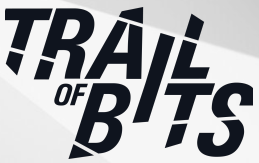

Lit Protocol Cait-Sith

Security Assessment

October 12, 2023

Prepared for:
Chris Cassano
Lit Protocol

Prepared by: **Marc Ilunga** 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

Executive Summary

Engagement Overview

Lit Protocol engaged Trail of Bits to review the security of Cait-Sith, a novel threshold-signing protocol for the Elliptic Curve Digital Signature Algorithm (ECDSA) with presignatures, which relies on oblivious transfer to generate committed Beaver triples.

A team of two consultants conducted the review from May 8 to June 23, 2023, for a total of 10 engineer-weeks of effort. Our testing efforts focused on reviewing the security proofs and the implementation of Cait-Sith. With full access to source code and documentation, we performed static and dynamic testing of Cait-Sith, using automated and manual processes. We also manually reviewed the security proofs of Cait-Sith and its subprotocols.

Observations and Impact

Overall, Cait-Sith appears to be a robust protocol that comes with a reasonable security analysis. The code is well structured, and the implementation offers a clear and easy-to-use interface. Furthermore, the protocol and the implementation are generally well documented. Nonetheless, we identified several inconsistencies and typos in the security proofs. The proofs are rather brief and omit several details, which hurts readability. Regarding the implementation, we observed a lack of negative testing; the specification also lacks several important details needed for safe deployment of the protocol.

We identified high-severity issues leading to the recovery of the signing key due to flaws in the implementation (TOB-LPCS-1) and the protocol specification itself (TOB-LPCS-7). We also identified other issues stemming from a lack of clarity in the protocol specification about the responsibilities of a user of the Cait-Sith protocol and library, which could potentially lead to signing key recovery (TOB-LPCS-3) and denial of service (TOB-LPCS-4, TOB-LPCS-8).

Recommendations

Based on the codebase maturity evaluation and the findings identified during the security review, Trail of Bits recommends that Lit Protocol take the following steps before deployment:

- **Remediate the findings disclosed in this report.** The issues reported in the Detailed Findings section highlight severe risks to the safe deployment of Cait-Sith. These findings should be addressed as part of a direct remediation or as part of any refactoring that may occur when addressing other recommendations.
- **Implement negative testing.** The implementation of Cait-Sith lacks testing for failure scenarios. Although the test coverage is satisfactory, the code is tested only for the intended behavior. Negative testing is important to ensure the robustness of

the implementation in unwanted circumstances and helps detect specific implementation flaws when test cases are properly defined based on the specification.

- **Improve the documentation.** The documentation of Cait-Sith, including the specification and code comments, lacks important information. We found a few instances where the documentation is unclear about the responsibilities of the caller to ensure safe use of the protocol.

- **Subject the Cait-Sith protocol and associated security proofs to peer review.** A time-boxed security assessment is not an appropriate venue to gain full assurance of the validity of cryptographic security proofs. The larger academic community should be involved to obtain such assurance.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	3
Low	0
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	5
Data Exposure	1
Data Validation	2
Denial of Service	2
Error Reporting	1
Patching	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Marc Ilunga, Consultant
marc.ilunga@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
May 5, 2023	Pre-project kickoff call
May 15, 2023	Status update meeting #1
May 22, 2023	Status update meeting #2
May 30, 2023	Status update meeting #3
June 5, 2023	Status update meeting #4
June 16, 2023	Status update meeting #5
June 29, 2023	Delivery of report draft
July 7, 2023	Report readout meeting
July 27, 2023	Delivery of comprehensive report
October 12, 2023	Delivery of comprehensive report with fix review

Project Goals

The engagement was scoped to provide a security assessment of the Cait-Sith protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

Cryptographic Proof Review

- Are the protocol's formal security proofs logically sound?
- Do the formal proofs rely on any unreasonable security assumptions?
- Does Cait-Sith use security definitions that appropriately capture the correctness and security requirements of secure threshold-signing schemes?
- Are there security proofs for each of the subprotocols in Cait-Sith?
- Does the use of the subprotocols in Cait-Sith meet the assumptions of their corresponding security proofs?
- Are there any critical design flaws that could result in the compromise of data confidentiality or integrity?
- Is the design vulnerable to any known cryptographic attacks?
- Does the system use any weak cryptographic algorithms?

Cryptographic Code Review

- Does the codebase rely on any known insecure or outdated dependencies?
- Does the implementation properly follow the protocol specification?
- Are the Beaver triples and presignatures generated and used securely?
- Are there any exposures to known cryptographic attacks?
- Can the support for arbitrary numbers of parties and thresholds be maliciously manipulated to produce unintended results?
- Are there gaps in the unit tests?
- Are there aspects of the API that seem error-prone or unintuitive?

- What key management practices are in place?
- Could the system experience a denial of service?
- Are all inputs and system parameters properly validated?
- Does the codebase conform to industry best practices?

Project Targets

The engagement involved a review and testing of the targets listed below.

Cait-Sith

Repository	cronokirby/cait-sith
Version	`e08a60f7601cc8d20ad78973e13ff1b7318f453b`
Type	Rust

Cait-Sith security analysis

Repository	cronokirby/blog4
Version	`bd1866706a3a9276821368595ca9a6420033d1fa`

Homogeneous Key Derivation Proposal Cait-Sith security analysis

Specification	Homogeneous Key Derivation
Version	Accessed on June 13th

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. We reviewed the following components:

- **Specification.** The Cait-Sith protocol is defined in a specification that outlines the subprotocols used in Cait-Sith and how they work together toward providing a secure threshold-signing service for the ECDSA. We reviewed the protocol specification and compared it to the implementation to ensure that the implementation is consistent with the protocol specification.

- **Triple generation.** Cait-Sith relies on committed Beaver triples to enable ECDSA threshold signing. To do so, triples are generated using several protocols for oblivious transfer and multiplication. We reviewed the triple generation protocol for various concerns, including whether subprotocols interact soundly. We also investigated how the protocol ensures that triples that are generated independently of the signing protocol are not reused, which could result in key recovery.

- **Key generation.** This protocol allows parties to generate a signing key in a distributed manner; any subset of parties that meets the threshold may reconstruct the key. We reviewed the key generation protocol and verified its adherence to the specification and its security against malicious parties that may try to influence or recover the shared signing key.

- **Signing.** The signing protocol proceeds in two phases: the presigning phase generates message-independent presignatures using Beaver triples, and the signing phase consumes the presignature to produce a signature on a given message. We focused on the correct implementation of the signing and presigning protocols and paid special attention to the management of presignatures wherein reuse may lead to forgeries or key recovery.

- **Security proofs.** Cait-Sith comes with security proofs carried out in the Modular Protocol Security framework. We reviewed the proofs, focusing on their high-level soundness. We performed an in-depth review of the echo broadcast and signing protocol proofs and a high-level review of the proofs for the key generation and triple generation protocols.

- **Key derivation and rerandomized presignatures.** Lit Protocol is considering using an additive key derivation scheme to derive additional signing keys from established keys. Lit Protocol is also considering a procedure for rerandomizing presignatures to prevent certain attacks on the threshold-signing protocol for ECDSA with presignatures. We reviewed both proposals to determine whether they are applicable to Cait-Sith and whether they provide additional security guarantees.

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:

- In-depth review of the security proof for triple generation
- In-depth review of the security proof for key generation

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 Cargo plugin for reviewing project dependencies for known vulnerabilities	Appendix C
`cargo-geiger`	A tool that lists statistics related to the use of unsafe Rust code in a Rust crate and all its dependencies	Appendix C
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage	Appendix C
`cargo-edit`	A Cargo plugin that identifies project dependencies with newer versions available	Appendix C
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Dylint	An open-source Rust linter developed by Trail of Bits to identify common code quality issues and mistakes in Rust code	Appendix C

Areas of Focus

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

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`

- Extensive use of unsafe code
- Poor unit and integration test coverage
- General issues with dependency management and known vulnerable dependencies

Test Results

The results of this focused testing are detailed below.

