HashCloak

# Code Review and Security Assessment For *Silence Laboratories*

Initial Delivery: December 13, 2024

Updated Report: April 9, 2025

**Prepared For:**

Jay Prakash | *Silence Laboratories*

Andrei Bytes | *Silence Laboratories*

Iraklis Leontiadis | *Silence Laboratories*

**Prepared by:**

Hernan Vanegas | *HashCloak Inc.*

## Table Of Contents

## Executive Summary

*Silence Laboratories* engaged *HashCloak Inc.* to audit their Schnorr threshold signing protocol implementation and the quorum change protocol.

During the one-week audit period, we familiarized ourselves with the quorum change protocol and sought to understand the overall details of the source code by reviewing it along with its provided documentation. We assessed the code coverage to gain a deeper understanding of the quality of the current tests written to evaluate the implementation. We then examined the code for errors, missing security checks, early panics, and the use of cryptographic primitives.

For the quorum change protocol, we found informational and low issues related to the checking of the signing threshold and the presence of unused variables. For the key derivation, we found a security issue related to a missing check of one of the variables. Correcting this issue will make the code compliant with BIP-032.

We found that the source code is of good quality. Furthermore, the source code is understandable and easy to follow. We also saw some of the functions and structs without documentation, and we consider that all the elements in the source code should be documented for future contributors and maintainers of the implementation. Although some elements were not documented, this did not affect the audit process.

Overall, we found the issues range from *Low* to *Informational*:

| **Severity** | **Number of Findings** |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 0 |
| Low | 3 |
| Informational | 4 |

## Scope

We audited the Rust code present in the repository https://github.com/silence-laboratories/multi-party-schnorr with tag v0.1.2-beta, which corresponds to commit `146d4a57a82c62cf8d24fbd6b713d9bfc7cd534c`.

## Overview

Threshold signature schemes are secure multiparty computation protocols that allow a set of parties to compute a signature on a message using a private key that is secret shared among the parties. In this audit, we are considering a threshold signature scheme for Schnorr signatures presented in the paper "*Simple Three-Round Multiparty Schnorr Signing with Full Simulatability*" by Yehuda Lindell, published in 2022. This threshold signature scheme has academic and industrial importance due to its use to protect cryptocurrencies.

On the other hand, the quorum change protocol enables the resharing of the secret key between two different sets of parties. Specifically, we assume that the secret key is Shared among a set of parties, called the *old parties*, and we have another set of parties (not necessarily disjoint from the first) called the *new parties*. The goal of the protocol is to compute new random shares of the secret key for the new parties without disclosing the underlying secret key and guaranteeing that the new shares are hiding the same secret key as the one held by the old parties. The specification of this protocol was provided by *Silence Laboratories* and is presented in the Appendix.

## Methodology

Our methodology consisted of studying the specification of each protocol as well as the source code. Following the specification, we mapped the key parts in the source code to see if it accomplishes the specification. Once we find an issue, we check if we write a test if possible to expose the issue, and then we include the issue in the report.

**Concerns from the client:**
According to the client, there are no particular concerns. The team indicated that the protocol does not include a networking implementation and also lacks authenticated channels between the parties. Therefore, these aspects will not be considered in the audit.

## Overview of Evaluated Components

During the audit process, we covered the following files:

- src/common/dlog_proofs.rs
- src/common/math.rs
- src/common/mod.rs
- src/common/utils.rs
- src/sign/messages.rs
- src/sign/types.rs
- src/sign/eddsa.rs
- src/sign/shared_rounds.rs
- src/sign/taproot.rs
- src/derive/derivation.rs
- src/derive/messages.rs
- src/derive/mod.rs
- src/keygen/message.rs
- src/keygen/dkg.rs
- src/quorum_change/messages.rs
- src/quorum_change/mod.rs
- src/quorum_change/pairs.rs
- src/quorum_change/qc.rs
- src/quorum_change/types.rs

Also, we evaluated whether the `curve25519_dalek` crate used in the tests validates if the point belongs to the curve when it is deserialized from bytes. As far as our analysis can tell, this validation is done. Hence, we will not report missing point validations as an issue in this document assuming that this validation is done by the corresponding elliptic curve library. However, we strongly recommend the Client verify whether the underlying elliptic curve performs a point validation in the deserialization process. The lack of point validation can have catastrophic security violations in case it is not performed correctly as specified in the reference paper of the Schnorr threshold signature protocol.

