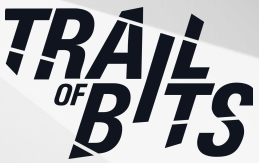

Silence Laboratories Silent Shard

Security Assessment

February 9, 2024

Prepared for:
Andrei Bytes and Jay Prakash

Silence Laboratories

Prepared by: **Joe Doyle and Joop van de Pol**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
November 17, 2023	Pre-project kickoff call
November 27, 2023	Status update meeting #1
December 04, 2023	Status update meeting #2
December 11, 2023	Delivery of report draft
December 12, 2023	Report readout meeting
February 9, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Silence Laboratories engaged Trail of Bits to review the security of the Silent Shard TSS library. Silent Shard implements threshold ECDSA based on the latest work from Doerner et al., also known as DKLs23. This implementation includes the distributed signature generation protocol, all underlying multi-party computation protocols, a distributed key generation algorithm due to Lindell, protocols for peer-to-peer secure messaging and broadcasting, and a system of message relays. The library implements the main parts of the threshold signature scheme, but it does not implement the storage of key shares or pre-signature data, nor does it implement any mechanism to generate consensus on the message to be signed. The library's modular design allows users to choose whether or not to use certain parts, such as the core DKLs23 protocol components and the messaging implementation.

A team of two consultants conducted the review from November 20 to December 8, 2023, for a total of five engineer-weeks of effort. Our testing efforts focused on determining adherence to the specifications provided by the corresponding academic papers, and on the correct usage of cryptographic primitives in the messaging protocols. With full access to source code and documentation, we performed static and dynamic testing of the Silent Shard code, using automated and manual processes. While the implementation supports multiple ways of performing key sharing, only the method based on Lagrange interpolation was in scope of the review.

Observations and Impact

We did not discover any vulnerabilities that directly lead to private key recovery. However, we found several issues that could lead to key destruction, due to several issues: nonce reuse in the peer-to-peer messaging (TOB-SILA-6); insufficient consensus on derived values (TOB-SILA-7, TOB-SILA-8); and replay of messages across sessions (TOB-SILA-9). Some of these issues can be exploited without participating in the protocol. Furthermore, incorrect error handling in the oblivious transfer protocols makes it impossible for a library user to correctly respond to selective abort attacks, which forces the user to choose between key destruction and adversarial key recovery (TOB-SILA-12). It is important to note that some findings apply only if certain components are used (such as the messaging system). However, replacing these components may carry its own risks, as the replacement components may contain security vulnerabilities.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Silence Laboratories take the following steps prior to the release and deployment of the library:

- **Remediate the findings disclosed in this report.** The main risks correspond to the potential of key destruction. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Improve the user documentation, specifically in terms of error handling.** Some very important aspects of the library must be explained to its users. These include user responsibilities, such as generating consensus on the message to be signed; securely storing the sensitive output of the library, including key shares and partial signatures; ensuring consensus on the use of partial signatures to prevent reuse; and providing unique high-entropy seeds for every invocation of the distributed key generation and signing protocols. Furthermore, the library performs several protocol checks that, if they fail, require a user to take specific steps to ensure security. Specifically, the library aborts the current session, but the user needs to abort all concurrent sessions that include the counterparty responsible for causing the failure of this check. Furthermore, the user needs to denylist this counterparty—i.e., refuse to participate in any future protocol execution that includes that specific participant. All of these aspects need to be documented, ideally in the documentation comments for each specific library interface function.

- **Add negative testing.** The implemented unit tests are aimed at verifying the correctness of the implementation in a context where every participant uses the library. However, this will not test the presence and sufficiency of checks that are supposed to ensure correct behavior of other participants during the protocol. We recommend adding negative unit tests that verify correct processing in the context of malicious behavior, including but not limited to receiving malformed messages (e.g., containing empty vectors, shorter vectors, longer vectors). We further recommend defining unit tests for every specific check specified in the academic papers that the implementation is based on, to help ensure that these checks are all present and functioning in the implementation. Randomized property-based testing libraries such as proptest can enable negative testing across a wide range of possible inputs and scenarios with a relatively small additional implementation.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	5
Low	2
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	9
Data Validation	5
Denial of Service	1

Project Goals

The engagement was scoped to provide a security assessment of the Silence Laboratories Silent Shard. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do the following protocols adhere to the specifications provided in the corresponding academic papers?
    - Distributed key generation, including key refresh (Lindell22)
    - Distributed signature generation (DKLs23)
    - Base oblivious transfers (MR19, Roy22)
    - Oblivious transfer extension (KOS15, DKLs18, Roy22)
    - Pairwise multiplication (DKLs18, DKLs19)
    - Zero-sharing sampling (DKLs23)
- Does the implementation enforce all specified verification checks during the protocol?
- Can a group of malicious participants that does not meet the threshold forge signatures, or even obtain the private key?
- Can a malicious minority group of participants destroy the private key?
- Does the implemented peer-to-peer messaging protocol guarantee the confidentiality and authenticity of transmitted messages?
- Does the implemented broadcast messaging protocol guarantee the authenticity of transmitted messages?
- Do the implemented messaging protocols prevent non-participants from influencing the protocol execution (besides preventing message delivery)?
- Is the implementation vulnerable to side-channel analysis?
- Does the library correctly use third-party libraries providing cryptographic primitives?
- Do the WASM bindings introduce any obvious risks to the system (besides potentially causing non-constant-time behavior)?

Project Targets

The engagement involved a review and testing of the targets listed below. Please note that the `sl-crypto` repository contains multiple crates. Only the crates that Silent Shard uses were in scope for the review.

Silent Shard DKLs23

Repository	https://github.com/silence-laboratories/dkls23-rs
Version	`1510c2fafe3cd6866581ce3e2c43c565561b929b`
Type	Rust library

sl-crypto

Repository	https://github.com/silence-laboratories/sl-crypto
Version	`a6b014722a29027d813bcb58720412da68f63d07`
Type	Rust library

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Manual review of distributed key generation:** We reviewed the high-level structure to ensure that the implementation is consistent with the specification in the referenced academic paper. We further focused on common issues in distributed key generation leading to key destruction or key recovery by malicious participants.
- **Manual review of distributed signature generation:** We reviewed the high-level structure to ensure that the implementation is consistent with the specification in the referenced academic paper. We further focused on specific issues related to the unintended denylisting of other parties leading to key destruction, and other common issues in threshold signature generation that can lead to key recovery by malicious participants.
- **Manual review of oblivious transfer protocols:** We reviewed the implementations of the various protocols and compared them against the referenced academic papers to ensure their consistency. We further focused on specific issues related to selective abort attacks and insufficient domain separation that can lead to key recovery.
- **Manual review of message formatting and cryptographic protection:** We reviewed the implementation of the peer-to-peer and broadcast protocols, focusing on the correct use of cryptographic primitives provided by third-party libraries. The main goal was to assess whether the respective protocols provide confidentiality and authenticity where appropriate.
- **Manual review of the message relay interface:** We reviewed the usage of the message relay interface and message tag generation, focusing on whether a malicious network operator can mount attacks by manipulating the delivery and content of messages.
- **Automated testing through static analysis:** We performed static analysis using open-source and in-house tools to detect common issues in Rust code.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- When implementing a protocol that is specified in an academic paper, it is common for the resulting implementation to deviate from the paper. This may happen because the specification in the academic paper is incomplete, or because the implementation requires additional features compared to the specification from the paper. However, these deviations may affect the applicability of the security proofs given in the academic paper. Where possible, we made a best effort to determine whether any obvious attacks exist. However, we cannot guarantee full coverage for all deviations (see also TOB-SILA-1, TOB-SILA-2, TOB-SILA-10, TOB-SILA-13, and TOB-SILA-15).

- We did not perform an in-depth analysis of the handling of cryptographic secrets. Zeroization in the presence of optimizing compilers is difficult. In Rust, it is particularly tricky because of the constraints the compiler imposes on memory management. The compiler can infer significant information about data aliasing and ownership and can create unexpected copies of secret data whenever it determines that there is only one active pointer to the data. The library uses the `zeroize` crate, but does not use the `secrecy` crate to prevent common cases where a move would cause a copy of the value.

- We discovered several issues that allow an attacker to mislead honest participants into believing that other honest participants cheated in such a way that the misled participants will denylist those other honest participants. This will eventually lead to key destruction, as an insufficient number of honest participants will agree to enter a signing session with the other honest participants. However, as this is a unique aspect of the DKLs23 signing scheme, and not much attention has been given in the academic literature to this kind of "framing" attack, it is possible that we did not discover all such issues in the implementation.

- We did not extensively test data serialization and deserialization. Several data structures are encoded and decoded using the bincode library's automatic derivation macros, which generate binary serializers and deserializers from Rust data structure declarations. The 2.0.0-rc3 version is not yet an official release, and may have unknown flaws that could be discovered through techniques such as fuzz testing.