Cait-Sith. We ran several static analysis tools such as Clippy, Dylint, and Semgrep to identify potential code quality issues in the codebase. We additionally used Cargo plugins `cargo-edit`, `cargo-audit`, and `cargo-geiger` to review the dependency management practices and the use of unsafe Rust code. Finally, we used `cargo-llvm-cov` to review the unit and integration test coverage of Cait-Sith.

Property	Tool	Result
The project adheres to Rust best practices by fixing code quality issues reported by linters such as Clippy.	Clippy	**Passed**
The project does not contain any vulnerable code patterns identified by Dylint.	Dylint	**Passed**
The project's use of panicking functions such as `unwrap` and `expect` is limited.	Semgrep	**Appendix D**
The project contains a reasonable amount of unsafe code for what the implementation is trying to achieve.	`cargo-geiger`	**Passed**
To avoid technical debt, the project continuously updates dependencies as new versions are released.	`cargo-edit`	**Passed**
The project does not depend on any libraries with known vulnerabilities.	`cargo-audit`	**TOB-LPCS-12**

The project contains a reasonable number of tests including negative tests.	`cargo-llvm-cov`	TOB-LPCS-1 TOB-LPCS-9 TOB-LPCS-10

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 properly uses mathematical operations and semantics. In most cases, appropriate crates are used (e.g., for field arithmetic and elliptic curve operations). Participant IDs are unsigned 32-bit integers, and to avoid the zero-share problem, they are converted to unsigned 64-bit integers and incremented by one.	**Strong**
Auditing	No specific auditing is implemented besides standard error reporting from function returns. Consequently, library users may not have enough insight to investigate failure scenarios such as the one described in TOB-LPCS-4.	**Moderate**
Authentication / Access Controls	This library relies on the caller to ensure that all accesses and incoming messages are properly authenticated.	**Not Applicable**
Complexity Management	The implementation has a clear structure that closely follows the protocol specification. Each subprotocol has its own file with the relevant implementation, and common utility functions are contained in their own modules. However, the transcript generation for the Fiat-Shamir transformation is scattered across multiple files, and verifying that all required quantities are included is more complex as a result (see appendix D).	**Satisfactory**
Cryptography and Key Management	The implementation closely follows the specification, and the cryptographic primitives in use are sound. However, there is a discrepancy between the protocol and the implementation that leads to key recovery (TOB-LPCS-1). Moreover, a round-reduced variant of Keccak is used (i.e., KitTen, with only 10 rounds instead of 24). While this choice is based on a well-reasoned analysis, it means that	**Moderate**

Category	Summary	Result
	the implementation is less resistant to potential advances in the cryptanalysis of Keccak.	
Data Handling	In general, data provided by other participants is handled well, with various checks ensuring the correctness of the received data. There are a few instances where additional checks could be implemented for the purposes of defense in depth (TOB-LPCS-11).	**Satisfactory**
Documentation	The implementation is generally well documented, containing detailed references to corresponding steps in the specification. However, there are some minor issues and discrepancies. See appendix F for details. In terms of user-facing documentation, we found some issues that may lead to misuse of the library (TOB-LPCS-3, TOB-LPCS-8)	**Moderate**
Memory Safety and Error Handling	By relying on the core language concepts of Rust and avoiding unsafe code, the implementation provides strong memory safety guarantees. Error handling is properly performed in most places, although there are a few instances where the implementation may panic that could be handled. See appendix D for details.	**Satisfactory**
Testing and Verification	The implemented tests focus only on minimal verification of correct functional behavior for straightforward, positive test cases. Negative test cases are missing, and several issues found during this audit could have been detected with more extensive testing. Additionally, we recommend adding fuzz testing on all functions dealing with input provided by other protocol participants. Finally, the library contains several test helper functions that should be wrapped in `#[cfg(test)]`, as opposed to the current `#[allow(dead_code)]`, to prevent their compilation for production code.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Correlated-OT-Extension does not properly use session ID in PRG	Cryptography	High
2	Timing differences in hash_to_scalar implementation may disclose information to the sender in Random-OT-Extension	Cryptography	Medium
3	Insufficient warnings or safeguards against reusing presignatures and triples	Cryptography	High
4	Cait-Sith does not time out if progress is not made	Denial of Service	Informational
5	Sensitive data is not zeroized upon completion of subprotocols	Data Exposure	Medium
6	Protocol implementation tells the user to wait after completion	Error Reporting	Informational
7	Iterated extended oblivious transfer is not secure against a malicious receiver	Cryptography	High
8	Caller responsibilities around aborts are unclear	Denial of Service	Medium
9	Different participants in triple generation and triple setup causes deadlock	Data Validation	Informational
10	Requirements on thresholds are unclear and inconsistently verified in the implementation	Data Validation	Informational
11	The receiver in Batch-Random-OT does not check that Y is nonzero	Cryptography	Informational

12	Cait-Sith is implemented with outdated dependencies	Patching	**Informational**

Detailed Findings

1. Correlated-OT-Extension does not properly use session ID in PRG	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-LPCS-1
Target: `src/triples/bits.rs`	

Description

According to the specification for the Correlated-OT-Extension protocol, the PRG needs to be parameterized by a session ID. This is done in the `expand_transpose` function by initializing a `Meow` object:

```
pub fn expand_transpose(&self, sid: &[u8], rows: usize) -> BitMatrix {
    assert!(rows % SECURITY_PARAMETER == 0);

    let mut meow = Meow::new(PRG_CTX);
    meow.meta_ad(b"sid", false);
    meow.ad(sid, false);
```

Figure 1.1: Initialization of the `Meow` object (`cait-sith/src/triples/bits.rs:303–308`)

For each row, this `Meow` object must be cloned to ensure that the row uses the same prefix of the session ID. However, instead of being cloned, a new `Meow` object is created, dropping the session ID from the state:

```
// We need to clone to make each row use the same prefix.
let mut meow = Meow::new(PRG_CTX);
meow.meta_ad(b"row", false);
meow.ad(b"", false);
for u in row.0 {
    meow.ad(&u.to_le_bytes(), true);
}
meow.prf(&mut expanded, false);
```

Figure 1.2: A new `Meow` object is created instead of a clone. (`cait-sith/src/triples/bits.rs:317–324`)

As a result, all invocations of the PRG are unparameterized.

Exploit Scenario

The attacker is the participant in the protocol that acts as the sender in all pairwise Random-OT-Extension protocol executions. The attacker runs this protocol several times

for each receiver (eight times for a 256-bit curve and 128-bits of security) with different seeds s to generate different masking values χ_i. Since the same setup is used and the session ID is not used, the random matrix T_{ij} will be the same in all executions for a given receiver. Furthermore, the receiver provides check values t_j, which are a linear combination of the masking values χ_i and chunks of the columns of the matrix T_{ij}. As a result, the attacker can recover columns of T_{ij} from the receiver's check values t_j by Gaussian elimination. For any column where the sender's setup choice vector $\Delta_j = 1$, the sender can now retrospectively recover the receiver's choice vector $b_i = Q_{ij} - T_{ij}$.

Knowledge of this vector, which corresponds to the vector t_i in the Multiplicative to Additive conversion (MTA) protocol, allows the attacker to receive the receiver's secret share b in this protocol. Therefore, the attacker can learn both receiver shares in the multiplication protocol for all receivers. For all generated triples, the attacker can learn all values of the triple.

In the signing protocol, one of the triples corresponds to the ephemeral key or nonce. Given this ephemeral key and the corresponding signature, the attacker recovers the signing key. As this leads to recovery of the signing key, the severity is set to high. However, the difficulty is set to medium because the attacker is a participant of the protocol and still needs to implement the parts of the attack that recover the various shares.

Recommendations

Short term, fix this issue by cloning the `Meow` object. Also consider adding the participant IDs to the parameters of the PRG to localize it for each pair of participants within the same session.

Long term, add testing, such as independently generated test vectors, to help catch such issues. Another option is to add a negative test that verifies that the outcomes of protocol executions with two different session IDs but otherwise identical input are different. Such negative tests should be added in cases where it is critical for security that repetitions do not occur.