## Code coverage

Using the command `cargo llvm-cov`, we generated the following table for the code coverage per file:

| **Filename** | **Function Coverage** | **Line Coverage** |
|---|---|---|
| common/dlog_proof.rs | 100.00% (6/6) | 96.55% (84/87) |
| common/math.rs | 72.73% (8/11) | 70.00% (91/130) |
| common/mod.rs | 50.00% (2/4) | 50.00% (6/12) |
| common/utils.rs | 77.78% (14/18) | 94.40% (118/125) |
| derive/derivation.rs | 100.00% (5/5) | 100.00% (35/35) |
| keygen/dkg.rs | 95.45% (21/22) | 94.10% (431/458) |
| keygen/messages.rs | 85.71% (12/14) | 92.98% (106/114) |
| keygen/refresh.rs | 71.43% (5/7) | 84.00% (42/50) |
| keygen/types.rs | 50.00% (1/2) | 77.27% (17/22) |
| keygen/utils.rs | 90.00% (9/10) | 99.29% (140/141) |
| quorum_change/messages.rs | 0.00% (0/8) | 0.00% (0/25) |
| quorum_change/pairs.rs | 75.00% (21/28) | 81.03% (94/116) |
| quorum_change/qc.rs | 100.00% (45/45) | 96.71% (1470/1520) |
| quorum_change/types.rs | 100.00% (2/2) | 100.00% (28/28) |
| sign/eddsa.rs | 90.91% (10/11) | 99.24% (131/132) |
| sign/messages.rs | 50.00% (2/4) | 50.00% (6/12) |
| sign/mod.rs | 100.00% (2/2) | 87.50% (21/24) |
| sign/shared_rounds.rs | 100.00% (9/9) | 97.55% (239/245) |
| sign/taproot.rs | 73.33% (11/15) | 93.37% (183/196) |
| sign/types.rs | 100.00% (1/1) | 100.00% (7/7) |
| **Totals** | **83.04% (186/224)** | **93.39% (3249/3479)** |

Four of the files have a code coverage below 80% which are highlighted in red in the previous table:

- For src/common/math.rs, the only function that is not covered is `schnorr_split_private_key_with_lost()`, however, this function is not used throughout the code. Given that this function is part of the public interface of the library, we recommend adding tests to check its correctness.
- For src/common/mod.rs, the functions with no tests are `reduce_from_bytes()` for Curve25591 and K256. We also advise adding tests for these functions.
- For src/keygen/types.rs, the only function that does not have coverage is `generate_refresh()`, which is not a very long function and it is easy to assess whether it has an error. Hence adding a test for this function is up to the client.
- In sign/messages.rs, the only two functions that do not have tests are the `session_id()` functions. Given that those are getter functions, it is not necessary to add tests for them.
- In quorum_change/messages.rs, we suggest the addition of tests for the computation of the sizes of the messages.

In general, we recommend adding tests that should make the protocol fail. For example, in src/keygen/dkg.rs, we recommend testing those corner cases where the protocol should fail, such as when the set of parties does not have the correct number of parties, or if the received party IDs do not contain the ID of the current participant.

