde", derive(serde::Serialize, serde::Deserialize))]
  #[cfg_attr(feature = "serde", serde(try_from = "ElementSerialization<C>"))]
  #[cfg_attr(feature = "serde", serde(into = "ElementSerialization<C>"))]
  pub struct NonceCommitment<C: Ciphersuite>(pub(super) Element<C>);
  ```

- In the following code excerpt from *frost-core/src/frost/round1.rs*, it is slightly unclear what `B` refers to in the documentation of the function `encode_group_commitments()`.

```
/// Implements [`encode_group_commitment_list()`] from the spec.
///
/// Inputs:
/// - commitment_list = [(j, D_j, E_j), ...], a list of commitments issued by each signer,
///   where each element in the list indicates the signer identifier and their
///   two commitment Element values. B MUST be sorted in ascending order
///   by signer identifier.
///
/// Outputs:
/// - A byte string containing the serialized representation of B.
```

  In comparison, the FROST specification under the relevant section (see section 4.3. List Operations) states:

  > This list MUST be sorted in ascending order by identifier.

  A similar observation can be made about the following comment in *frost-core/src/frost.rs*:

```
// Ala the sorting of B, just always sort by identifier in ascending order
```

- The documentation at the beginning of the file *frost-core/src/frost.rs* seems slightly outdated, as the distributed key generation is now implemented.

```
//! This implementation currently only supports key generation using a central
//! dealer. In the future, we will add support for key generation via a DKG,
//! as specified in the FROST paper.
```

- There are two outstanding `TODO`s in the code base (not including ones in tests):

```
// TODO: when serde serialization merges, change this to be simpler?
```

  Figure 3: frost-core/src/frost.rs

```
// TODO: remove this function and use `div_ceil()` instead when `int_roundings`
// is stabilized.
```

  Figure 4: frost-core/src/scalar_mul.rs

Update: pull request 489 addresses the miscellaneous documentation notes provided above.

Notes on the FROST Book

- The FROST Book's tutorial page (https://frost.zfnd.org/tutorial.html), when describing how to add FROST to a project, suggests adding the following to `Cargo.toml`:

```
[dependencies]
frost-ristretto255 = "0.3.0"
```

  This is clearly out of date and will cause the following example code to fail to compile due to function signature mismatches.

- Also out of date are the remarks around serialization, which describes it as an application responsibility and remarks that "The ZF FROST library will also support serde in the future, which will make this process simpler". Serialization support had already been added (using the `serde` feature) at the time of the engagement.

- The tutorial page describes trusted-dealer key generation in section `2`, while distributed key generation is relegated to section `2.1`; this mismatch is somewhat surprising. This page also makes no mention of the fact that full code samples for trusted-dealer key generation and key use exist within backend crates' documentation.
- The subpage for DKG (section 2.1) uses incomplete code samples with inconsistent formatting, and is missing critical logic, as can be seen in this code snippet reproduced verbatim:

```rust
use rand::thread_rng;
use std::collections::HashMap;

use frost_ristretto255 as frost;

let mut rng = thread_rng();

let max_signers = 5;
let min_signers = 3;