2. Timing differences in hash_to_scalar implementation may disclose information to the sender in Random-OT-Extension

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LPCS-2
Target: `src/triples/random_ot_extension.rs`	

Description

In the Random-OT-Extension protocol, the goal is for a sender to obtain κ pairs of field elements and a receiver to obtain exactly one of each of these pairs, based on a sequence of κ secret bits that the receiver generates as part of the protocol. At the end of the protocol, both parties use the `hash_to_scalar` function to derive the field elements, where the sender derives both field elements in the pair, and the receiver derives only the field element corresponding to their secret bit.

However, `hash_to_scalar` is implemented using rejection sampling. Therefore, depending on the input, there will be a variable number of rejections. Since the sender derives both field elements in a pair, they will know exactly how many rejections occur during the derivation of each field element in the pair. The receiver, on the other hand, will derive only one of the field elements in the pair. Consequently, if there are a different number of rejections depending on the bit, the receiver's execution time can tell the sender which of the pair's field elements is derived by the receiver. This would reveal the secret receiver bit to the sender.

```
loop {
    let mut field_bytes = FieldBytes::<C>::default();
    meow.prf(&mut field_bytes, true);
    // This will return None if the bytes we generated were not <
    // the modulus of the field.
    let maybe_scalar: Option<C::Scalar> =
        <C::Scalar as PrimeField>::from_repr(field_bytes).into();
    if let Some(done) = maybe_scalar {
        return done;
    }
}
```

Figure 2.1: Rejection sampling in `hash_to_scalar` (`cait-sith/src/triples/random_ot_extension.rs:32–42`)

In practice, these hash derivations are batched together in κ pairs, where κ is at least the bit size of the curve (e.g., 256) plus the security parameter (e.g., 128), times two (e.g., 768). It will be difficult to get much information about specific pairs from the batched timing of

receiver execution. If local timing information (e.g., from power or EM traces) is available, it might be possible to recover some of the receiver bits.

Exploit Scenario

As described in finding TOB-LPCS-1, if a participant can obtain the choice vector of all receivers, they can recover the signing key as well. However, it is unlikely that an attacker could recover the full choice vector of all receivers, even given very precise local timing information. Still, recovering the shares of even one other participant will allow an attacker to replace that participant in a signing ceremony, so the severity is set to medium. The difficulty is set to high due to the timing information required and the probability that sufficient information is disclosed from the timing side channel.

The probability of a rejection depends on the curve order. For example, for the `secp256k1` curve, the probability that a randomly generated bit string of the appropriate length is larger than the field prime is negligible, so this vulnerability is not practically exploitable for this curve.

Recommendations

Short term, consider documenting this vulnerability and warning the user not to use this implementation for curves where the rejection probability is non-negligible.

Long term, consider either implementing a wide reduction as part of the library, or consider using the `hash_to_scalar` function provided by the `elliptic_curve` crate to support generic hashing to a scalar.

3. Insufficient warnings or safeguards against reusing presignatures and triples

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-LPCS-3
Target: `src/sign.rs`, `docs/signing.md`	

Description

It is critical for the security of the ECDSA that presignatures are not reused. The specification in the `Orchestration.md` file contains a section on discarding information, which states the following about the output produced in each phase:

> *Then, this output is consumed by having a set of parties use it for a subsequent phase. It's critical that the output is then destroyed, so that no other group of parties attempts to re-use that output for another phase.*

The specification in the `Signing.md` file states that, "In the signing phase, a group of parties [...] consumes this presignature to sign a message *m*."

From either of these descriptions, it is unclear that the user should not reuse presignatures (and triples) even when the signature generation fails. Furthermore, there is no specific step in the specification that mandates destruction of presignatures.

In the implementation, the signing protocol's `sign` function takes ownership of the presignature. However, the `PresignOutput` struct implements the `Clone` trait (as does the `TripleShare` struct):

```
#[derive(Debug, Clone)]
pub struct PresignOutput<C: CSCurve> {
    /// The public nonce commitment.
    pub big_r: C::AffinePoint,
    /// Our share of the nonce value.
    pub k: C::Scalar,
    /// Our share of the sigma value.
    pub sigma: C::Scalar,
}
```

Figure 3.1: The `PresignOutput` struct implements the `Clone` trait (`cait-sith/src/presign.rs:18–26`)

Therefore, crate users could clone the presignature and use it in signing protocols until a signature is successfully produced, while still believing they follow the specification.

Exploit Scenario
Library users clone the presignature and use it in signing protocols until a signature is successfully produced.

One of the participants waits until step 2.3 of the first signing protocol execution for a specific presignature to obtain a valid signature and reports that the signature is invalid. If the protocol is now repeated with the same presignature but a different message, the malicious participant will have two valid ECDSA signatures on different messages with the same nonce, which allows them to recover the signing key. Reusing triples similarly leads to recovery of the signing key.

The severity of this issue is set to high because it leads to loss of the signing key. The difficulty is set to medium because it relies on a misunderstanding of the specification. Nevertheless, this sort of misunderstanding has occurred in practice more than once, which is why the specification and implementation should reduce the possibility of mistakes and incorrect use as much as possible.

Recommendations
Short term, update the specification with explicit steps that mandate destruction of the presignature and triples (both in cases of success and failure) and remove the `Clone` trait from the `PresignOutput`, `TripleShare`, and `PresignArguments` structs.

Long term, update the implementation to destroy the presignature and triple information.

4. Cait-Sith does not time out if progress is not made	
Severity: **Informational**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-LPCS-4
Target: `src/keyshare.rs`	

Description

The key sharing protocol uses an additive secret sharing scheme, so all participants are required to generate a shared secret and corresponding public key. Consequently, without a set timeout, the other participants will indefinitely wait on a faulty participant and stall the protocol when a participant does not send shares.

```
while !all_commitments.full() {
    let (from, commitment) = chan.recv(wait0).await?;
    all_commitments.put(from, commitment);
}
```

Figure 4.1: Participants indefinitely waiting in the key sharing protocol (`cait-sith/src/keyshare.rs:52–55`)

Cait-Sith does not aim to provide identifiable aborts where a faulty participant can be identified and, if needed, excluded from a new run of the protocol. However, it is advisable to provide a timeout mechanism so that online participants do not stall and resources are not needlessly wasted.

The issue does not lead to the loss of a private key or a signature forgery, so the severity is set to only informational. However, a single malicious party can easily halt the protocol by withholding a protocol message, so the difficulty is set to low.

Recommendations

Short term, clarify the responsibilities for timeouts. Update the specification with those considerations and explicitly mention whether the calling application should handle timeouts. Furthermore, consider adding monitoring capabilities so that the calling applications can decide how to handle a faulty participant in an eventual rerun of the protocol.

Long term, add a section to the specification documenting all responsibilities of the calling applications.

5. Sensitive data is not zeroized upon completion of subprotocols	
Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-LPCS-5
Target: `src/keyshare.rs`	

Description
Many subprotocols in Cait-Sith generate temporary secret data that is not deleted upon completion of the subprotocol. For instance, in the resharing protocol, old shares are not necessarily deleted from memory.

As described in finding TOB-LPCS-3, there is currently no mechanism to ensure that presignature values are destroyed upon completion of the protocol, even if the signature generation fails. However, even upon a successful signature generation, the values will remain in deallocated memory due to the lack of zeroization. The same consideration applies to triples.

Exploit Scenario
Protocol users run the resharing protocol to increase the threshold. An attacker gains access to the deallocated memory that still contains old shares of several participants. The number of participants for whom memory was recovered matches the previous threshold, so the attacker reconstructs the secret even though the protocol now uses a higher threshold.

The severity is set to medium, given the consequences when the attack is mounted. However, the difficulty is set to high since the attack requires memory access for several users. Nevertheless, following a more conservative approach to managing secret data is recommended.

Recommendations
Short term, consider zeroing sensitive values appropriately when they are not needed anymore.