# Findings

## SIL-1: Missing checks in the new quorum threshold

**Severity:** Low

**Files affected:**

- src/quorum_change/qc.rs

**Description:** The current implementation does not check that the threshold for new participants should be less than the quorum size. This issue is seen in `QCPartyNew::new()`, `QCPartyOldToNew::new()` as well as in `QCPartyOld::new()`. The current version of the conditional that checks the threshold for new participants is presented next:

```
if new_t < 2 {
   return Err(QCError::InvalidT);
}
```

**Recommendations:** we suggest to modify three of the conditionals that check for the size of `new_t` as follows:

```
if new_t < 2 || new_t > new_parties.len() {
   return Err(QCError::InvalidT);
}
```

**Status:** Solved in commit 00ab81d708875dc7078ed652e927c45d6f591427.

## SIL-2: The messages do not contain the signatures and encryption of their contents

**Type**: Informational

**Files affected**:

- src/sign/messages.rs

- src/keygen/messages.rs

**Description:** In case the protocol is implemented in the presence of a coordinator party, the messages should include the signature of the content of their message. In the current implementation, the structs `KeygenMsg1`, `KeygenMsg2`, `SignMsg1`, `SignMsg2`, and `SignMsg3` do not contain these signatures. We consider this issue not of higher severity because the audited source code does not involve networking. Hence, the addition of the signatures will depend on the type of channels used between parties and the network architecture.

A similar situation is presented in the quorum change protocol. Specifically, Lines ***18***, ***19***, ***24***, ***25***, ***29***, ***35***, ***43***, and ***46*** of the quorum change protocol specification state operations with encryption keys and signatures that are not present in the code. The encryption keys are not set up, the encryptions are not performed, and the signatures are not signed nor checked inside the presented code. Similarly, as mentioned before, those signatures and encryptions are not needed if the parties use private and authenticated channels. However, we recommend taking this into account in case that the implementation may use a channel with other features for the network architecture.

**Impact:** This vulnerability can be exploited by a corrupt coordinator by tampering with the messages before forwarding them to the other parties.

**Suggestion**: Add the signatures to the messages as specified in *Pro*tocol 12, *Steps 2.e,* and *4.c*, and in *Protocol 9*, *Steps 2.d*, and *4.d*, and the mechanisms for checking them in case the library is used in the presence of a coordinator party.

**Status:** Acknowledged.

## SIL-3: Vectors `r2_j_list` and `p_i_j_list` are not being used

**Type:** Informational

**Files affected:**

- src/quorum_change/qc.rs

**Description:** The vectors `r2_j_list` and `p_i_j_list` in the function `process()` for `QCPartyOld<R1Old<G>, G>` (created in Lines 283 and 284) are being filled with data (in Lines 297 and 301, respectively), but they are not being used afterward.

**Suggestion:** We suggest deleting those variables and their posterior calls to the `push()` function.

**Status:** Solved in commit 00ab81d708875dc7078ed652e927c45d6f591427.

## SIL-4: Some `process()` functions may panic

**Type:** Low

**Files affected:**

- src/derive/derivation.rs

**Description:** The functions `DeriveParty::process()` and `SignerParty<R2<G>, G>::process()` in Line 336-339, may panic and do not return a `Result` to handle the error outside the function.

**Suggestion:** For the case of `DeriveParty::process()`, we suggest to return a `Result` in as follows:

```
fn process(self, _: ()) -> Self::Output {
      let (_additive_offset, derived_public_key) = self
            .keyshare
            .derive_with_offset(&self.derivation_path)
            .map_err(|_| DeriveError::DerivationError)?;

      Ok(derived_public_key)
}
```

And for the case of `SignerParty<R2<G>, G>::process()`, we recommend to return an error as follows:

```
let (additive_offset, derived_public_key) = self
            .keyshare
            .derive_with_offset(&self.derivation_path)
            .map_err(|err| SignError::InvalidKeyDerivation(err))?;
```

After creating a proper new variant for the enum `SignError`.