- The `dkls-party` crate provides a command-line interface to run the full system, including distributed key generation and distributed signature generation. It further provides several scripts that invoke this CLI. We did not review this part, choosing instead to focus on the core of the protocol.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
`cargo-audit`	A tool for checking for vulnerable dependencies	Appendix E
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage	Appendix E
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix E
Dylint	An open-source Rust linter developed by Trail of Bits to identify common code quality issues and mistakes in Rust code	Appendix E

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- General code quality issues and unidiomatic code patterns
- General issues with dependency management and known-vulnerable dependencies

Test Results

The results of this focused testing are detailed below.

`cargo-audit`

We ran `cargo-audit` and the results showed that the only dependencies with existing RUSTSEC vulnerabilities come from test code, or code that was not in scope of the audit.

cargo-llvm-cov

The coverage reports for the `dkls23-rs` and `sl-crypto` illustrate areas of the codebase that are missing potential test coverage.

Filename	Function Coverage	Line Coverage	Region Coverage
keygen/dkg.rs	81.82% (45/55)	92.99% (677/728)	73.56% (242/329)
keygen/key_refresh.rs	78.95% (15/19)	92.63% (176/190)	66.94% (81/121)
keygen/messages.rs	42.86% (6/14)	35.71% (15/42)	16.88% (13/77)
keygen/mod.rs	100.00% (2/2)	100.00% (2/2)	100.00% (2/2)
keygen/types.rs	50.00% (6/12)	82.35% (70/85)	65.22% (15/23)
keygen/utils.rs	100.00% (15/15)	96.67% (116/120)	91.84% (45/49)
lib.rs	66.67% (2/3)	93.33% (14/15)	66.67% (4/6)
pairs.rs	87.10% (27/31)	87.93% (102/116)	86.96% (60/69)
proto.rs	90.00% (18/20)	91.89% (68/74)	90.00% (18/20)
setup.rs	25.00% (1/4)	16.67% (1/6)	20.00% (1/5)
setup/keygen.rs	80.85% (38/47)	90.64% (242/267)	76.19% (128/168)
setup/sign.rs	77.27% (34/44)	86.49% (256/296)	72.29% (120/166)
sign/dsg.rs	85.07% (57/67)	93.24% (690/740)	75.94% (322/424)
sign/messages.rs	56.25% (9/16)	56.25% (9/16)	25.00% (18/72)
sign/mod.rs	100.00% (6/6)	100.00% (54/54)	100.00% (11/11)
sign/pairwise_mta.rs	95.00% (19/20)	99.31% (286/288)	90.48% (76/84)
sign/types.rs	0.00% (0/6)	0.00% (0/14)	0.00% (0/8)
Totals	**78.74% (300/381)**	**90.99% (2778/3053)**	**70.75% (1156/1634)**

Filename	Function Coverage	Line Coverage	Region Coverage
sl-mpc-mate/src/bip32.rs	70.37% (19/27)	88.53% (247/279)	73.91% (68/92)
sl-mpc-mate/src/coord.rs	60.00% (18/30)	67.87% (150/221)	54.00% (54/100)
sl-mpc-mate/src/coord/buffered.rs	0.00% (0/15)	0.00% (0/71)	0.00% (0/43)
sl-mpc-mate/src/coord/stats.rs	0.00% (0/14)	0.00% (0/77)	0.00% (0/27)
sl-mpc-mate/src/lib.rs	68.75% (11/16)	79.55% (35/44)	70.59% (12/17)
sl-mpc-mate/src/math.rs	0.00% (0/37)	0.00% (0/192)	0.00% (0/60)
sl-mpc-mate/src/matrix.rs	100.00% (24/24)	93.36% (211/226)	86.76% (59/68)
sl-mpc-mate/src/message.rs	35.51% (49/138)	67.73% (550/812)	38.83% (113/291)
sl-oblivious/src/endemic_ot/endemic_ot.rs	60.87% (14/23)	93.96% (140/149)	52.50% (21/40)
sl-oblivious/src/endemic_ot/messages.rs	16.67% (2/12)	16.67% (2/12)	16.67% (4/24)
sl-oblivious/src/endemic_ot/mod.rs	100.00% (2/2)	100.00% (23/23)	100.00% (8/8)
sl-oblivious/src/lib.rs	50.00% (4/8)	46.27% (31/67)	50.00% (4/8)
sl-oblivious/src/soft_spoken/all_but_one.rs	68.75% (11/16)	97.72% (257/263)	82.35% (70/85)
sl-oblivious/src/soft_spoken/mul_poly.rs	100.00% (3/3)	100.00% (42/42)	100.00% (25/25)
sl-oblivious/src/soft_spoken/soft_spoken_ot.rs	60.00% (27/45)	89.11% (442/496)	69.92% (172/246)
sl-oblivious/src/zkproofs/dlog_proof.rs	64.29% (9/14)	94.34% (100/106)	51.72% (15/29)
Totals	**45.52% (193/424)**	**72.40% (2230/3080)**	**53.74% (625/1163)**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The implementation mainly uses scalar arithmetic as provided by the RustCrypto `k256` crate. The use of the `NonZeroScalar` type prevents many of the common issues in threshold signature scheme implementations.	**Strong**
Auditing	The library does not provide specific auditing functionality.	**Not Applicable**
Authentication / Access Controls	The library does not provide authentication of users or access controls.	**Not Applicable**
Complexity Management	The implementation has a clear structure, with modules dedicated to each of the distinct components. Several of these modules are in separate crates for the purposes of code reuse.	**Satisfactory**
Cryptography and Key Management	The main purpose of this library is to implement a cryptographic scheme, and it correctly uses the required cryptographic primitives to implement this scheme. In some cases, there are issues related to insufficient verification that parties derive the same values (TOB-SILA-7, TOB-SILA-8). Furthermore, the implementation uses cryptography to ensure confidentiality and authenticity of messages exchanged during the protocol. Several issues related to the use of cryptography were found in this context (TOB-SILA-6, TOB-SILA-9, TOB-SILA-14).	**Moderate**
Documentation	At a high level, the code is documented reasonably well. The references to the academic papers that form the basis of each of the components are especially helpful. However, these references are not always fully complete	**Moderate**

	(TOB-SILA-13). Furthermore, the documentation of some of the code is sparse in terms of code comments. Several functions and code snippets have explanatory comments, but the code would benefit from additional comments explaining the purpose of other functions and code snippets. The user documentation is insufficient, as the implemented protocol has strict requirements on error handling by the signing parties, and the library does not provide sufficient information to its users for them to meet these requirements (TOB-SILA-12).	
Memory Safety and Error Handling	The memory safety is strong due to the usage of safe Rust, with only a very limited usage of unsafe Rust. The error handling is insufficient, as the implementation will trigger a panic in many cases where an error could be returned (TOB-SILA-3). In some cases, this even leads to security issues (TOB-SILA-12).	**Moderate**
Testing and Verification	The codebase has reasonable test coverage, but this is exclusively due to positive test cases. There is a distinct lack of negative testing, specifically in cases where other participants may send malformed input (TOB-SILA-1, TOB-SILA-2). We recommend improving coverage with randomized testing and fuzz testing of message deserialization.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	DKG implementation does not enforce length check of committed polynomials	Data Validation	Medium
2	DKG implementation does not enforce zero-knowledge proof verification	Data Validation	Medium
3	Malicious participant can cause panic in targeted participants during DKG	Data Validation	Low
4	Proactive security model is not specified	Cryptography	Informational
5	Parties use same session ID for all-but-one-OT	Cryptography	Informational
6	Communication channels between parties can reuse nonces	Cryptography	High
7	Parties may not agree on root chain code after DKG	Data Validation	Medium
8	Inconsistent DSG session ID causes honest parties to denylist each other	Cryptography	Medium
9	Messages from previous signing sessions can be replayed	Cryptography	Medium
10	Distributed signature generation session ID is not tied to key generation	Cryptography	Informational
11	Additional domain separation can improve defense in depth	Cryptography	Informational

12	Implementation mishandles selective abort attacks	Cryptography	**High**
13	Combining aspects of different protocols has unclear security implications	Cryptography	**Informational**
14	Participants abort when receiving inauthentic messages	Denial of Service	**Low**
15	DSG setup validation does not verify threshold	Data Validation	**Informational**

Detailed Findings

1. DKG implementation does not enforce length check of committed polynomials

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SILA-1
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description

During distributed key generation (DKG), a malicious participant can create and commit to a polynomial that has a larger degree than the threshold. The implementation does not verify the length of the transmitted committed polynomial, and the following uses of the corresponding vector are not consistent.