Long term, consider implementing the `zeroize` trait wherever appropriate for customer types so that sensitive values will be zeroized when dropped.

6. Protocol implementation tells the user to wait after completion

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-LPCS-6
Target: `src/protocol/internal.rs`	

Description

The Cait-Sith implementation has a streamlined interface with a `poke` function to make progress and a `message` function to provide messages received from other parties. The `poke` function returns either an `Action`, telling the user what needs to be done, or a `ProtocolError`, telling the user that the protocol was aborted. The project `README` describes the options for the `Action`: The protocol has finished, a message should be sent to all parties or privately to one party, or no more progress can be made until the protocol receives new messages.

The first of these options is represented by `Action::Return(T)` in the implementation, whereas the last option is represented by `Action::Wait`. The `poke` interface is implemented for the `ProtocolExecutor` type, which has an internal flag stating whether the protocol is finished. This flag is set whenever the `ProtocolExecutor` obtains a return value or an error, after which it respectively responds with `Action::Return(T)` or `ProtocolError`. However, any time the user calls `poke` after this point, the `ProtocolExecutor` will return `Action::Wait`.

```
fn poke(&mut self) -> Result<Action<Self::Output>, ProtocolError> {
    if self.done {
        return Ok(Action::Wait);
    }
```

Figure 6.1: The protocol implementation tells the caller to wait if it is done. (`cait-sith/src/protocol/internal.rs:497–500`)

This is confusing and contradicts the meaning of `Action::Wait` as described by the `README`. The issue does not impact security, so the severity is set to informational. The difficulty is set to high because it relies on a user incorrectly processing the return value of the API.

Recommendations

Document the meaning of `Action::Wait` more accurately in the `README`, or consider adding another option—such as `Action::Done`—to the `Action` enum for the return value of the `ProtocolExecutor` when the protocol is finished.

7. Iterated extended oblivious transfer is not secure against a malicious receiver

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-LPCS-7
Target: `docs/triples.md`	

Description

According to the specification for the Correlated-OT-Extension protocol, the same setup can be used for multiple extensions. In the setup, the party that acts as the sender in Correlated-OT-Extension generates a bitvector Δ_j that corresponds to its choices in the Batch-Random-OT protocol. As part of the Random-OT-Extension protocol, the receiver provides check values x and t_j. The sender computes their own check value q_j, checks whether $q_j = t_j + \Delta_j \cdot x$, and aborts if this is not the case. The goal of these checks is to verify whether the receiver has correctly computed the matrix U_{ij} during the Correlated-OT-Extension protocol.

However, if the receiver has cheated in column j by flipping some bits and provides otherwise correct check values x and t_j, the sender will detect the cheating if and only if $\Delta_j = 1$. As a result, a malicious receiver can learn Δ_j by observing whether the sender aborts.

Exploit Scenario

The attacker is the participant in the protocol that acts as the receiver in all pairwise Random-OT-Extension protocol executions. The attacker runs this protocol 128 times (equal to the security parameter) for each sender, flipping a bit in a different column j of matrix U_{ij} in the underlying Correlated-OT-Extension protocol each time. By observing whether each sender aborts—which happens half the time on average—the receiver recovers the choice vector.

Knowledge of this vector allows the receiver to retrospectively compute the sender matrix Q_{ij} and both values v_i^0 and v_i^1 from any run of the protocol where no sender aborted. These latter two values correspond to the blinding values that the sender uses in the MTA protocol. This allows the attacker to receive the sender's secret share a in this protocol. Consequently, the attacker learns both sender shares in the multiplication protocol for all senders. Therefore, for all generated triples, the attacker learns all values of the triple.

In the signing protocol, one of the triples corresponds to the ephemeral key or nonce. Given this ephemeral key and the corresponding signature, the attacker recovers the signing key. Due to recovery of the signing key, the severity is set to high.

The difficulty is set to medium because the attacker is a participant of the protocol and still needs to implement the parts of the attack that recover the various shares. Furthermore, the attacker must distinguish exactly which senders aborted, which—depending on the consensus mechanisms—may require the attacker to spread out the cheating against different senders over different executions of the protocol. This attack further assumes that the malicious receiver will not be punished, as the protocol does not aim to provide identifiable aborts, and the receiver can spread their cheating over an arbitrary number of protocol executions.

Recommendations

Short term, instruct the user to discard triple setup information when a single sender aborts during the Random-OT-Extension protocol.

Long term, remain vigilant when using protocols to ensure that their use is covered by the corresponding security proof.

8. Caller responsibilities around aborts are unclear	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-LPCS-8
Target: `src/protocol/internal.rs`	

Description

The Cait-Sith security proofs specify that participants should notify all other participants if they abort and that those participants should subsequently abort (and notify all other participants, etc.). However, the implementation currently lets the caller know only that an error occurred, and it is up to the caller to create consensus among all participants that the protocol should be aborted.

Neither the specification nor the implementation provides sufficient information for a caller to understand their responsibilities in this case. They mention a lack of identifiable aborts but do not mention the general issue of consensus around aborts. Several of the subprotocols have assertions at the end without any further communication rounds, which means that some participants may consider the subprotocol successful, even when others aborted. Such differences in views can lead to denial-of-service (DoS) attacks, such as the example from the paper "Attacking Threshold Wallets."

Exploit Scenario

A malicious participant in the key refresh protocol chooses a subset of participants and sends bad shares to those participants and good shares to the other participants. The participants that receive the good shares successfully finish the protocol and delete their old shares, whereas the participants that receive the bad shares abort.

Depending on the number of participants and the threshold, it may be possible to end up in a scenario where there are not enough shares to reconstruct the private key, which means the private key is effectively destroyed.

The severity of this attack is set to medium because an unavailable private key could pose financial risks, depending on the use case of the protocol. The difficulty is also set to medium because the exploitability relies on the implementation of the user. Nevertheless, such attacks have been discovered by auditing real systems, which is why the specification and implementation should reduce the possibility of insecure use as much as possible.

Recommendations

Short term, update the implementation API description with the caller's responsibilities for aborts.

Long term, update the specification with all relevant consensus aspects that are not covered by the protocol and implementation, including but not limited to aborts, timeouts, and the message to be signed.

9. Different participants in triple generation and triple setup causes deadlock

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-LPCS-9
Target: `src/triples/multiplication.rs`	

Description

As part of the triple generation protocol, all pairs of participants must execute the multiplication protocol based on the output of a previously performed triple setup protocol. However, the multiplication implementation does not take the participants of triple generation as input, but rather takes the list of participants directly from the triple setup output.

```
pub async fn multiplication<C: CSCurve>(
    ctx: Context<'_>,
    sid: Digest,
    me: Participant,
    setup: Setup,
    a_i: C::Scalar,
    b_i: C::Scalar,
) -> Result<C::Scalar, ProtocolError> {
    let mut tasks = Vec::with_capacity(setup.setups.len());
    for (p, single_setup) in setup.setups.into_iter() {
```

Figure 9.1: Multiplication takes the list of participants from the `Setup` struct. (`cait-sith/src/triples/multiplication.rs:88–97`)

As a result, running triple generation with a strict subset of the triple setup participants will cause a deadlock, as all participants will be waiting for messages from others who are not participating in the protocol. The fact that a user needs to remove all nonparticipants from the `Setup` struct is not documented for the triple generation API. Furthermore, the triple generation implementation will verify that all participants are present in the setup, but not whether it contains nonparticipants.

As an aside, the multiplication protocol consumes the `Setup` struct by calling the `into_iter` function. This means that a user must clone the `Setup` struct when calling triple generation if they want to reuse it later, which is not documented. The issue does not impact security, so the severity is set to informational. The difficulty is set to low because users could plausibly trigger this issue due to the lack of documentation.

Recommendations

Short term, choose one of the following options:

- Update the implementation of triple generation to clone the `Setup` struct and remove nonparticipants.
- Update the documentation to instruct the user to do the above when performing triple generation.

Long term, add test cases where a strict subset of participants from previous protocols participates. To detect protocol deadlocks during testing, update the `run_protocol` and `run_two_party_protocol` test helpers to return an error if all parties obtain `Action::Wait` from `poke`.

