Zcash Foundation

Audit of FROST implementation

JP Aumasson, Adrien Hamelink (Taurus), Omer Shlomovits (ZenGo)

20210323

Contents

1

Summary

The Zcash Foundation solicited our services to review the security of Zcash's new implementation of the FROST (version: Dec. 22, 2020) threshold signature protocol for the RedJubJub elliptic-curve EdDSA-based signature scheme, relying on the JubJub curve.

This FROST implementation is in the RedJubJub repository under src/frost.rs, which includes 404 lines of code including unit tests, with an additional 40 lines of test code in test/frost.rs. We reviewed the version of FROST in the `main` branch at commit `2ebc08f` (Feb. 25, 2021).

The implementation consists of the 2-round FROST protocol, and does not yet implement a distributed key generation (DKG) protocol. Instead, a trusted dealer performs verifiable secret sharing.

We reviewed the implementation in terms of:

- Protocol security in the agreed threat model (active attacker, dishonest marjority)
- Protocol correctness with respect to the original FROST specification
- Potential abuse of weak instances or parameter sets
- Implementation-specific design choices and security controls
- Cryptographic primitives and parameters safety
- General software security and reliability

Note that the code aims to mitigate the leak of sensitive values by zeroizing them, using what may be the most reliable method offered by the Rust language. The implementation also aims to eliminate timing side channels, notably by relying on constant-time arithmetic of the JubJub curves, which ultimately relies on Rust's `conditional_select()`.

We did not find any major security issue, and can state with a reasonable degree of confidence that the implementation is free of serious defects. The code is written in a clear way and well documented, making it easy to follow and match with the specified functionality. Specifically, we observed that preprocessing (`preprocess()` function) and the threshold signinng (`sign()` and `aggregate()` funnctions) match the logic described in figures 2 and 3 of the paper (with the exception of the secure deletion of nonces and commitments in Signing step 6).

Security in the context of the Zcash integration nonetheless depends on other factors, such as the centralized key generation, the aggregator's reliability, the choice of `(t,n)` parameters, the shareholders' adequate protection of shares and other sensitive values, the refreshing of shares (and secure wiping of old shares), and so on.

We report:

- 2 medium-severity issues
- 3 low-severity issues
- 6 observations of quality improvement

Zcash addressed the issues reported and we comment on the mitigations and patches in the "Status" section associated to each issue.

We would like to thank the Zcash Foundation for their trust.

2

Security issues

2.1 Corrupted aggregator behavior goes undetected

Severity: medium

Description

The protocol assumes two roles during signing: regular parties and a “semi-trusted” aggregator party. It is stated in the protocol that the aggregator can be one of the parties participating in the threshold signing and may become corrupted by the adversary. In the protocol, the participants are required to send their signature share to the aggregator who validates it. The aggregator is never required to reveal their share and therefore it is never validated.

While this may not lead to signature forgeries, it enables a malicious aggregator to impact other properties of the protocol. For example, the aggregator could provide a wrong signature share and release a signature that will fail verification, without being detected. If a regular party is trying to cheat during signing, the paper specifies that the aggregator must “. . . identify and report the misbehaving participant, and then abort.”. Here again a malicious aggregator can deviate from the protocol and let colluding regular parties cheat without detection.

Recommendation

We suggest to further clerify the assumptions on the aggregator party. A partial list of solution ideas includes:

1. Add a sub protocol to validate the aggregator actions in case of failled signature: for instance, if all parties play both regular parties and aggregators, assuming reliable broadcast, each party will be able to locally repeat the computation and identify misbehaving aggregator.

2. Limit the aggregator power to specifc time frames by letting regular parties take turns in playing the aggregator.
3. Make sure the aggregator is an outsider party, not holding any input.

Status

Discussed with Zcash, agreed that in the Zcash use case and threat model, this does not constitute a significant risk (as opposed to a more general case). Clarification added in https://github.com/ZcashFoundation/redjubjub/pull/64.

2.2 No strict domain separation in hashing

Severity: medium

Description

In the `gen_rho_i()` function, rho is computed as

```
rho_i = H("FROST_rho" || i || message || B)
```

where `B` is the concatenation of three values `[ j || D_j || E_j ]` for all signers `j`. The length of `message` is variable, and therefore it may be possible to construct the nonces `rho_i` for a different set `B'` than intended.

Without loss of generality, we assume `t=n-1` (full threshold) and that the party IDs used are 1, . . . , `n`. We consider the two following situations in which different sets of parties are asked to generate a signature:

Situation 1

- Signers: {1, ..., `n`},
- Message: `M`
- Commitments: `B = [ 1 || D_1 || E_1 || ... || n || D_n || E_n ]`

The resulting binding factor is `rho_i = H("FROST_rho" || i || M || B)`.

Situation 2

- Signers = {2, ..., `n`}
- Message: `M' = M || [ 1 || D_1 || E_1 ]`
- Commitments: `B' = [ 2 || D_2 || E_2 || ... || n || D_n || E_n ]`

The resulting binding factor is `rho_i' = H("FROST_rho" || i || M' || B') = H("FROST_rho" || i || M || B) = rho_i`. It is the same as in the first situation. Moreover, given `(D_1, E_1)`, it is possible to compute the same nonce `R`

Analysis

We were not able to exploit this any further. In particular, we considered the situation where a single honest signer is given the fake message `M'`. However, this would cause a different challenge `c' = H(R, Y, M')` to be computed.

We do consider this issue as a problem, since an honest party must trust the message and the commmitments given by the signature aggragator, who may be acting maliciously.