For example, each individual committed polynomial is added into the aggregate committed polynomial $F(x)$ using the function `add_mut`. This function ignores any higher-degree coefficients that were not already present in the original vector. As a result, all honest parties will truncate the contribution of the individual polynomial to the first T coefficients, where T is the threshold.

However, the implementation later uses the full-length committed polynomial to perform Feldman verification of the shares. This means that honest participants will accept shares that lie on a larger degree polynomial, which would increase the threshold of the resulting key. The only reason why this is not fully exploitable is that the participants use the truncated aggregate vector to verify the public shares of all participants at a later stage.

Nonetheless, it still means that malicious participants can send different committed polynomials to different honest participants, as long as the first T coefficients are the same. By sending a very long committed polynomial to one specific party, they could force this party to spend extra time performing Feldman verification. Additionally, malicious participants can exploit this issue to force other participants to derive incorrect public key shares.

Two malicious participants could collaborate and each send higher order coefficients that cancel each other out to one specific party. This would prevent detection in any of the further consistency checks.

Another side effect is that malicious participants can send shorter committed polynomials. As a result, the higher degree coefficients of the polynomial are implicitly zero. This is disallowed by step 6(b)iii of the DKG protocol as described in section 6.1 of "Simple Three-Round Multiparty Schnorr Signing with Full Simulatability" by Lindell. It is not directly clear that this enables any attack, but any deviation from the protocol may invalidate the security proof.

Exploit Scenario

Two malicious participants send very long committed polynomials to one specific participant, where the higher degree coefficients are complementary (i.e., they negate each other). The specific participant has to process the full polynomials for Feldman verification, which causes other participants to believe that it has timed out. As a result, the participant is punished.

Recommendations

Short term, verify the length of received committed polynomials against the expected length corresponding to the threshold. As a measure for defense in depth, we further recommend repeating this length check for every following use of the received vector, specifically when the vector is zipped together with other vectors.

Long term, ensure that sufficient data validation is in place for all messages received during the protocol execution. This should include, but not be limited to, the length of vectors.

References

- "Simple Three-Round Multiparty Schnorr Signing with Full Simulatability" by Lindell

2. DKG implementation does not enforce zero-knowledge proof verification	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SILA-2
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description

As part of the DKG protocol, each participant must generate a polynomial and share a commitment to this polynomial. This commitment consists of a vector of elliptic curve points, where each point corresponds to one polynomial coefficient times the generator. To show that this commitment was generated honestly, the participant further has to provide a set of zero-knowledge proofs that the participant knows the discrete logarithm of each of the elliptic curve points.

The verification routine in the function `verfiy_dlog_proofs` combines the vector of points and the vector of proofs using `zip`, and returns an error if any of the proofs in the resulting pairs fails to verify. However, if a malicious participant sends an empty vector of proofs (or an empty vector of points), no proofs are verified at all. Alternatively, they could send a shorter vector, which would result in the verification of a smaller number of proofs than expected.

```
for (proof, point) in proofs.iter().zip(points) {
    proof
        .verify(point, &ProjectivePoint::GENERATOR, &mut dlog_transcript)
        .then_some(())
        .ok_or(KeygenError::InvalidDLogProof)?;
}
```

Figure 2.1: The vector of points and the vector of proofs are combined using `zip` (`dkls23-rs/src/keygen/dkg.rs#838–843`)

There are three reasons why this is difficult to exploit directly:

- the committed polynomials are part of the first hash commitment,
- the implementation verifies the public key share of each participant against the aggregate truncated committed polynomial, and
- each participant needs to provide a zero-knowledge proof of knowledge of the discrete logarithm of their public key share.

As a result, it is not possible to mount rogue key attacks. However, this is still a deviation from the documented DKG protocol that affects the validity of the security proofs.

Specifically, the corresponding paper describes that, without the zero-knowledge proofs, the real and ideal distributions of the public key shares can be distinguished.

Recommendations

Short term, verify the length of both the received committed polynomials and of the list of zero-knowledge proofs against the expected length corresponding to the threshold. As a measure for defense in depth, it is further recommended to repeat this length check for every following use of the received vector, specifically when the vector is zipped together with other vectors.

Long term, ensure that sufficient data validation is in place for all messages received during the protocol execution. This should include, but not be limited to, the length of vectors.

3. Malicious participant can cause panic in targeted participants during DKG

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SILA-3
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description

During the DKG protocol, each participant receives a secret share and a committed polynomial from each other participant. They can use their interpolation coordinate and the committed polynomial to verify whether the received secret share really corresponds to an evaluation of the polynomial at the interpolation coordinate. In the implementation, this happens as part of the `feldman_verify` function. If the elliptic curve point that results from evaluating the committed polynomial at the interpolation coordinate is the identity point, `feldman_verify` will return None and key generation will panic, as shown in figures 3.1 and 3.2.

```
let valid = feldman_verify(
    coeffs,
    &x_i_list[my_party_id as usize],
    f_i_val,
    &ProjectivePoint::GENERATOR,
)
.expect("u_i_k cannot be empty");
```

Figure 3.1: Key generation panics if `feldman_verify` returns None (`dkls23-rs/src/keygen/dkg.rs#577–583`)

```
/// Feldman verification
pub fn feldman_verify<C: CurveArithmetic>(
    u_i_k: impl Iterator<Item = C::ProjectivePoint>,
    x_i: &NonZeroScalar<C>,
    f_i_value: &C::Scalar,
    g: &C::ProjectivePoint,
) -> Option<bool> {
    let point: C::ProjectivePoint = u_i_k
        .enumerate()
        .map(|(i, coeff)| {
            // x_i^i mod p
            let val = x_i.pow([i as u64]);

            // x_i^i * coeff mod p
            coeff * val
```

```
        })
        .sum();

    if point.is_identity().into() {
        return None;
    }

    let expected_point = *g * f_i_value;

    Some(point == expected_point)
}
```

Figure 3.2: Feldman verification returns None if the polynomial evaluates to zero (sl-crypto/crates/sl-mpc-mate/src/math.rs#195–220)

A malicious participant can trigger this panic either by sending an empty list of coefficients or by sending participant i a commitment to a polynomial $F(X)$ such that $F(x_i) = 0$.

Exploit Scenario

Alice participates in distributed key generation, and wishes to stall progress. Each round, she chooses another participant to target, and sends an empty polynomial, causing them to crash, and causing a denial of service.

Recommendations

Short term, implement additional data validation on the received polynomials to avoid passing an empty vector to `feldman_verify`, and handle the error case of non-empty polynomials that evaluate to 0 at x_i by returning an error instead of panicking.

Long term, validate all data received from the network as early as possible, and evaluate all panics to ensure that they cannot be deliberately triggered.

4. Proactive security model is not specified

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SILA-4
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description

The DKG protocol can be re-run in "key refresh" mode with the goal of achieving proactive security—i.e., the security of the shared key is restored after a partial compromise of the parties involved. The precise security model intended by the Silence Laboratories team has not been specified, and therefore the security of key refresh cannot be fully evaluated.

The README describes the DKG protocol as based on section 6.1 of "Simple Three-Round Multiparty Schnorr Signing with Full Simulatability" by Lindell. The key refresh protocol implemented in this library appears to follow the technique described in section 8 of that paper: key refresh performs the DKG normally, except that each party's $F_i(X)$ polynomial is required to have $F_i(0) = 0$, and the resulting new shares are added to each party's previous shares. That paper does not describe an adversary model for proactive security.

Related work on proactive security in threshold signature schemes by Boneh, et al. (linked below) illustrates that this definition must be implemented carefully. As discussed in section 8 and appendix B, if the adversary is allowed to corrupt parties during the key refresh process, those corruptions must count toward the "security budget" of both the next and previous epochs. Observe that if the adversary controls $t - 1$ parties during key refresh, learning the secret share of a single additional party in either the prior or following epoch suffices to learn the shared secret key.

This is due to the fact that fixing the constant term of every polynomial reveals one interpolation point to everyone. Therefore, a coalition of $t - 1$ parties can recover the randomizing polynomial of every other party and derive all randomization shares.

Recommendations

Short term, document the intended proactive security model, especially focusing on the adversary's ability to corrupt parties during and between key refreshes.

Long term, specify and document the security models for all provided features.

References

- "Simple Three-Round Multiparty Schnorr Signing with Full Simulatability" by Lindell

- Proactive Refresh for Accountable Threshold Signatures, by Boneh, et al.

5. Parties use same session ID for all-but-one-OT

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SILA-5
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description

All pairs of parties perform two sets of oblivious transfers (OT) during the DKG protocol. In the first set, one party is the sender and the other party is the receiver, whereas in the second set, the roles are reversed. The oblivious transfers comprise so-called “base OTs,”which are implemented using Endemic OT, followed by an OT extension into “all-but-one OTs,” which are implemented using an algorithm from the SoftSpokenOT paper.