10. Requirements on thresholds are unclear and inconsistently verified in the implementation

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LPCS-10
Target: `docs/orchestration.md`, `docs/signing.md`, `src/presign.rs`, `src/sign.rs`, `src/keyshare.rs`, `src/triples/generation.rs`, `src/triples/triple_setup.rs`	

Description

The `Orchestration.md` specification considers four different phases: key generation, triple generation, presigning, and signing. It states that the thresholds for each phase can be different, under the single constraint that each threshold must be at least as large as the threshold for key generation. This is not correct from a functional perspective, as triple generation can be performed without knowledge of the key (and therefore its threshold), and a resulting triple with any threshold could be used in a presigning protocol, resulting in a valid presignature.

These statements also contradict the signing specification, which states that the threshold of the private key and the threshold of both triples must be the same value in the presigning protocol to create shares of a presignature with the same threshold. This is also incorrect from a functional perspective, as different thresholds for the private key and triples could be used, and the resulting threshold would be the maximum of all underlying thresholds.

The implementation further confuses matters by implementing some checks on the thresholds but not others. The `KeyGenOutput` struct does not contain the threshold, which means that the private key threshold cannot be (and is not) verified by the presigning implementation. Instead, the presigning implementation verifies that the `threshold` provided as a parameter is equal to the `threshold` of the `TriplePub` structs for both input triples.

Consider the case where the threshold for the triples is lower than that of the private key. Now, if the number of participants is lower than the private key threshold but sufficient for the triples, the implementation will consider this a valid protocol as long as the provided threshold parameter value equals the threshold of the triples. However, executing this protocol will fail at step 2.6 (because the private key cannot be reconstructed) and provide the error "received incorrect shares of additive triple phase," which does not correctly reflect the source of the issue.

As an aside, the specification further mandates that all participants verify whether participants in presigning were also present for both triple generation executions, but the implementation omits this "because presumably they need to have been present in order to have shares." However, a participant that was not present for the triples can still execute this protocol because they can recover the share they should have contributed from the shares of the other parties. This will not be detected unless this participant was required to meet the threshold for the triples.

In general, the checks that lead to an `InitializationError` are implemented inconsistently, with several protocols not verifying whether the current participant, `me`, is part of the participant set. While it is unlikely that any of these inconsistencies will lead to security issues, there is a possibility for issues to arise if users do not pay attention to the thresholds.

Exploit Scenario

A set of participants that meets the private key threshold inadvertently performs triple generation and generates the corresponding presignature with a lower threshold. If this presignature is subsequently used for signing, any subset of participants that meets the lower threshold can recover the ephemeral key and therefore the signing key.

As it is highly unlikely that a set of users would make the exact mistakes required to trigger this issue, the severity of this issue is set to informational and the difficulty is set to high.

Recommendations

Short term, implement the following changes:

- Add the threshold to the `KeyGenOutput` struct.
- Compare the thresholds of the `KeyGenOutput` and `TriplePub` structs and continue only if the threshold for the triples is at least as high as that of the private key.
- Add a threshold to the `PresignOutput` struct that is equal to the maximum of the `KeyGenOutput` and `TriplePub` thresholds used to construct the presignature.
- Verify this threshold against the number of participants during signing to prevent burning a presignature that will not lead to a successful signature.
- Add the set of participants to all output structs and check that the specified subset relations hold during initialization of all protocols.
- Harmonize the initialization checks across all protocol APIs to reduce the number of doomed protocol executions.

Long term, add both positive and negative test cases for various combinations of thresholds and participants to confirm that the implementation matches the expectations of the specification.

11. The receiver in Batch-Random-OT does not check that Y is nonzero

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-LPCS-11
Target: `src/triples/batch_random_ot.rs`	

Description

Cait-Sith uses the Simplest OT protocol of Chou and Orlandi as the basis for batched random oblivious transfer. This protocol is a tweak to the Diffie-Hellman (DH) key exchange protocol. A deviating sender may send the point at infinity in all instances of the Batch-Random-OT protocol and force the resulting shared secret to also be the point at infinity.

However, the sender does not learn the choice vector of the receiver. Therefore, it does not seem immediate that the sender can cause much damage in the subsequent protocols. In fact, the sender will disclose its own choice vector in the subsequent OT extension protocols as a result. Consequently, the severity is set to informational and the difficulty is set to high. However, as a defense-in-depth strategy, the protocol could ensure that the sender sends a nonzero element to safeguard against potential randomness generation failure that produces a zero element.

Recommendations

Short term, implement a check to ensure that the point received by the receiver is not the point at infinity. Furthermore, consider adding the participant IDs as input to the `Meow` hash to localize the hash for each pair of participants.

Long term, consider the impact of receiving the point at infinity for all places in the specification where an elliptic curve point is received.

12. Cait-Sith is implemented with outdated dependencies

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-LPCS-12
Target: Cait-Sith	

Description
The current implementation of Cait-Sith relies on the outdated dependencies shown below:

Dependency	Version	ID	Description
`ansi_term`	0.12.1	RUSTSEC-2021-0139	Unmaintained
`atty`	0.2.14	RUSTSEC-2021-0145	Potentially unaligned read

Table 12.1: Outdated dependencies

Neither dependency poses a direct threat to Cait-Sith, but an unmaintained dependency may hide vulnerabilities that could pose a threat to Cait-Sith in the future. Since these outdated dependencies are not immediately exploitable, the severity is set to informational and the difficulty is set to high.

Recommendations
Short term, update the outdated dependencies to ensure the code is not affected by any potential vulnerability.

Long term, run `cargo-audit` as part of the CI/CD pipeline and ensure that the developers are alerted to any vulnerable dependencies that are detected.

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
Data Handling	The safe handling of user inputs and data processed by the system
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. 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 C.1: Metadata required to run Dylint

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

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a rule set hosted on the Semgrep registry. Trail of Bits' public ruleset can be used by running `semgrep --config "p/trailofbits"`.

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-geiger

The `cargo-geiger` Cargo plugin provides statistics on the use of unsafe code in the project and its dependencies. The plugin can be installed using `cargo install cargo-geiger`. To run the tool, run `cargo geiger` in the crate root directory.

cargo-edit

The `cargo-edit` Cargo plugin identifies project dependencies with newer versions available. The plugin is installed by running `cargo install cargo-edit`. To run the tool, run `cargo upgrade --dry-run` 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.

D. Code Quality Recommendations

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

- **Rename the `commit` function in the `math.rs` file to reflect that it is not the actual commitment scheme used in the protocol.** In Cait-Sith, committing to a polynomial is done with a (randomized) hash-based commitment scheme that provides hiding guarantees.

```
pub fn commit(&self) -> GroupPolynomial<C> {
    let coefficients = self
        .coefficients
        .iter()
        .map(|x| C::ProjectivePoint::generator() * x)
        .collect();
    GroupPolynomial { coefficients }
```

Figure D.1: `commit` computes the group polynomial. (`cait-sith/src/math.rs:94–101`)

- **Introduce new functions in the `ck-meow` crate to distinguish between a hash computation and a PRF computation.** The readability of `ck-meow` would be improved by adding two different functions for PRF computation and hashing. Currently, the `prf` method is used to do both, so the expected result is unclear on the first read. The example in figure D.2 does not compute a PRF (i.e., there is no key involved, and the behavior is easily distinguished from that of a random function). Though the naming is inherited from Strobe, where PRF refers to both PRF computation and hashing, it would be better to improve the readability of the code by introducing a new function to indicate that only a hash is performed and not a PRF computation.

```
fn root_shared() -> Self {
    let mut out = [0u8; Self::SIZE];
    let mut meow = Meow::new(MEOW_DOMAIN);
    meow.meta_ad(b"root shared", false);
    meow.prf(&mut out, false);
    Self(out)
}
```

Figure D.2: `meow.prf` computes channel tags with a hash function and not a keyed PRF. (`cait-sith/src/protocol/internal.rs:73–79`)