// create `participant_identifier` somehow

    let (round1_secret_package, round1_package) = frost::keys::dkg::part1(
        participant_identifier,
        max_signers,
        min_signers,
        &mut rng,
    )?;
```

  It is not clear, for instance, how `participant_identifier` should be created, what its type signature should be, what properties it should or should not have, etc. Full working example code will dramatically reduce the likelihood of errors in user code written by consumers of this crate.

Update: pull request 491 addresses the various notes provided above.

7 FROST Security Requirements

This section aims at collecting security requirements from the latest FROST draft specification[8]. While many requirements are explicitly stated (for example, using MUST statements), some requirements are implicit and could be missed by implementers.

The table below tracks these requirements. The first column represents the section or appendix in which the requirement was found. While the requirements using key words described in RFC2119 and RFC8174 such as "MUST" are straightforward, some requirements are stated using the lowercase version of these key words, and other requirements are implicitly stated, without using any associated key word. The second column identifies the type of requirement, and uses "Implicit" when no key word is attached to that requirement. In the third column, the corresponding snippets from the reference are provided. Most excerpts are copied from the specification as is, although some excerpts have been slightly altered for easier understanding.

Finally, the FROST specification imposes some implicit restrictions on the values of certain parameters, for example by specifying that variables are of type `NonZeroScalar`. These implicit requirements were left out of the following table.

Section	Type	Requirement
2.	Implicit	we assume that secrets are sampled uniformly at random using a cryptographically secure pseudorandom number generator (CSPRNG)
3.1.	Implicit	we use the type NonZeroScalar to denote a Scalar value that is not equal to zero, i.e., Scalar(0)
3.1.	Implicit	DeserializeElement(buf): Attempts to map a byte array buf to an Element A, and fails if the input is not the valid canonical byte representation of an element of the group.
3.1.	Implicit	DeserializeElement(buf): (...) This function raises an error if deserialization fails or if `A` is the identity element of the group.
3.1.	Implicit	DeserializeScalar(buf): (...) This function raises an error if deserialization fails.
3.2.	SHOULD	For concrete recommendations on hash functions which SHOULD be used in practice, see Section 6.
4.2.	Implicit	Under `Errors` in the `derive_interpolating_value()` function: "invalid parameters", if 1) x_i is not in L, or if 2) any x-coordinate is represented more than once in L
4.3.	MUST	Under `Inputs` in the `encode_group_commitment_list()` function: This list MUST be sorted in ascending order by identifier.
4.3.	MUST	Under `Inputs` in the `participants_from_commitment_list()` function: This list MUST be sorted in ascending order by identifier
4.3.	Implicit	Under `Errors` in the `binding_factor_for_participant()` function: "invalid participant", when the designated participant is not known.
4.4.	MUST	Under `Inputs` in the `compute_binding_factors()` function: This list MUST be sorted in ascending order by identifier.
4.5.	MUST	Under `Inputs` in the `compute_group_commitment()` function: This list MUST be sorted in ascending order by identifier.

8. https://www.ietf.org/archive/id/draft-irtf-cfrg-frost-14.html

Section	Type	Requirement
5.	Implicit	`MIN_PARTICIPANTS <= MAX_PARTICIPANTS`, `MIN_PARTICIPANTS` is a positive non-zero integer and `MAX_PARTICIPANTS` is a positive integer less than the group order.
5.	Implicit	`NUM_PARTICIPANTS` is a positive integer at least `MIN_PARTICIPANTS` but no larger than `MAX_PARTICIPANTS`.
5.	MUST	An identifier, which is a NonZeroScalar value denoted `i` in the range `[1, MAX_PARTICIPANTS]` and MUST be distinct from the identifier of every other participant.
5.	SHOULD	The Coordinator SHOULD abort if the signature is invalid
5.	Implicit	FROST assumes that all inputs to each round, especially those of which are received over the network, are validated before use.
5.	Implicit	Any value of type Element or Scalar is deserialized using DeserializeElement and DeserializeScalar.
5.	Implicit	All messages sent over the wire are encoded appropriately, e.g., that Scalars and Elements are encoded using their respective functions.
5.	Implicit	FROST assumes reliable message delivery between the Coordinator and participants in order for the protocol to complete.
5.	Implicit	in order to identify misbehaving participants, we assume that the network channel is additionally authenticated; confidentiality is not required.
5.1.	should	The outputs `nonce` and `comm` from participant `P_i` should both be stored locally and kept for use in the second round.
5.1.	MUST NOT	The `nonce` value is secret and MUST NOT be shared
5.1.	MUST NOT	The nonce values produced by this function MUST NOT be used in more than one invocation of `sign`
5.1.	MUST	the nonces MUST be generated from a source of secure randomness
5.2.	require	Signers additionally require locally held data
5.2.	MUST	Each participant MUST validate the inputs before processing the Coordinator's request
5.2.	MUST	In particular, the Signer MUST validate commitment_list, deserializing each group Element in the list using DeserializeElement from Section 3.1.
5.2.	MUST	If deserialization fails, the Signer MUST abort the protocol
5.2.	MUST	each participant MUST ensure that its identifier and commitments (from the first round) appear in commitment_list
5.2.	require	Applications which require that participants not process arbitrary input messages are also required to perform relevant application-layer input validation checks
5.2.	MUST	Under `Inputs` in the `sign()` function: This list MUST be sorted in ascending order by identifier.
5.2.	MUST	Each participant MUST delete the nonce and corresponding commitment after completing `sign`

Section	Type	Requirement
5.2.	MUST NOT	[Each participant] MUST NOT use the nonce as input more than once to `sign`.
5.3.	MUST	Before aggregating, the Coordinator MUST validate each signature share using DeserializeScalar.