Both of these oblivious transfer implementations rely on hash functions to instantiate random oracles and pseudo-random generators. The use of session IDs prevents the possibility of collisions from one use of the hash function to another through repetition of inputs. For the base OTs, this session ID consists of the final session ID of the DKG protocol combined with some constant strings; the index of the party acting as sender; and the index of the party acting as receiver. This guarantees that no pair of parties will have the same instantiation of the hash function, even between the two sets of oblivious transfers with reversed roles.

However, for the OT extension, the session ID merely consists of the final session ID of the DKG protocol, combined with a constant string. This means that all pairs of parties share the same instantiations within one DKG session. It will be difficult, if not impossible, to exploit this, as it would require a large amount of control of the output of the base OTs. Nonetheless, deriving pairwise session IDs similar to those of the base OTs would add additional domain separation at limited additional cost, while improving defense in depth.

Recommendations

Short term, consider deriving pairwise unique session IDs for the all-but-one OTs, similar to the base OT case.

Long term, specify and document choices regarding session IDs.

6. Communication channels between parties can reuse nonces

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-SILA-6
Target: `sl-crypto/crates/mpc-mate/src/message.rs`	

Description

Peer-to-peer communication between participants in both the DKG and signing procedures must be both authenticated and confidential. During key generation and signing, each party generates a fresh X25519 keypair to use for communication. Each pair of parties creates a secure communication channel between them by performing ECDH and deriving an encryption key from the resulting shared secret. That encryption key is used to encrypt each message using the `ChaCha20-Poly1305` authenticated encryption scheme. Each message includes a nonce generated by a `NonceCounter` object that is shared for the duration of the given procedure. Figure 6.1 below excerpts message encryption, and figure 6.2 illustrates the usage of the `NonceCounter` object.

```
pub fn encrypt(
    self,
    start: usize,
    secret: &ReusableSecret,
    public_key: &PublicKey,
    counter: NonceCounter,
) -> Result<Vec<u8>, InvalidMessage> {
    let Self {
        mut buffer,
        kind: _,
    } = self;

    let last = buffer.len() - (TAG_SIZE + NONCE_SIZE);
    let (msg, tail) = buffer.split_at_mut(last);

    // TODO Review key generation!!!
    let shared_secret = secret.diffie_hellman(public_key);

    if !shared_secret.was_contributory() {
        return Err(InvalidMessage::EncPublicKey);
    }

    let key = Zeroizing::new(hchacha::<U10>(
        GenericArray::from_slice(shared_secret.as_bytes()),
        &GenericArray::default(),
    ));
```

```
    let cipher = Aead::new(&key);

    let (data, plaintext) = msg.split_at_mut(start);

    let nonce = counter.nonce();

    let tag = cipher
        .encrypt_in_place_detached(&nonce, data, plaintext)
        .map_err(|_| InvalidMessage::BufferTooShort)?;

    tail[..TAG_SIZE].copy_from_slice(&tag);
    tail[TAG_SIZE..].copy_from_slice(&nonce);

    Ok(buffer)
}
```

Figure 6.1: Message encryption, with shared key derivation and nonce generation highlighted (sl-crypto/crates/sl-mpc-mate/src/message.rs#500–541)

```
handle_encrypted_messages(
    &setup,
    &enc_key,
    &enc_pub_key,
    &abort_tags,
    relay,
    DKG_MSG_R2,
    |base_ot_msg1, party_id| {
…
        let msg3 = KeygenMsg3 {
            base_ot_msg2,
            pprf_output,
            seed_i_j,
            d_i: Opaque::from(d_i),
            big_f_vec: big_f_vec.clone(),
            chain_code_sid: Opaque::from(chain_code_sid),
            r_i_2: Opaque::from(r_i_2),
        };
        Ok(Some(Builder::<Encrypted>::encode(
            &setup.msg_id(Some(party_id), DKG_MSG_R3),
            setup.ttl(),
            &enc_key,
            &enc_pub_key[party_id as usize],
            &msg3,
            nonce_counter.next_nonce(),
        )?))
    },
)
.await?;
```

Figure 6.2: Each KeygenMsg3 message is sent with a distinct nonce (dkls23-rs/src/keygen/dkg.rs#469–517)

Although the `NonceCounter` object prevents each party from using the same nonce twice, no mechanism prevents parties from using a nonce the other party has already used. If two parties send messages to each other using the same nonce, an adversary who intercepts both encrypted messages can craft certain messages that pass message authentication, and can learn the bitwise exclusive-or of those two messages.

The exact impact of this on the system is difficult to characterize precisely, but we have confirmed that it allows a malicious relay that observes a nonce reuse between Alice and Bob to permanently prevent either Alice or Bob from participating in signatures. The relay can either corrupt their state during key generation or cause them to denylist each other during signing. As shown in figure 6.3, a low-entropy or public portion of Alice's message may line up with a secret value in Bob's message, and an adversary can extract the corresponding secret value.

The `ChaCha20-Poly1305` AEAD scheme loses authentication after nonce reuse, but only for that specific nonce. So, for example, a relay who sees messages with reused nonces could perform arbitrary bit flips on the messages and cause them to pass authentication at any later point. By relaying a message with a modified value of `seed_i_j` during key generation, the relay could cause a chosen party to be unable to ever participate in signing together with the corresponding counterparty, since they would consistently generate incorrect zero-shares. During a signing session, a relay that observes a nonce reuse could cause those parties to permanently denylist each other, by corrupting some message between them and triggering an abort. The relay would be able to predictably corrupt only messages that have the reused nonce, so some attacks that would be possible with other AEAD schemes (e.g., AES-GCM) are not possible in this case.

In counter-based encryption such as `ChaCha20-Poly1305`, nonce reuse allows an adversary to learn the exclusive-or of the underlying plaintext values. Due to the pattern of messages exchanged during the protocol, it appears that nonce reuse between Alice and Bob can occur only when they are both on the same step of the protocol, and thus sending the same type of message. This reduces the apparent exploitability of learning the exclusive-or of the plaintext values, since each party's secret values typically appear in the same position in each plaintext. When an eavesdropper only learns their exclusive-or, it does not reveal either secret unless the eavesdropper already knows one of them.

One exception is the `seed_i_j` value used for generating zero-shares. Since only the parties with lower IDs send a seed value, a nonce reuse on `KeygenMsg3`, shown below in figure 6.3, will reveal the exclusive-or of `seed_i_j` and `chain_code_sid`. Since `chain_code_sid` is consistent and is sent to all other parties, the adversary learns the value of `seed_i_j`. Learning enough values of `seed_i_j` would allow an adversary to learn an honest party's value of `mu_i`. According to section 3.2 of "Threshold ECDSA in Three Rounds," an adversary who knows the value of `mu_i` may be able to learn that party's share

"by using its mask values inconsistently among the honest parties," but we have not determined whether this attack is possible in the current implementation.

```
#[derive(Debug, bincode::Encode, bincode::Decode)]
#[derive(Zeroize, ZeroizeOnDrop)]
pub struct KeygenMsg3 {
    /// Participants Fi values
    #[zeroize(skip)]
    pub big_f_vec: GroupPolynomial<Secp256k1>, // == t-1, FIXME:

    ///
    pub d_i: Opaque<Scalar, PF>,

    /// base OT msg 2
    pub base_ot_msg2: EndemicOTMsg2,

    /// pprf outputs
    pub pprf_output: Vec<PPRFOutput>, // 256 / SOFT_SPOKEN_K

    /// seed_i_j values
    pub seed_i_j: Option<[u8; 32]>,

    /// chain_code_sid
    pub chain_code_sid: Opaque<SessionId>,

    /// Random 32 bytes
    pub r_i_2: Opaque<[u8; 32]>,
}
```

Figure 6.3: When the `seed_i_j` field is None, the `chain_code_sid` field is serialized in the same position where the contents of `seed_i_j` would be. (dkls23-rs/src/keygen/messages.rs#19–43)

Exploit Scenario

Alice and Bob are connected to each other through a malicious relay during key generation for a 3-of-5 threshold wallet. They send `KeygenMsg3` messages using the same nonce, and the relay flips a bit in the value of `seed_i_j` Alice sends to Bob. Because of this, Alice and Bob are unable to sign messages with each other. The two malicious parties refuse to sign any messages, creating a denial of service and preventing transfer of funds from the wallet.

Alternatively, during a signing session Alice and Bob send `Round1Output` messages using the same nonce, and the relay flips a bit in the value of `u`, `w_prime`, or `v_prime` in both directions. Because of this, Alice and Bob will encounter errors during specific verification steps of the SoftSpokenOT sender processing. As a result, they need to denylist each other according to the protocol specification.

Note that any attacker positioned in the network could achieve this, so a malicious relay is not strictly required. The attacker does not need to be a participant of the protocol, unlike

most other key destruction attacks described in this report. Therefore, a single attacker could attack different sets of users simultaneously.