## SIL-5: Missing checks for scalar `I_l`

**Type:** Low

**Files affected:**

- src/keygen/messages.rs

**Description:** According to the specification in BIP-032, inside the function `derive_child_pubkey()`, the algorithm must check that `il_int` is not greater than or equal to the order of the curve, otherwise, the function should return an error given that this is an invalid key.

**Suggestion:** As an example, to perform the check in the `curve25519-dalek`, you can use the following instruction:

```
curve25519_dalek::Scalar::from_canonical_bytes(*il_int)
            .into_option()
            .ok_or(BIP32Error::InvalidChildScalar)?;
```

This works given that the function `from_canonical_bytes()` checks whether the provided number in bytes is in the canonical form for the prime field.

**Status:** During the audit process, the *Silence Laboratories* team realized that BIP-032 does not apply to Curve25591 as BIP-032 is for the secp256k1 curve, and both curves have significant structural differences. Hence, the *Silence Laboratories* team suggested another key derivation algorithm proposed by Khovratovich & Law (2017), where a key derivation algorithm is presented to be compatible with Curve25591. We reviewed the implementation and this issue was solved in commit 00ab81d708875dc7078ed652e927c45d6f591427.

## SIL-6: The file `messages.rs` is empty

**Type:** Informational

**Files affected:**

- src/derive/messages.rs

**Description:** The file `src/derive/messages.rs` is empty.

**Suggestion:** Remove the file to clean the code.

**Status:** Solved in commit 00ab81d708875dc7078ed652e927c45d6f591427.

## SIL-7: Lack of documentation

**Type:** Informational

**Description:** The following files have some struct or function that is missing documentation:

- src/common/math.rs
- src/keygen/types.rs – clarify what is `extra_data` in `KeygenParams`: what is the use of this extra data?
- src/keygen/utils.rs
- src/sign/mod.rs
- src/sign/taproot.rs
- src/derive/derivation.rs
- src/derive/mod.rs

## Additional recommendations and comments

Here, we present some recommendations that are not necessarily a security issue but can be taken into account to improve the quality of the code:

- To implement `BaseMessage` for `KeyGenMsg2`, consider using the macro impl_basemessage.
- Clean up src/keygen/utils.rs by removing the commented source code of the function `_check_secret_recovery()`.
- The parameters `t` and `n` do not need to be in the `KeyEntropy` struct given that they are not random parameters.

Overall, the comments in the key generation protocol are constructive and valuable which makes the protocol easy to follow. This not only makes the audit process easy but also helps the reader to understand the code faster.

We suggest adding more documentation to the derivation algorithms inside the code. In particular, it is important to add what is the reference specification for this implementation. Regarding the quorum change protocol, the *Silence Laboratories* team should consider adding more documentation to the `Pairs` struct to clarify the goal of the struct and its necessity.

Given that the code is structured as a state machine for the parties, it is not always possible to zeroize the secret data stored in the computer memory to prevent attacks. However, we ***strongly*** recommend zeroizing the secret data after its use. The lack of memory zeroization allows an adversary to look at the state of the RAM for sensitive information and use it to tamper with another signing session. It is important to remember that stack-allocated data does not need zeroization. Because of this, a good strategy is storing the data in the stack as much as possible, and implementing the zeroization for the data that must be stored in the heap.

## References

- Lindell, Y. (2022). Simple Three-Round Multiparty Schnorr Signing with Full Simulatability. Cryptology ePrint Archive, Paper 2022/374. https://eprint.iacr.org/2022/374
- Katz, J., & Lindell, Y. (2021). Introduction to modern cryptography (3rd ed.). CRC Press.
- D. Khovratovich and J. Law, "BIP32-Ed25519: Hierarchical Deterministic Keys over a Non-linear Keyspace," *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Paris, France, 2017, pp. 27-31, doi: 10.1109/EuroSPW.2017.47.