- **Centralize the Fiat-Shamir computation at a single location to improve readability.** The Fiat-Shamir transformation used for zero-knowledge protocols constructs the transcript across many files. From looking at the `prove` function

below, it is not immediately clear whether the caller of the prove function included all context information required to prevent attacks.

```
pub fn prove<'a, C: CSCurve>(
    rng: &mut impl CryptoRngCore,
    transcript: &mut Transcript,
    statement: Statement<'a, C>,
    witness: Witness<'a, C>,
) -> Proof<C> {
    transcript.message(STATEMENT_LABEL, &encode(&statement));
```

Figure D.3: Fiat-Shamir transform (cait-sith/src/proofs/dlog.rs:61–67)

- **Use a consistent encoding of numeric values.** The implementation encodes numerical values using a mix of little-endian and big-endian encoding.
- **Use the effective byte length of a `Waitpoint` type to compute the byte length of the `MessageHeader` header**. The byte length of wait points is hard coded in the computation of a message header instead of being taken for the `Waitpoint`.

```
const LEN: usize = ChannelTag::SIZE + 8;
```

Figure D.4: The message header size hard codes the size of `MessageHeader`. (cait-sith/src/protocol/internal.rs#134)

- **Change the `phi` instance method in the `dlog.rs` and `dlogeq.rs` files to a static method.**

```
fn phi(&self, x: &C::Scalar) -> C::ProjectivePoint {
    C::ProjectivePoint::generator() * x
}
```

Figure D.5: The `phi` method computing the homomorphism (cait-sith/src/proofs/dlog.rs#29–31)

- **Avoid panics in functions that return a `Result` type**. Several inner function calls, such as `unwrap`, can potentially panic, although the outer function returns a `Result`. Similarly, the signature verification function that is used as part of the signing protocol uses `unwrap` on the inversion of s, whereas it could return `false` if this inversion has no result.

```
pub async fn batch_random_ot_sender<C: CSCurve>(
    ctx: Context<'_>,
    mut chan: PrivateChannel,
) -> Result<BatchRandomOTOutputSender, ProtocolError> {
    [....]
    let big_k0: BitMatrix = out.iter().map(|r| r.0).collect();
```

```
    let big_k1: BitMatrix = out.iter().map(|r| r.1).collect();
    Ok((big_k0.try_into().unwrap(), big_k1.try_into().unwrap()))
```

Figure D.6: Potential panic in a function that returns a `Result` (`cait-sith/src/triples/batch_random_ot.rs#41–72`)

- **Verify the length given to the `Vec::with_capacity()` function to prevent panics in certain circumstances**. The `Vec::with_capacity()` function is used when creating a `ParticipantMap` type, when creating a `Polynomial` type using the `extend_random` function, and when generating shares using the `deal` function. Depending on the size of the vector elements, this function will panic when the capacity (measured in bytes) exceeds `isize::MAX` of the platform (i.e., $2^{31} - 1$ for 32-bit systems).

  For example, for 256-bit curves, the second `ParticipantMap` in the `do_generation` function stores a 144-byte `ProjectivePoint` type for every participant. Therefore, it will panic on 32-bit systems if the number of participants is larger than $(2^{31} - 1)/144 \approx 2^{24}$. During protocol initialization, the number of participants should be checked against the correct proportion of `isize::MAX` to prevent panicking (which, for Cait-Sith, would happen only on 32-bit systems), even though it may be unlikely that the protocol will be run with this many participants.

E. Review of the Cait-Sith Security Proofs

Cait-Sith comes with security proofs that we reviewed during our engagement. This section contains our observations on the proofs, coverage limitations, and general recommendations.

Coverage

Due to the time-boxed nature of the engagement, our review focused on the high-level aspects of the proofs. In particular, we reviewed the Modular Protocol Security (MPS) framework at a high level to become familiar with its mechanics and its relationship to the Universal Composability (UC) framework. We did not investigate the soundness of MPS as a general framework or whether it fully instantiates the UC framework. Limitations of MPS are described in the MPS paper.

We verified whether the protocols and associated functionalities analyzed in MPS correctly capture Cait-Sith. We also verified whether the ideal protocols were sensible—that is, the ideal protocol implements the intended protocol in terms of correctness and security guarantees. For example, we verified that the ideal presigning protocol does not reveal more information than what is truly accessible to each party in the presigning protocol.

In the proofs, we verified at a high level the soundness of the game hops, which includes sound simulation of timing and abort characteristics, simulation of messages from honest to malicious parties, and extraction of effective inputs of the malicious users when needed. Finally, we checked that the advantage bounds in each hop were reasonably justified.

Verification of proofs is a long and demanding process. Unfortunately, many examples of proofs (one famous example being OCB2) have passed initial peer review, but errors were discovered many years later. As is standard for cryptography research, we recommend that the larger community examine the proofs (and the proof system) to increase confidence in their validity.

High-level observations and remarks

The MPS framework

The framework is built to enable modular security proofs whereby a simulator can be specified in multiple steps via presumably simple hops, as is the case for code-based games. In contrast to UC, protocols are more formally described via pseudocode, and the proofs enjoy a more modular presentation, making them potentially easier to follow.

The analysis of Cait-Sith is carried out in the MPS framework. For some hops in the proof, details are omitted for brevity and simply replaced by "...", which hurts readability. Because of the modularity, the final simulator is not fully described, although the soundness of the simulation can be deduced by the soundness of each step and the fact that the number of

steps is small. Finally, the security analysis is performed in the asymptotic setting. We believe that the protocol still enjoys good concrete bounds. However, the bounds in most hops are not explicitly stated, even when the hops are not between perfectly equivalent protocols.

Editorial issues and definition inconsistencies

Our review identified several issues in the proof due to typos or inaccurate definitions. We believe most issues can be easily fixed by fixing the typos or amending definitions. There are further issues with incorrect variable and function names. These are likely caused by the manual typesetting of the games, which can be error-prone and require thorough checking. For instance, some functions (e.g., `Presign`) call themselves due to naming collisions, creating the impression of an infinite loop. Even in cases where this is intentional because it corresponds to functionality calling other instances of functionality, it leads to greatly reduced clarity of the proof. The time required to parse and understand proofs increases significantly when the reader must consider which instance of a function is referenced for every function.

Detailed observations

In this section, we give further details on issues identified in the proof.

Synchronous communication functionality

- The definition of the functionality does not allow a party to send a message to itself, which is a sensible definition. However, many protocols in the proof assume that parties can send messages to themselves. A case where this happens is in the echo broadcast proof.

Zero-knowledge functionalities

- The functionalities for zero knowledge (ZK) are ill defined. Currently, the ZK functionalities are defined by an internal table mapping proofs to witnesses. Consequently, verifying a proof requires that the `Verify` functions take the proof and the witness as arguments, which does not accurately define a ZK protocol, where witness values are held private by the prover and not shared with the verifier.

Echo broadcast

- The input (x) to the `StartBroadcast` function is not saved in a state variable, so it is impossible for the confirmation hashes to be equal since parties do not send their inputs to themselves in the first round. Hence, the vector of inputs in the second round always excludes the party's input.

- The `WaitBroadcast` function should not take any input argument.

- Protocol hop 1: Honest users are not returning the vector x in `WaitBroadcast`.

- Protocol hop 2: In `WaitBroadcast`, parties resend a vector x (free variable) in round 0. The parties do not return the vector of elements received.

- Protocol hop 9: In `WaitBroadcast`, the parties are awaiting on $\hat{x}_{ij}$, rather than $\hat{x}_{ji}$.

Signing

- The numbering of rounds in the presigning protocol is inconsistent with the other proofs. The count starts at 1. Then in later protocol descriptions, the rounds start at 0.

- Parties in the presigning protocol never use the `Setup` functionality to get their key shares or the common public key.

- `Protocol` $\mathcal{P}$[`IdealPresign`]: The `Presign` function seemingly calls itself due to a naming collision.

- Presign Hop 1 ($\mathcal{P}^0$): The simulator for the first hop is only sketched and omits information about simulating the communication. Though these details are trivially filled in, the argumentation would be more complete if such details were briefly included.