Recommendations

Short term, modify the implementation of `Builder<Encrypted>` and `Message` to ensure that a given nonce can never be used twice with the same symmetric key. We recommend implementing a key derivation scheme similar to the one used in the The Noise Protocol Framework, where each side of the encrypted channel derives two different symmetric encryption keys, one for sending and one for receiving. For example, after deriving a shared secret `secret`, Alice derives a receive key `recv_key := hmac(secret, alice_pk)` and a send key `send_key := hmac(secret, bob_pk)`. In effect, two distinct encrypted channels are created, so nonces will be reused only if a single party sends two messages with the same nonce, which a pattern such as the `NonceCounter` type prevents.

Long term, document and review all cryptographic primitives to ensure that all required security properties are satisfied. Whenever possible, use existing mature implementations such as libsodium's `crypto_secretstream` API.

References

- "Threshold ECDSA in Three Rounds" by Doerner et al.

7. Parties may not agree on root chain code after DKG	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SILA-7
Target: `dkls23-rs/src/keygen/dkg.rs`	

Description
One part of the DKG protocol corresponds to the derivation of a common root chain code to be used for hierarchical key derivation. Each party randomly generates a chain code session ID, broadcasts a hash commitment to this value, and finally shares the value when all commitments have been received. All parties verify the commitments and use the IDs to derive a common root chain code, similar to the session ID of the DKG protocol itself.

However, for the root chain code, the parties do not verify that each party has derived the same value, e.g. using an additional broadcast. Therefore, a malicious participant could commit to and subsequently send different chain code session IDs to different parties. As a result, key derivation during the distributed signature generation (DSG) protocol will not be consistent if those parties participate.

Exploit Scenario
In a two-of-three system, a malicious participant Charlie generates two different chain code session IDs and corresponding commitments. Charlie sends one commitment to Alice and the other commitment to Bob. Later, Charlie sends the respective corresponding values to Alice and Bob, who each derive a different root chain code. As a result, Alice and Bob cannot derive consistent child keys, and signing sessions between them will fail. Unless they save transcripts from the DKG session, they will not be able to identify who among Alice, Bob, and Charlie acted maliciously.

Note that due to the use of the relay system, Charlie would have to control the relays that Alice and Bob use to receive broadcast messages.

Recommendations
Short term, add consistency checks on the derivation of the common root chain code, e.g. by broadcasting the result, or by including all received commitments in some computation such that inconsistent commitments are detected.

Long term, specify and document how the consistency of all “common” values is guaranteed.

8. Inconsistent DSG session ID causes honest parties to denylist each other	
Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-SILA-8
Target: `dkls23-rs/src/sign/dsg.rs`	

Description

During the distributed signature generation protocol, each party broadcasts a random session ID. These session IDs are combined into a final session ID, which is used throughout the protocol execution as a domain separator from other sessions. Specifically, it is used to generate the pairwise session IDs for the MtA protocol.

After receiving the first broadcast message and deriving the final session ID, the DSG protocol continues with the following steps:

- Initialize the MtA receivers, perform the first round (which includes the first receiver round of COTe), and provide the resulting peer-to-peer messages to the relay
- Initialize the MtA senders
- Create a digest including the session IDs and commitments of all participants and use it to perform keyshare randomization
- Linearize the keyshare and randomize it
- Receive the peer-to-peer messages and process them as the MtA senders (which includes the first sender round of COTe)

If a malicious participant sends different session IDs to different participants by cheating on the broadcast, this will not be detected until the last of the aforementioned steps. In fact, participants that have inconsistent final session IDs will fail on the COTe consistency check, which requires the sender to denylist the receiver, according to the specification.

Exploit Scenario

A malicious participant sends a different session ID to each other participant. As a result, all of these participants will derive different final session IDs, and they will fail when performing the first MtA sender round from the participant whose message they process first. This will cause them to abort the current session, as well as any session including this participant, and they will not participate in any future sessions with this participant.

By repeating this attack sufficiently many times, it will no longer be possible to reach a threshold and the private key will effectively be destroyed.

Recommendations

Short term, ensure that a mismatch in session ID is detected as soon as possible. For example, including the final session ID in the message ID derivation for all messages after the first broadcast message will prevent any communication between parties with a different view of the final session ID.

Long term, analyze for every message how a malicious party might introduce different views for other participants, and determine during which protocol step these differences will be detected.

9. Messages from previous signing sessions can be replayed

Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-SILA-9
Target: `dkls23-rs/src/{keygen/dkg.rs,sign/dsg.rs}`	

Description

Broadcast and peer-to-peer messages both include a "message ID," which is the hash of the sender's signature verification key, the receiver's verification key (only for peer-to-peer messages), the tag of the message, and the `InstanceId` field of the `setup` struct. This message ID is used to identify which party's message has been received, and to choose the key with which to verify that message, as shown below in figure 9.1.

```
fn decode_signed_message<T: bincode::Decode>(
    tags: &mut Vec<(MsgId, usize)>,
    mut msg: Vec<u8>,
    setup: &ValidatedSetup,
) -> Result<(T, usize), InvalidMessage> {
    let msg = Message::from_buffer(&mut msg)?;
    let mid = msg.id();

    let party_id = pop_tag(tags, &mid).ok_or(InvalidMessage::RecvError)? as _;

    let msg = msg.verify_and_decode(setup.party_verifying_key(party_id).unwrap())?;

    Ok((msg, party_id))
}
```

Figure 9.1: The verifying key used to check a message's signature is chosen depending on the message's ID field (`dkls23-rs/src/sign/dsg.rs#94–107`)

Unless the user-provided `InstanceId` is unique for every session, a malicious relay can record broadcast messages from one signing session, then deliver them again during a later signing session, causing a denial of service. This denial of service is caused by the fact that some parties will derive different session IDs and different shared secrets for peer-to-peer communication. A difference in shared secrets will lead to an abort as soon as one of the parties attempts to process a received peer-to-peer message. The difference in session IDs will trigger the issues described in TOB-SILA-8.

Exploit Scenario

A malicious relay records an old broadcast message from the first round of the DSG protocol. The malicious relay waits for another round of the DSG protocol that has the

same `InstanceId` and where the original sender of the recorded broadcast message is a participant.

In this new round, the malicious relay sends the correct broadcast message of this participant to the first half of the participants, and the old broadcast message to the second half of the participants. This triggers the issues described in TOB-SILA-8, and causes participants to denylist each other.

By repeating this attack sufficiently many times, it will no longer be possible to reach a threshold, and the private key will effectively be destroyed.

Recommendations

Short term, add extra data to the message ID scheme to identify the session that themessages belong to—for example, include the final session ID for all messages after the first broadcast message. Alternatively, add guidance for the library user to choose only unique `InstanceId` values when setting up a DSG session.

Long term, specify and document the message format and ensure that messages can be successfully delivered only once.

10. Distributed signature generation session ID is not tied to key generation

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SILA-10
Target: `dkls23-rs/src/{keygen/dkg.rs,sign/dsg.rs}`	

Description

The paper on which this implementation is based specifies that each signing session is parameterized by two identifiers:

- `sid`, corresponding to the DKG protocol of the private key to be used for signature generation, and
- `sigid`, corresponding to the current signing session.

However, the implementation uses only the second identifier. As a result, the signature generation protocol execution is not tied to any specific DKG execution. It is unlikely that this can be exploited, as the signing session ID is still unique for every session, and the signature generation is tied to a particular key pair through the setup message. Nonetheless, including the first identifier would add additional domain separation to separate signing sessions that occur before and after the participants perform a key refresh.

Recommendations

Short term, add the final session ID from the DKG execution to the `KeyShare` struct and include it in the final session ID of the signature generation protocol.

Long term, create a detailed specification for your own implementation that shows how all aspects of the cited academic papers are addressed.

11. Additional domain separation can improve defense in depth

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SILA-11

Target:
`dkls23-rs/src/keygen/constants.rs`
`dkls23-rs/src/keygen/dkg.rs`
`dkls23-rs/src/sign/dsg.rs`
`sl-crypto/crates/sl-oblivious/src/zkproofs/dlog_proof.rs`
`sl-crypto/crates/sl-oblivious/src/endemic_ot/endemic_ot.rs`
`sl-crypto/crates/sl-oblivious/src/soft_spoken/all_but_one.rs`
`sl-crypto/crates/sl-oblivious/src/soft_spoken/soft_spoken_ot.rs`

Description

The implementation contains various constants used to ensure that the results from hash function invocations in one context cannot be used in another context.

```
let mut hasher = Sha256::new();
hasher.update(b"SL-ChainCodeSID-Commitment");
hasher.update(session_id);
hasher.update(chain_code_sid);
hasher.update(r_i);

HashBytes::new(hasher.finalize().into())
```

Figure 11.1: A constant string is used to initialize a hash function (dkls23-rs/src/keygen/dkg.rs#712–718)

Adding a library version to one or more of the constants, such that a difference in library version leads to an early abort of the protocol, would improve domain separation. If the library ever needs to be updated (e.g., to remediate a vulnerability), this will prevent the case where different protocol participants use an implementation based on different versions of the library.