## Appendix

### Specification of the quorum change protocol provided by *Silence Laboratories*

```
Input:
      We have two sets of parties, the old group are the P_i's and new
gropup are the P'_i's
      Each of P_1, ... , P_n1 has a PKI of signing keys {pk_1, ... , pk_n1}
and its own signing key sk_i.
      Each of P'_1, ... , P'_n2 has a PKI of signing keys {pk'_1, ... ,
pk'_n2} and its own signing key sk'_i.
      Each P_i, i in {1..n1} has a keyshare with threshold t1, t1 >= n1,
      after the completion of the protocol each P'_i, i in {1..n2} will
output a keyshare TSS(t2, n2).
      The existing public_key known to all parties.

The protocol:
      Phase 1:
                  Each old party P_i:
                  - chooses a random sid_i
                  - chooses a random public encryption key enc_key_i
                  - calculates s_i_0, additive share of the existing
distributed private key, so sum_{i=1}^{n1} s_i_0 = private_key - OK
```

```
                    - samples a random polynomial p_i(x) of degree (t2 - 1)
with a predetermined zero term coefficient p_i(0) = s_i_0 - OK
                    - let P_i(x) = p_i(x) * G - OK
                    - chooses a random r1_i and sets commitment1_i = H(sid_i,
pid_i, P_i(x), r1_i)
                    - calculates signature_i = sign_{sk_i}(sid_i, enc_key_i,
commitment1_i)
                    - broadcast to all (sid_i, enc_key_i, commitment1_i,
signature_i)

                    Each new party P'_i:
                    - chooses a random sid_i
                    - chooses a random public encryption key enc_key_i
                    - calculates signature_i = sign_{sk'_i}(1, sid_i,
enc_key_i)
                    - broadcast to all (sid_i, enc_key_i, signature_i)

        Phase 2:
                    Each old and new parties:
                    - receives all broadcast messages (1, sid_j, enc_key_j,
commitment1_j, signature_j)
                    - calculates final_session_id = H({sid_j})
                    - stores {sid_j}, {enc_key_j} and {commitment1_j}

                    Each old party P_i for each j /in {P'_j}:
                    - chooses a random r2_j and sets commitment2_i =
H(final_session_id, pid_i, pid_j, p_i(j), r2_j)
                    - encrypts commitment2_i with enc_key_j and sends commit
to P'_j

                    Each new party P'_i
                    - receives commitment2_j from all old parties, j /in
{P_j}
                    - stores {commitment2_j}

        Phase 3:
                    Each old party P_i for each j /in {P'_j}:
                    - encrypts (p_i(j), r2_j) with enc_key_j and sends
decommit to P'_j

                    Each old party P_i:
                    - calculates signature_i = sign_{sk_i}(P_i(x), r1_i)
```

```
                  - broadcast to all (P_i(x), r1_i, signature_i)

      Phase 4:
                  Each old party P_i:
                  - receives all broadcast messages (P_j(x), r1_j,
signature_j)
                  - checks that all commitment1_j, j in {P_j} are valid
                  - checks that len(P_i(x)) == t2 and all points are valid
                  - sets P(x) = sum_{j=1}^{n1} P_j(x)
                  - checks that P(0) = public_key
                  - if all checks pass, completes the protocol.

                  Each new party P'_i:
                  - receives all broadcast messages (P_j(x), r1_j,
signature_j)
                  - checks that all {commitment1_j} are valid
                  - checks that len(P_i(x)) = t2 and all points are valid
                  - sets P(x) = sum_{j=1}^{n1} P_j(x)
                  - checks that P(0) = public_key
                  - receives (p_j(i), r2_j) from all old parties, j /in
{P_j}
                  - checks that all {commitment2_j} are valid
                  - checks that P_j(x_i) = p_j(i) * G
                  - sets p_i = sum_{j=1}^n1 p_j(i)
                  - checks that P(x_i) = p_i * G

      Phase 5:
                  New parties P'_i, i in {1..n2} generates OT_seeds.

Output:
      Each new party P'_i outputs (final_session_id, public_key, p_i,
OT_seeds).
```