5.3.	MUST	If validation fails, the Coordinator MUST abort the protocol as the resulting signature will be invalid.
5.3.	MUST	Under `Inputs` in the `aggregate()` function: This list MUST be sorted in ascending order by identifier
5.3.	SHOULD	The Coordinator SHOULD verify this signature using the group public key before publishing or releasing the signature.
5.3.	should	Recall that the Coordinator is configured with “group info” which contains the group public key PK and public keys PK_i for each participant, so the group_public_key and PK_i function arguments should come from that previously stored group info.
5.3.	MUST	Under `Inputs` in the `verify_signature_share()` function: This list MUST be sorted in ascending order by identifier
5.3.	Implicit	If the aggregate signature verification fails, the Coordinator can verify each signature share individually to identify and act on misbehaving participants.
5.4.	SHOULD	When the signing protocol does not produce a valid signature, the Coordinator SHOULD abort
5.4.	Implicit	FROST assumes the network channel is authenticated to identify which signer misbehaved.
6.	must	A FROST ciphersuite must specify the underlying prime-order group details and cryptographic hash function.
6.	RECOMMENDED	The RECOMMENDED ciphersuite is (ristretto255, SHA-512)
6.	MUST	The DeserializeElement and DeserializeScalar functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in Section 3.1.
6.	MUST	Future ciphersuites MUST describe how input validation is done for DeserializeElement and DeserializeScalar.
6.	MUST	Future ciphersuites MUST also adhere to these requirements.
6.1.	MUST	Note that this means the top three bits of the input MUST be zero.
6.1.	MUST	implementations MUST check the group equation `[8][z]B = [8]R + [8][c]PK`
6.2.	MUST	Note that this means the top three bits of the input MUST be zero.
6.3.	MUST	implementations MUST check the group equation `[4][z]B = [4]R + [4][c]PK`
6.6.	MUST	Future documents that introduce new ciphersuites MUST adhere to the following requirements.
6.6.	Implicit	H1, H2, and H3 all have output distributions that are close to (indistinguishable from) the uniform distribution.
6.6.	MUST	All hash functions MUST be domain separated with a per-suite context string.

Section	Type	Requirement
6.6.	MUST	The group MUST be of prime-order
6.6.	MUST	deserialization functions MUST output elements that belong to their respective sets of Elements or Scalars, or failure when deserialization fails.
6.6.	Implicit	The canonical signature encoding details are clearly specified
7.	may	The Coordinator may also abort upon detecting a misbehaving participant to ensure that invalid signatures are not produced.
7.1.	Implicit	Mitigating these side-channels requires implementing `G.ScalarMult()`, `G.ScalarBaseMult()`, `G.SerializeScalar()`, and `G.DeserializeScalar()` in constant (value-independent) time
7.2.	NOT RECOMMENDED	As such, the optimization is NOT RECOMENDED [sic], and it is not covered in this document.
7.3.	MUST	The randomness produced in this procedure MUST be sampled uniformly at random.
7.3.	MAY	The Coordinator MAY further hedge against nonce reuse attacks by tracking participant nonce commitments used for a given group key, at the cost of additional state.
7.5.	Implicit	We assume that every participant receives as input from an external source the message to be signed prior to performing the protocol
7.5.	Implicit	After having received all signature shares from all other participants, each participant will then perform `verify_signature_share` and then `aggregate` directly.
7.5.	must	the channel simply must be reliable
7.5.	may	To avoid this denial of service, implementations may wish to define a mechanism where messages are authenticated, so that cheating players can be identified and excluded.
7.6.	must	the entire message must be known in advance of invoking the signing protocol
7.6	MUST	pre-hashing MUST use a collision-resistant hash function with a security level commensurate with the security inherent to the ciphersuite chosen.
7.6	RECOMMENDED	It is RECOMMENDED that applications which choose to apply pre-hashing use the hash function (`H`) associated with the chosen ciphersuite in a manner similar to how `H4` is defined.
7.6	SHOULD	a different prefix SHOULD be used to differentiate this pre-hash from `H4`.
7.7.	RECOMMENDED	it is RECOMMENDED that applications take additional precautions and validate inputs so that participants do not operate as signing oracles for arbitrary messages
C.	Implicit	The function `prime_order_verify` (...) assumes that signature R component and public key belong to the prime-order group.

Section	Type	Requirement
D.	Implicit	The dealer that performs trusted_dealer_keygen is trusted to 1) generate good randomness, and 2) delete secret values after distributing shares to each participant, and 3) keep secret values confidential
D.	MUST	Under `Inputs` in the `trusted_dealer_keygen()` function: `secret_key`, a group secret, a Scalar, that MUST be derived from at least Ns bytes of entropy.
D.	Implicit	It is assumed the dealer then sends one secret key share to each of the NUM_PARTICIPANTS participants, along with vss_commitment
D.	MUST	After receiving their secret key share and `vss_commitment`, participants MUST abort if they do not have the same view of vss_commitment.
D.	MUST	Furthermore, each participant MUST perform `vss_verify(secret_key_share_i, vss_commitment)`, and abort if the check fails.
D.	MUST	The trusted dealer MUST delete the secret_key and secret_key_shares upon completion.
D.	Implicit	Use of this method for key generation requires a mutually authenticated secure channel between the dealer and participants to send secret key shares, wherein the channel provides confidentiality and integrity.
D.1.	Implicit	Under `Inputs` in the function `secret_share_shard()`: MAX_PARTICIPANTS, the number of shares to generate, an integer less than 2^16.
D.1.	MUST	i MUST never equal 0
D.1.	Implicit	Under `Errors` in the function `secret_share_combine()`: "invalid parameters", if fewer than MIN_PARTICIPANTS input shares are provided.
D.2.	MUST	If `vss_verify` fails, the participant MUST abort the protocol, and failure should be investigated out of band.
E.1.	Implicit	Failure to implement DeserializeScalar in constant time can leak information about the underlying corresponding Scalar.