Furthermore, the SoftSpokenOT implementation uses an uninitialized hash function to derive the masking values, as shown in the following code excerpt. Including the SoftSpokenOT session ID as part of the transcript would strengthen the domain separation.

```
let mut matrix_hasher = blake3::Hasher::new();

for i in 0..KAPPA_DIV_SOFT_SPOKEN_K {
```

```
    [...]
    matrix_hasher.update(u[i].as_ref());
}
```

Figure 11.2: The hash function is not initialized when used to derive masking values. (sl-crypto/crates/sl-oblivious/src/soft_spoken/soft_spoken_ot.rs#226–254)

Finally, these constants are currently spread over the codebase. It would be better to have one location per crate for each of these constants. This makes updating the constants in a consistent way much easier, should it ever be necessary.

Recommendations

Short term, add a library version to one of the constants, such that a difference in library version leads to an early abort of the protocol. Add the SoftSpokenOT session ID to the derivation of masking values to bind it to the transcript.

Long term, restructure the implementation such that all constants for each crate are in one dedicated location.

12. Implementation mishandles selective abort attacks	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-SILA-12
Target: `dkls23-rs/src/sign/pairwise_mta.rs`	

Description

The implemented pairwise multiplication (MtA) relies on correlated oblivious transfer with errors (COTe). The two participants of the COTe protocol are called “sender” and “receiver,” and during one step of the protocol, the receiver has to provide some check values to the sender. If the receiver cheats during the protocol, the sender will detect this using the check values with some probability.

However, whether the sender detects the cheating by the receiver depends on some secret value that belongs to the sender, which was generated during the distributed key generation procedure and is reused in every signing session. Therefore, the response of the sender leaks information about their secret value to a cheating receiver. This is known as a “selective abort attack,” where the receiver selectively causes the sender to abort and learns information about the secret value. After recovering the full secret value, the receiver can learn the sender’s secret inputs to the MtA protocol. In the context of the distributed signing protocol, this will allow a cheating receiver to eventually obtain the private signing key, after compromising sufficiently many senders.

As a result, the paper “Threshold ECDSA in Three Rounds”—that serves as the basis for this implementation—states that any participant experiencing a failure during the COTe procedure needs to abort the current signing session, abort any concurrent signing sessions with the same counterparty that caused the failure, and refuse to participate in any subsequent signing sessions that include this counterparty.

The MtA implementation does not handle the error returned by the COTe implementation, but instead triggers a panic, which gives no information on which party caused the issue. This prevents users of the library from properly handling this situation in the code that calls the library. In the worst case scenario, they would continue new signing sessions with the offending party that they cannot identify. Alternatively, they would have to denylist all other participants in the session, which may lead to key destruction if it is no longer possible to reach a threshold afterwards.

```
let (cot_sender_shares, round2_output) = self
    .state
    .cot_sender
    .process((&round1_output, &alice_input))
    .expect("error while processing soft_spoken ot message round 1");
```

Figure 12.1: The MtA implementation triggers a panic when the COTe process returns an error. (dkls23-rs/src/sign/pairwise_mta.rs#278–282)

Exploit Scenario

A malicious participant performs selective abort attacks on other participants during the COTe protocol in a signing session, potentially causing the processes of the other participants to panic. Since the other participants cannot identify the cause of the crash, they continue new signing sessions with this malicious participant. Over time, the malicious participant recovers the base OT choices of the other participants. They use the knowledge of these choices to (retroactively) recover the input of these other participants in another signing session, one of which corresponds to their private key share. Once the malicious participant has as many shares as the threshold requires, they can reconstruct the private key.

Recommendations

Short term, implement error handling of the COTe function at all layers of the protocol. This means that the error needs to be propagated through the MtA function to the caller in the DSG function. The DSG function subsequently needs to add the specific participant ID to the error when propagating it to the caller in order to enable a library user to implement the required denylisting. Update the documentation of all errors to specify whether the user needs to denylist another user (and which one, if so) when receiving this error.

Long term, add negative test cases for all checks that should result in denylisting another participant as specified by the underlying paper.

References

- "Threshold ECDSA in Three Rounds" by Doerner et al.
- Don’t overextend your oblivious transfer, Trail of Bits blog post

13. Combining aspects of different protocols has unclear security implications

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-SILA-13
Target: `sl-crypto/crates/sl-oblivious/src/soft_spoken/soft_spoken_ot.rs`	

Description

The documentation states that SoftSpokenOT is based on the paper "SoftSpokenOT: Communication–Computation Tradeoffs in OT Extension" (Roy22), but that the instantiation is based on the paper "Actively Secure OT Extension with Optimal Overhead" (KOS15). However, the implementation also uses aspects from "Secure Two-Party Threshold ECDSA from ECDSA Assumptions" (DKLs18), which gives its own specification of the KOS15 protocol which has some differences when compared to the original work.

The original KOS15 paper had a flawed security proof, and this flaw was copied into the specification provided by DKLs18. The Roy22 paper describes this flaw and provides a way to fix the protocol. The authors of the KOS15 paper then released an updated version, providing a new specification for their protocol based on the fixes described in the Roy22 paper. Their specification does not use all optimizations proposed by the Roy22 paper, but notes that those could also be implemented.

The implementation takes various pieces from each of the aforementioned papers:

- DKLs18:
  - Generation of masking values by hashing the receiver's matrix
  - Use of a correction factor τ by the sender
- KOS15:
  - Correctness check against a cheating receiver based on finite field multiplication
- Roy22:
  - Use of all-but-one oblivious transfer to optimize the communication complexity

However, it is not clear whether the individual security proofs offered by the various papers cover the protocol that results from the combination of these aspects. The DKLs18 paper does not give an explicit proof of security for its modifications of the protocol from the KOS15 paper (and the updated version of the DKLs18 paper recommends not relying on its

description of KOS15). The updated KOS15 paper in turn does not describe how the optimizations from the Roy22 paper can be integrated into its protocol. The Roy22 paper does not give very clear specifications for each of its protocols. Finally, all of these papers use different notation, which makes it difficult to determine whether all parameter choices between the different papers match and result in a secure combination.

Without an explicit proof of security for the exact modifications of these protocols, it is possible that a gap exists in the current security proofs that allows for some security flaw to exist in the resulting protocol. Producing an explicit proof of security for the used modifications would give a much higher level of assurance that such flaws do not exist.

Recommendations

Short term, create a specification for your SoftSpokenOT implementation that clarifies which part comes from which referenced academic paper, and analyze how the combination of aspects impacts the security proofs of those papers.

Long term, create a detailed specification for the complete protocol implementation that inlines the specifications for all referenced academic papers.

14. Participants abort when receiving inauthentic messages

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-SILA-14
Target: `dkls23-rs/src/{keygen/dkg.rs,sign/dsg.rs}`	

Description

During both signing and key generation, messages are sent with cryptographic authentication. For broadcasts, the sender's signature is attached to the message. Peer-to-peer messages are encrypted using an Authenticated Encryption with Associated Data (AEAD) scheme, which encrypts some part of the message and attaches a Message Authentication Code (MAC). As illustrated below in figure 15.1, each party generates a list of message tags they will retrieve from the relay. Then, for each message, the tag is used to select the key used for the message's authentication.

```
async fn handle_signed_messages<T, R, F>(
    setup: &ValidatedSetup,
    abort_tags: &[(MsgId, usize)],
    relay: &mut R,
    tag: MessageTag,
    mut handler: F,
) -> Result<(), SignError>
where
    T: bincode::Decode,
    R: Relay,
    F: FnMut(T, usize) -> Result<(), SignError>,
{
    let mut tags = request_messages(setup, tag, relay, false).await?;
    while !tags.is_empty() {
        let msg = relay.next().await.ok_or(SignError::MissingMessage)?;

        check_abort_message(abort_tags, &msg)?;

        let (msg, party_id) = decode_signed_message::<T>(&mut tags, msg, setup)?;

        handler(msg, party_id)?;
    }

    Ok(())
}
```

Figure 14.1: Messages are retrieved based on their tags. (dkls23-rs/src/sign/dsg.rs#721–745)

If any message retrieved from the relay fails this authentication process, the signing or key generation process returns an error. However, any malicious network participant could send a message with a correct message tag and an incorrect signature or MAC, allowing them to force any participant to error out at will.

Exploit Scenario

Alice and Bob attempt to generate a shared key. Carol repeatedly sends both Alice and Bob messages with correct message IDs but with incorrect signatures, causing them to exit and creating a denial of service.