- The simulation of the value b_k is incorrect.

- Signing protocol: The `Hash` functionality is not explicitly included in the signing protocol.

- The protocol $\mathcal{P}$[`IdealSign`] does not need to be composed with the protocol $\mathcal{P}$[`Presign`].

Key sharing

- The hop from $\mathcal{P}$[`Convert`] to $\mathcal{P}^0$ is missing an advantage bound induced by the need for collision resistance, which is 2^{λ}. The simulation is not perfect because replacing the protocol $\mathcal{P}$[`Commit`] with $\mathcal{P}$[`IdealCommit`] incurs a negligible distinguishing advantage due to collision resistance.

- In the game Γ^0, access to the `Commit` function is not given to the adversary, likely by omission.

- In the game Γ^1, all parties can arbitrarily modify values such as Z_{ij}, π_{ij} of the functionality. The seeming implication is that there are no restrictions on who can set these values, including the adversary.

- In the simulator $\mathcal{S}$ for conversion, the function $\texttt{Sample}_k$ is performing nontrivial operations on an already committed polynomial. It generates a random polynomial, commits to it, and then overwrites evaluations of the committed polynomial. To improve readability, state that Lagrange interpolation can be used to achieve this.

Multiplication and triples

- The description of the multiplicative-to-additive conversion solely as a functionality is confusing since the protocol $\mathcal{P}[\texttt{MTA}^2]$ is later defined by composition with $\mathcal{P}[\text{MTA}]^2$. Using a functionality instead of a protocol is inconsistent with the style of the other proofs. Although the Deidealization Lemma justifies the semantics, the proof would be more readable by directly specifying a dummy MTA protocol.

- In the functionality $F[\texttt{MTA}]$, the `Cheat` function is described as accessible to every party, even honest ones, rather than a function reserved for the adversary.

- The `MTA` functionality differs from the `WeakMult` functionality described in the paper Highly Efficient OT-Based Multiplication Protocols. Namely, the proof mentions that the `MTA` functionality is a simplification of `WeakMult`; however, a major difference is the restrictions `WeakMult` puts on the adversary when it comes to corrupting the parties' output. Concretely, it is unclear why the newly defined functionality simplifies the original one and gives the adversary more power. Any dishonest protocol participant is always capable of introducing an offset to the result by outputting an additive share with the same offset.

- In $\mathcal{P}[\texttt{MTA}^2]$ the $\texttt{Start}_{ij}$ and $\texttt{EndMTA}_{ij}$ functions have the incorrect subscript ij, which is also inconsistent with the simulator.

- The notation $\texttt{Flip}_{ij}(a, b)$ (and later $\texttt{Flip}_i(a, b)$) is confusing. It would be better to simply specify the order of inputs and use an `if` statement ($i < j$) to encode the order of the inputs.

- In the simulator for $F[\texttt{MTA}^2]$, all functions from the $F[\texttt{MTA}]$ functionality are improperly indexed. For example, `Start` should be $\texttt{Start}_2$

- The `Leak` function in $F[\texttt{MTA}^2]$ is unmotivated. It is not clear that the adversary may gain that leakage from the protocol. Furthermore, the same effect can be achieved by calling End_i, and if there's an output, a_1, $b_1$must have been set. Finally, the `Leak` function does not require Δ as an argument.

Recommendations

This section lists recommendations to improve the proof. In addition to resolving the issues mentioned in the Detailed observations section, implementing the following recommendations will improve the overall quality and readability of the proof.

The MPS framework

- Consider writing a monolithic simulator or consolidating a few steps into a single step when appropriate, which may simplify reviewing the proof. The proof may still enjoy modularity using a series of hybrid experiments in this case.
- Consider keeping details from hop to hop to increase readability, though functions that do not change may simply have the name without the details.
- Consider giving detailed advantage bounds for protocol hops.

Editorial issues and definition inconsistencies

- Thoroughly review the proof and fix typos.
- Review definitions of functionalities for consistency and correctness.
- Consider using macros to systematically write variables and function names.
- Consider a more systematic approach to naming variables and functions to prevent naming collisions inside games.

Synchronous communication functionality

- Allocate state variables to store the values any party needs for computing different functions.

Zero-knowledge functionalities

- The internal proof table should map proofs to valid statements. For instance, for a tuple witness and statement (x, X) where $X = xG$, the table Π should be a mapping of a proof π to X—that is, the `Prove(X; x)` function assigns $\Pi[\pi] \leftarrow X$.

F. Specification Review

This section enumerates observations on the specification of Cait-Sith. We list issues in the specification, minor issues in the implementation, and discrepancies between the specification and the implementation. Discrepancies between the proof of security and the specification are listed in appendix E. Most issues are minor typographical errors or clarity issues that do not necessarily affect security. They are listed here for completeness.

General

The following are comments on the specification:

- The symbols that are used to prefix some steps, such as triangle (assertion), star (broadcast in the KeyShare and signing protocols, private send in the triple generation protocol), red star (private send in KeyShare), circle (wait), square (return but only for triple generation), are not defined or explained, and star is not necessarily consistent between different pages.
- The notation with a circle in the subscript of a matrix or vector (denoting all elements indexed by that subscript) is not explained.
- When the specification indicates a computation, sometimes the symbol = is used for assignments (e.g., Correlated-OT-Extension step 3), but other times, the symbol ← is used.
- In the `proofs.md` file, the first sentence is incomplete: "Various protocols use ZK proofs using the Fiat-Shamir. These proofs need [...]".
- The `key-generation.md` page has a typo in the second paragraph: “particapant.”
- The specification does not explicitly discuss the care taken in Cait-Sith to prevent zero-share issues in subprotocols that rely on secret sharing. The implementation prevents zero-share issues by converting participants' `u32` IDs to `u64` values and increasing by one. The specification should explicitly discuss the proper implementation considerations to help potential alternative implementations of Cait-Sith avoid zero-share issues.

The following are comments on the implementation:

- Channel tags are set to 160 bits, offering 80 bits of collision resistance. Although this is more than enough for the application, it creates an inconsistency with the remaining codebase, where the overall security should be 128 bits.

Key generation

Key resharing

The following are comments on the specification:

- The set difference P'/P is using the wrong slash.
- The linearization coefficient is inaccurate, as it should be $\lambda(P \cap P')_i$, rather than $\lambda(P)_i$. The implementation performs the computation correctly.

The following are discrepancies between the implementation and the specification:

- The implementation has a check that $|P \cap P'| \geq t$, which is mentioned in the specification for key resharing but not as a specific step in the KeyShare protocol. Consider adding this to the specification as a check to be performed if $S \neq \perp$.
- The implementation also has a check that all participants in P verify that they have a share. Consider adding this to the specification.

Triple generation

Triple setup (Batch-Random-OT)

The following are discrepancies between the implementation and the specification:

- The implementation is missing annotation of steps 2 (lines 50–52), 7 (57–58), and 8 (60–63) in the `batch_random_ot_sender` function. Also, sometimes the code comments refer to "Spec X" and sometimes to "Step X."
- The implementation contains random generation of delta in `batch_random_ot_receiver` function, which is an input in the specification. Consider making this consistent (e.g., by adding this step to the specification). Also, consider swapping generation of delta with step 3 for efficiency since it can be done while waiting for Y. Furthermore, steps 6 and 5 are swapped between the specification and implementation. The implementation makes more sense because it will allow the sender to compute in parallel while the receiver performs hashing.
- In the implementation of the `hash` function in the `batch_random_ot.rs` file, i, X_i, and Y are added to the hash computation in that order, but the specification lists $H_{(i,Y,X_i)}$.

The following are comments on the implementation:

- In the implementation of `batch_random_ot_sender` and `batch_random_ot_receiver`, the variable name `wait0` is used for two different

wait points (lines 50–52 and 57–58 for the sender, and lines 82–84 and 97–99 for the receiver).

Correlated-OT-Extension

The following are discrepancies between the implementation and the specification:

- For completeness, annotate step 7 of Correlated-OT-Extension in lines 38 (for the sender) and 60 (for the receiver) in the `correlated_ot_extension.rs` file.