Recommendation

This type of collision can be avoided by ensuring the messages given to `gen_rho_i()` are of fixed size. If this is not the case, then it should also be possible to hash the message into a fixed size byte array. For example, `gen_rho_i()` could instead compute `rho_i = H("FROST_rho" || i || H(message) || B)`.

Status

Addressed in https://github.com/ZcashFoundation/redjubjub/pull/63, by hashing the message first.

2.3 Incomplete zeroization of sensitive values

Severity: low

Description

`Secret` type values, namely private key shares `secret_share`, are properly zeroized, however signing nonces in `SigningNonces` structures are not. Furthermore the current implementation of `Zeroize` does not guarantee that zeroizing is not optimized away by the compiler, or reordered.

Recommendation

Currently, `jubjub:Fr` does not implement the `Zeroize` trait, adding it would help zeroizing scalars. We also suggest to implement the `DefaultIsZeroes` trait for `Secret` types.

Status

Addressed in https://github.com/ZcashFoundation/redjubjub/pull/63, by implementing `Zeroize` for `SigningNonces` and calling it on drop, and by implementing `DefaultIsZeros` for every type that uses `jubjub::Fr/Scalar`.

2.4 Deletion of nonces is not handled

Severity: low

Description

The signing protocol security depends on nonces not being reused. To enforce it, the protocol (paper figure 3) requires each party to delete the nonce right after use and before publishing signature share. In the code this translates to each party safely deleting nonces `participant_nonces` in function `sign()`, before returning a `SignatureShare`.

In the code however, as suggested by a comment, the nonces are called by reference and are not deleted. This deviation from the protocol may lead to unsafe re-use of the nonces.

Recommendation

Rust memory safety mechanisms may be leveraged to ensure that `sign()` consumes the nonces instead of passing a reference. This might not be enough and explictly zeroizing the nonces after use should be added as another precaution.

Status

Addressed in https://github.com/ZcashFoundation/redjubjub/pull/63, by taking ownership of the variable and thus leading to a (zeroizing) drop.

2.5 Null nonces supported

Severity: low

Description

If a signer set their nonces `binding` and `hiding` to zero, then the corresponding commitments will be the identity. This leads to the issuer's nonce `k_i = hiding_i + rho_i * binding_i` (in the paper's notation) being equal to zero and independent of the message. Given the resulting signature share `z_i = k_i + lambda_i * s_i * c = lambda_i * s_i * c`, it is possible to recover the secret key `s_i` since `lambda_i` and `c` are public.

Recommendation

The issuer of the commitments can check beforehand that `binding` and `hiding` are not zero. During the group commitment computation, other parties can also verify that the commitments are not the identity. This prevents a party from accidentally revealing their share.

Status

Addressed in https://github.com/ZcashFoundation/redjubjub/pull/65, notably checking that commitments are not the identity.

3

Other observations

Here we report potential improvements and issues that are not security risks:

3.1 Potential API improvements

We highlight several potential API changes that might help consumers of the library:

1. Restrict the maximum number of signers and threshold parameters. At the moment these parameters are defined using `u32` type. Consider using `u8` or to put some hard-coded constant for the maximal intended numbers.
2. Allow to import secret keys. This can be required in certain use cases. The trusted dealer will have a way to instead of randomly picking the secret key, to take existing one.
3. Derive `Debug`, `Eq`, `PartialEq` for participants local public keys. This will allow implementors more freedom to work with public outputs of the key generation.
4. Serialization and deserialization may be implemented for messages. It is hard to imagine how the code can be run over distributed network without such functionality.

Unused values

The `KeyPackage` structure is never used, which leads to the following warnings:

```
warning: field is never read: ‘index‘
   --> src/frost.rs:150:5
    |
150 |     index: u32,
    |     ^^^^^^^^^^
    |
    = note: ‘#[warn(dead_code)]‘ on by default

warning: field is never read: ‘secret_share‘
```

```
   --> src/frost.rs:151:5
    |
151 |     secret_share: Secret,
    |     ^^^^^^^^^^^^^^^^^^^^

warning: field is never read: ‘public‘
   --> src/frost.rs:152:5
    |
152 |     public: Public,
    |     ^^^^^^^^^^^^^^

warning: field is never read: ‘group_public‘
   --> src/frost.rs:153:5
    |
153 |     group_public: VerificationKey<SpendAuth>,
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: 4 warnings emitted
```

3.2 Outdated dependencies

The `byteorder` and `rand_core` crates are outdated versions (presumably, `rand_core` is on purpose pre-0.6). We suggest to update to the latest version after testing it, and potentially reviewing the changes.

3.3 Test coverage

The amount of unit testing should be extended to better cover different success and failure scenarios, and generally better code coverage. For example, we recommend tests for various number of participants and thresholds, valid and invalid ones, and for different preprocessing scenarios.

3.4 `unwrap()` risks

We recommend to be careful with `unwrap()` usage, as it could lead to panics upon `None/Err`. Instead, use pattern matching to gracefully fail.

3.5 Null ID support

When performing Shamir secret sharing, a polynomial `f(x)` is used to generate each party's share of the secret. The actual secret is `f(0)` and the party with ID `i` will be given a share with value `f(i)`. Since a DKG may be implemented in the future, we recommend that the ID 0 be declared invalid.

3.6 Typo in a comment

“langrange” instead of “lagrange” in https://github.com/ZcashFoundation/redjubjub/blob/2ebc08f91078617e5f7f34c37cd244beb850fe9e/src/frost.rs#L492.