Recommendations

Short term, continue polling the relay for messages until a message with a valid signature is received, and return an error only if a correctly authenticated message fails deserialization or other validation.

Long term, ensure that all data is authenticated before performing any parsing, and discard any non-authenticated data rather than raising an error.

15. DSG setup validation does not verify threshold

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-SILA-15
Target: `dkls23-rs/src/setup/sign.rs`	

Description

During the decoding and validation of the setup message, the threshold of the DSG `setup` struct is defined as the number of participants in the message. However, this number is not compared against the threshold defined by the `KeyShare` instance. As a result, users could inadvertently start signing sessions with the wrong number of participants.

Attempts to sign with fewer participants will fail, either because the resulting public key does not match the expected one, or because assembling the partial signatures will cause an out-of-bounds read of the additive share vectors.

Attempts to sign with more participants will not result in correct presignatures, because assembling the partial signatures uses the threshold defined by the `KeyShare` instance. Any subsequent attempt to combine the partial signatures will fail, either because the number of participants does not match the threshold defined by the `KeyShare` instance, or because the signature verification will fail with high probability.

While this does not appear to be exploitable, it should be noted that this is a deviation from the specification, which states that every participant must check whether the number of participants in the signing protocol is equal to the threshold of the private key.

Recommendations

Short term, compare the threshold from the `KeyShare` struct to the number of parties in the signing `setup` struct.

Long term, create a detailed specification for your own implementation that shows how all aspects of the cited academic papers are addressed.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

We identified the following code quality issues through manual and automatic code review.

- **Use specific types where possible.** Parties have a `party_id` of type `u8` and a `party_idx` of type `usize`. Defining a new type through `typedef` makes the code more readable and prevents certain misuses.
- **Be complete when documenting function parameters for publicly available library functions.** For example, the documentation of the function `dkg::run_inner` explains all parameters except the `relay` parameter.
- **Use parameters as they are documented for better readability.** The DKG function `get_base_ot_session_id` takes a `sender_id` and a `receiver_id` to derive a session identifier for the base OT session. However, when deriving the session identifier for the sender side, the current party ID is used as the `receiver_id` parameter, and the other party ID is used as the `sender_id` parameter. As the opposite is true for the receiver side, the resulting code works. However, it becomes difficult to determine who is the sender and who is the receiver from reading the code.
- **Mark testing code appropriately.** The SoftSpokenOT implementation defines a helper function `generate_all_but_one_seed_ot` that is used in testing only to locally generate all-but-one OT seeds to kickstart the SoftSpokenOT protocol. This function should be marked as `#[cfg(test)]` such that is not compiled in release builds.
- **Unnecessary usage of unsafe.** The implementations of `BorrowDecode` for `Opaque<'de [u8;N]>` and `Opaque<'de ByteArray<N>>` use `unsafe` code in an unnecessary and potentially dangerous way. These `BorrowDecode` implementations do not appear to be needed, and should be removed.

```
impl<'de, const N: usize> BorrowDecode<'de> for Opaque<&'de [u8; N]> {
    fn borrow_decode<D: BorrowDecoder<'de>>(
        decoder: &mut D,
    ) -> Result<Self, DecodeError> {
        let array = decoder.borrow_reader().take_bytes(N)?;

        Ok(Opaque(
            unsafe { &*(array.as_ptr() as *const [u8; N]) },
            PhantomData,
        ))
    }
}
```

Figure C.1: sl-crypto/crates/sl-mpc-mate/src/message.rs#909–920

```
impl<'de, const N: usize> BorrowDecode<'de> for Opaque<&'de ByteArray<N>> {
    fn borrow_decode<D: BorrowDecoder<'de>>(
        decoder: &mut D,
    ) -> Result<Self, DecodeError> {
        let array = decoder.borrow_reader().take_bytes(N)?;

        Ok(Opaque(
            unsafe { &*(array.as_ptr() as *const ByteArray<N>) },
            PhantomData,
        ))
    }
}
```

Figure C.2: sl-crypto/crates/sl-mpc-mate/src/message.rs#953–964

- **Use of `.zip()` iterator method.** The Rust `Iterator::zip` method is frequently used to iterate through two collections in lockstep. However, the `zip` method silently truncates the longer of the two collections, leading to potential flaws such as TOB-SILA-2 if the collections are not the same length. Consider replacing all uses of `zip` with `zip_eq`, which will panic if the collections differ in length, or `zip_longest`, which forces explicit handling of the cases where either collection is shorter.

D. Side-Channel Analysis

As described in the project goals, one of the questions was whether the implementation is vulnerable to side-channel analysis. In general, the implementation consists of constant-time code, using the `subtle` crate where appropriate.

However, as noted in a comment in the implementation of the `eval_pprf` function of the all-but-one oblivious transfer protocol[1], this implementation is not constant time. It contains branches that are conditional on `y_star`, which is a sensitive value corresponding to the receiver's choice bits. While the total number of loop iterations does not depend on this value, for several specific loops, it determines which specific iteration of the loop is skipped. Because the loop iterator is used to index vectors, this may lead to timing differences based on the value.

It is unlikely that this can be exploited using side-channel analysis, because the attacker would need to recover a four-bit value for each of the 64 independent loop iterations from a single execution. Furthermore, each participant executes this function for every other participant and does not initiate any external communication until later in the protocol. Therefore, the attacker would require a local timing side-channel, as the noise caused by the irrelevant operations would hide the timing information required to mount the attack.

Nonetheless, as part of defense in depth, it is always better to prefer constant-time code for handling sensitive values. As shown in the following code segment, the first `continue` statement highlighted in the code segment below can be removed without affecting the result, because the relevant child node of `y_star` will be overwritten after the loop, and the other child node is irrelevant.

```
for i in 1..k {
    let mut s_star_i_plus_1 = vec![[0u8; DIGEST_SIZE]; two_power_k];
    for y in 0..2usize.pow(i as u32) {
        // TODO: Constant time?
        if y == (y_star as usize) {
            continue;
        }

        let mut shake = Shake256::default();
        shake.update(session_id.as_ref());
        shake.update(b"SL-SOFT-SPOKEN-PPRF");
        shake.update(&s_star_i[y]);
        let mut res = [0u8; DIGEST_SIZE * 2];
        shake.finalize_xof().read(&mut res);
        s_star_i_plus_1[2 * y] =
            res[0..DIGEST_SIZE].try_into().unwrap();
        s_star_i_plus_1[2 * y + 1] =
```

[1] Located in `sl-crypto/crates/sl-oblivious/src/soft_spoken/all_but_one.rs`

```
            res[DIGEST_SIZE..].try_into().unwrap();
    }

[...]

    // TODO: fix clippy
    #[allow(clippy::needless_range_loop)]
    for b_i in 0..DIGEST_SIZE {
        s_star_i_plus_1[2 * y_star as usize + ct_x][b_i] =
            t_x_i[i - 1][ct_x][b_i] ^ big_f_i_star[b_i];

        for y in 0..2usize.pow(i as u32) {
            if y == y_star as usize {
                continue;
            }

            s_star_i_plus_1[2 * (y_star as usize) + ct_x][b_i] ^=
                s_star_i_plus_1[2 * y + ct_x][b_i];
        }
    }
```

Figure D.1: The highlighted `continue` statement is not required, as subsequent code overwrites the relevant result.
(sl-crypto/crates/sl-oblivious/src/soft_spoken/all_but_one.rs#157–203)

All other `continue` statements can be replaced by using a combination of dummy vectors and the `subtle` crate to write to the dummy vector when the loop reaches `y_star` and to write to the real vector in all other cases.

E. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

Dylint

Dylint is a linter for Rust developed by Trail of Bits. It can be installed by running the command `cargo install cargo-dylint dylint-link`. To run Dylint, we added a `Cargo.toml` file to the root of the repository with the following content.

```
[workspace.metadata.dylint]
libraries = [
  { git = "https://github.com/trailofbits/dylint", pattern = "examples/general/*" },
]
```

Figure E.1: Metadata required to run Dylint

To run the tool, run `cargo dylint --all --workspace`.

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source-based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the tool, run the command `cargo llvm-cov` in the crate root directory.

F. Supplemental Review of the RVOLE Implementation

In addition to reviewing the provided fixes, Trail of Bits reviewed the modifications made in commit 7422762 of `dkls23-rs` and commit d760a6a in `sl-crypto`. These modifications replace the pairwise MtA protocol with an implementation of the Random Vector Oblivious Linear Evaluation (RVOLE) protocol presented as Protocol 5.2 in the DKLs23 paper.

The implementation follows the specification provided in the DKLs23, except that it fixes two typos in the protocol description:

- The paper incorrectly uses the expression $\theta_{k,k}$ instead of $\theta_{i,k}$ in the formula computing μ in step 3, while the implementation correctly uses the expression `theta[k][i]` – note that the implementation consistently uses this reversed order of indices for `theta`.
- The paper incorrectly uses the expression $\hat{a}_i$ instead of $\hat{a}_k$ in the formula computing $\tilde{a}$, while the implementation correctly uses the expression `a_hat[k]`.

The implementation also specifies that its computational security parameter λ_c is set to 256, and uses a non-standard construction with BLAKE3 to compute a 512-bit digest, shown below in figure H.1. This digest is calculated by appending `H(data||"p1")` and `H(data||"p1"||"p2")`, rather than using BLAKE3's extendable-output-function mode.

```
hasher.update(b"p1");
let mu_prime_hash_1: [u8; 32] = hasher.finalize().into();
hasher.update(b"p2");
let mu_prime_hash_2: [u8; 32] = hasher.finalize().into();
let mu_prime_hash: [u8; 64] =
    [mu_prime_hash_1.as_slice(), mu_prime_hash_2.as_slice()]
        .concat()
        .try_into()
        .expect("Invalid length of mu_prime_hash");

if rvole_output.mu_hash.ct_ne(&mu_prime_hash).into() {
    return Err("Consistency check failed");
}
```

Figure F.1: a 512-bit digest is calculated by calling BLAKE3 with "p1" and "p1p2" appended to the data (`sl-crypto/crates/sl-oblivious/src/rvole.rs#283–295`)

While the DKLs23 RVOLE protocol supports a 256-bit security level, several primitives used in this implementation target only the 128-bit security level, including BLAKE3 and X25519. An improperly high value of λ_c does not obviously lead to an attack against this protocol. Nevertheless, we strongly recommend choosing $\lambda_c = 128$ to ensure that the primitives used achieve the expected security level.

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From January 22 to January 23, 2024 and from January 26 to January 30, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Silence Laboratories team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

For each issue, we reviewed one or more commits that include changes addressing that issue. The commits addressing TOB-SILA-13 also replaced the pairwise MtA protocol with an implementation of the DKLs23 RVOLE protocol. We separately reviewed that new implementation, and provide recommendations in appendix F.

In summary, of the 15 issues described in this report, Silence Laboratories has resolved 14 issues, and has partially resolved the remaining one issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	DKG implementation does not enforce length check of committed polynomials	**Resolved**
2	DKG implementation does not enforce zero-knowledge proof verification	**Resolved**
3	Malicious participant can cause panic in targeted participants during DKG	**Resolved**
4	Proactive security model is not specified	**Resolved**
5	Parties use same session ID for all-but-one-OT	**Resolved**
6	Communication channels between parties can reuse nonces	**Resolved**
7	Parties may not agree on root chain code after DKG	**Resolved**

8	Inconsistent DSG session ID causes honest parties to denylist each other	**Resolved**
9	Messages from previous signing sessions can be replayed	**Resolved**
10	Distributed signature generation session ID is not tied to key generation	**Resolved**
11	Additional domain separation can improve defense in depth	**Resolved**
12	Implementation mishandles selective abort attacks	**Resolved**
13	Combining aspects of different protocols has unclear security implications	**Partially Resolved**
14	Participants abort when receiving inauthentic messages	**Resolved**
15	DSG setup validation does not verify threshold	**Resolved**

Detailed Fix Review Results

TOB-SILA-1: DKG implementation does not enforce length check of committed polynomials
Resolved in commit `2ad27d0`. The DKG implementation now checks the length of the committed polynomials against the threshold, returning an error unless there is an exact match.

TOB-SILA-2: DKG implementation does not enforce zero-knowledge proof verification
Resolved in commit `2ad27d0`. The DKG implementation now checks the length of the list of proofs against the threshold, returning an error unless there is an exact match.

TOB-SILA-3: Malicious participant can cause panic in targeted participants during DKG
Resolved in commit `8b7e704` and commit `d67ecce` The `feldman_verify` function now always returns a `bool` type and the `expect` statement is removed from the DKG implementation. Therefore, the implementation will no longer panic in this scenario.

TOB-SILA-4: Proactive security model is not specified
Resolved in commit `7422762`. The `README.md` document now references section 2 of "Simplified Threshold RSA with Adaptive and Proactive Security" for the definition of proactive security.

TOB-SILA-5: Parties use same session ID for all-but-one-OT
Resolved in commit `f61ba24`. The DKG implementation now derives a dedicated all-but-one session ID which includes the final session ID, the sender ID, and the receiver ID.

TOB-SILA-6: Communication channels between parties can reuse nonces
Resolved in commit `2894325`. The parties now derive different keys depending on the communication direction by computing a SHA-256 hash of the concatenation of the receiver's public key and the shared secret. As a result, the same nonce will never be used twice with the same symmetric key. Although this scheme appears to be secure, the Silence team should consider using a purpose-built cryptographic primitive such as HKDF. For example, deriving the key by running HKDF-extract with the shared secret as the input key material and the receiver's public key as the info field.

TOB-SILA-7: Parties may not agree on root chain code after DKG
Resolved in commit `1573732`. The implementation of DKG now combines the root chain code and the final session ID to create a combined session ID. This combined session ID is used in the transcript for generating a challenge according to the Fiat-Shamir transform. As a result, the protocol will fail if parties do not agree on the root chain code.

TOB-SILA-8: Inconsistent DSG session ID causes honest parties to denylist each other
Resolved in commit daefd6d. The `MtA` messages now include the final session ID as a field in the DSG implementation. If the final session ID of the message does not match the final session ID derived by the receiver, the library returns a dedicated error.

TOB-SILA-9: Messages from previous signing sessions can be replayed
Resolved in commit ed7f592. The documentation comment, shown below in figure F.1, for `SetupBuilder::build` explicitly states that the instance ID must be unique in each setup.

```
/// Build a setup message using the provided parameters and sign it with the given
signing key.
///
/// # Arguments
///
/// * `id`  - The message ID.
/// * `ttl` - The time-to-live for the message.
/// * `key` - The signing key.
///
/// It is *ABSOLUTELY* essential to use a unique `InstanceId` to
/// avoid replay attacks. Replay attacks occur when an attacker
/// intercepts a message and retransmits it multiple times to the
/// recipient, causing unexpected behavior or security
/// vulnerabilities.
///
/// An `InstanceId` is used to derive an ID of all messages within some session.
///
/// # Returns
///
/// A `Vec<u8>` containing the built and signed message, or None if the message
building fails.
///
pub fn build(self, id: &MsgId, ttl: u32, t: u8, key: &SigningKey) -> Option<Vec<u8>>
{
```

Figure F.1: The documentation of `SetupBuilder::build` (dkls23-rs/src/setup/keygen.rs#359–379)

TOB-SILA-10: Distributed signature generation session ID is not tied to key generation
Resolved in commit 3e2188f. The `Keyshare` struct now contains the final session ID from the DKG session. During distributed signature generation, the DKG final session ID is used to derive the DSG final session ID.

TOB-SILA-11: Additional domain separation can improve defense in depth
Resolved in commit cfc2276, commit e57281b, commit a85b1da, and commit 7fefa7c. Each crate now has its own `Label` module, which generates labels from a hard-coded

version and different constants. Additionally, the SoftSpokenOT implementation now initializes the hash function using the session ID when deriving the masking values.

TOB-SILA-12: Implementation mishandles selective abort attacks
Resolved in commit `e540a60` and commit `1d56095`. The SoftSpokenOT implementation now returns an `AbortProtocolAndBanReceiver` error if the consistency check fails. When receiving this error, the DSG implementation now propagates it to the caller as an `AbortProtocolAndBanParty` error, together with the party ID of the receiver. Additionally, the DSG implementation returns the same error when either of the consistency checks fails.

TOB-SILA-13: Combining aspects of different protocols has unclear security implications
Partially resolved in commit `7422762` of `dkls23-rs` and commit `d760a6a` in `sl-crypto`. The SoftSpokenOT implementation no longer uses the modifications from DKLs18, and uses the Fiat-Shamir modification suggested in section 5.1 of DKLs23. However, this modified version still combines aspects of Roy22, KOS15, and the modifications suggested in DKLs23. Silence Laboratories should create an explicit specification of the version of SoftSpokenOT they have implemented, in order to facilitate direct security analysis.

TOB-SILA-14: Participants abort when receiving inauthentic messages
Resolved in commit `23610c4`. The implementation now checks the result of the `decode_signed_message` and `decode_encrypted_message` functions, and ignores the messages that are not properly authenticated or malformed.

TOB-SILA-15: DSG setup validation does not verify threshold
Resolved in commit `d6f196d`. The function `validate_decoded_setup` now verifies whether the DSG threshold is below the `Keyshare` threshold, or whether there are fewer participants than the threshold.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.