Random-OT-Extension

The following are comments on the specification:

- The notation in the specification of Random-OT-Extension is confusing:
  - The statement $X_{ij} \leftarrow b_i 1_j$ could be more clearly written as $X_{ij} \leftarrow b_{i'}$ or equivalently that X_{ij} is the matrix consisting of λ copies of the vector b_i as its columns.
  - In step 6, it is stated that adjacent bits are grouped, but it is not clear that for matrices, this means chunks of the column are grouped (as opposed to chunks of the row).
  - The keyword `mul` is supposed to be used for explicit multiplication in the field, but it is not used to show the field multiplication $\hat{b}_i \times \chi_i$ in step 9.
  - The inner product notation is difficult to parse, together with the `mul` keyword and 1_j vectors. Using the $\langle A_{ij}, x_j \rangle_i$ format would be clearer and more consistent with the `intro.md` file. Removing the `mul` keyword while stating that the component-wise multiplication is over the field would further improve readability. Alternatively, use a Σ summation (which would also help explain the relation $q_j = t_j + \Delta_j \cdot x$ to the reader).
  - Some additional explanatory comments would be helpful in this protocol. The protocol includes various calculations, and explaining why some equalities must hold becomes much easier if each step includes how the resulting value relates to other values— for example, $q_j \leftarrow \langle \hat{Q}_{ij}, \chi_i \rangle_i = \langle (\hat{T}_{ij} + \Delta_j \hat{b}_i), \chi_i \rangle_i = \langle \hat{T}_{ij}, \chi_i \rangle_i + \Delta_j \langle \hat{b}_i, \chi_i \rangle_i$.

The following are discrepancies between the implementation and the specification:

- The difference between κ versus κ' is not explained in the specification; and the computation of μ also differs between the specification and implementation. The implementation uses the `adjust_size` function to compute

$\kappa' = \kappa + \lambda - (\kappa \, mod \, \lambda) + 2 \cdot \lambda$ and, later, computes $\mu = \kappa'/\lambda$ where no rounding is needed. The specification mentions both a κ and a κ' but does not explain any relation between them and defines $\mu = \lceil \kappa'/\lambda \rceil$.

- The `random_ot_extension_sender` function contains a few issues:
    - The annotated step 5 in line 91 is actually step 3 in the specification.
    - In the implementation, the `seed` value is computed using a hard-coded length of 32 bytes in line 92, whereas this value should depend on the `SECURITY_PARAMETER` macro.
    - Lines 94–95 are step 4 in the specification but are not annotated.
    - The specification and implementation annotations around steps 10 and 11 are confusing. First, in the specification, step 11 combines a "wait for receive" (with a dot) with an `assert` statement (but only the word "check" is used, and the triangle is missing). It is unclear from the specification that there should be an abort if it does not match. In the implementation, step 11 annotates the waiting in lines 104–105, but not the check/computation in lines 122–126. Step 10 is annotated in line 107, whereas the length check in lines 108–112 is not specified, and the actual computation of q_j happens in lines 117–120. The implementation performs steps 10 and 11 out of order, as it computes q_j in a loop while checking, whereas this could be computed before waiting to improve the running time.
    - Step 14 in line 129 is actually step 12; there is no step 14 in the spec.
- The `random_ot_extension_receiver` function contains a few issues:
    - In the implementation, the `seed` value is received using a hard-coded length of 32 bytes in line 172, whereas this value should depend on the `SECURITY_PARAMETER` macro.
    - Step 11 in line 195 is actually step 9 in the specification.
    - Step 15 in line 199 is actually step 13; there is no step 15 in the spec.

MTA

The following are comments on the specification:

- Step 2 is inaccurate: $-a$ should be on the other side, such that $c_i^b = (-1)^b \cdot a + \delta_i + v_i^b$. The implementation is correct here.

- Step 4 has a typo: “S then sets” should be “R then sets.”
- Steps 5 and 6 can be swapped for efficiency.

The following are discrepancies between the implementation and the specification:

- The seed length is hard coded in the `mta.rs` file (lines 36 and 70), whereas it should depend on the `SECURITY_PARAMETER` macro.
- Step 8 of the MTA specification can be annotated in the code (on line 46), and it should include that R outputs β (which can be annotated on line 94).

Multiplication

The following are comments on the specification:

- There is a typo: κ is defined as $\lceil q \rceil + \lambda$, which should be $\lceil lg\ q \rceil + \lambda$.
- The variable names a and b are overloaded in the description, especially for step 2. Stating in step 2 that MTA is run using the input (a_i, b_j) to produce γ_j^0 for R and γ_i^0 for S (and then the same for (b_i, a_j)) would be clearer. Furthermore, indexing γ with i and j separately means that they are not uniquely defined because participant P_i will produce different values that are all denoted as γ_i^b for every participant $P_j > P_i$.
- Step 3 is not really a step (i.e., where some action is performed by either party). Instead, it is the output of the actions described in step 2. Consider describing it as a renaming of variables, as is done in step 1.2 of the presigning protocol.

The following is a discrepancy between the implementation and the specification:

- In the implementation, step 4 is not annotated (should be lines 116–119).

Triple generation

The following is a comment on the specification:

- There is a typo in step 3.9: P_I should be P_i (lowercase *i* instead of capital *I*).

The following are discrepancies between the implementation and the specification:

- It would be clearer to number all checks in step 3.4 individually (and to annotate all of them separately in the implementation).
- Some steps in the implementation are important but not easily linkable to the specification:

    - In line 243, all $E_j(0)$ values are collected as part of annotated steps 3.3, 3.4, 3.6, and 5.3, which is required for step 4.2 in the spec.
    - In line 368, c_i is initialized as $l(i)$ as part of annotated step 4.9, which is needed for annotated step 5.6 in line 415. (This is opposed to other cases, where the initialization typically happens as part of the step or just before.)

- The implementation annotation is incorrect in some places: "Spec 4.8" is step 4.7 in the specification, whereas "Spec 4.9" is step 4.8 (step 4.9 does not exist in the specification).
- Steps 5.8 (lines 425–427) and 5.9 (lines 429–442) should be annotated in the implementation for completeness.

Signing

Presigning

The following is a comment on the specification:

- It would be helpful to note in step 2.8 that σ_i is a threshold sharing of kx.

The following are discrepancies between the implementation and the specification:

- The implementation annotation is incorrect in some places: "Spec 1.9" is step 1.5 in the specification, and "Spec 1.10" is step 1.6.
- The specification lists some redundant steps: d_i and c'_i are not used, so it is unclear why they are multiplied by the Lagrange coefficient. The implementation does not include this computation.
- All variables get a prime when multiplied by the Lagrange coefficient, except d_i (not used) and kd_i. This is a bit confusing.

The following is a comment on the implementation:

- There is a typo in the presigning API documentation, which states “it's crucial that a presignature is never used.” It should instead state that the presignature is never “reused.”

Signing

The following is a comment on the specification:

- There is a typo in step 1.3: "M" should be "m."

The following is a discrepancy between the implementation and the specification:

- The `do_sign` function normalizes the value s, which is not specified as a step.

G. Key Derivation and Rerandomized Presignatures

Homogeneous Key Derivation

Lit Protocol is considering using Homogeneous Key Derivation, which is a kind of deterministic key derivation defined in the paper "On the security of ECDSA with additive key derivation and presignatures." The main benefit of this key derivation scheme is that many child key pairs can be derived from a single master key pair, which works for both the private and public components of the key pairs. The paper also provides a security proof showing that the security is equivalent to plain ECDSA without key derivation.

The master key pair comprises two ECDSA key pairs (i.e., the private part consists of two distinct ECDSA private keys d and d', and the public part consists of the corresponding ECDSA public keys $D = d \cdot G$ and $D' = d' \cdot G$). Child keys can now be derived using a (generally publicly known) tweak e by computing $\tilde{d} = d + e \cdot d'$ and $\tilde{D} = D + e \cdot D'$. This is amenable to the threshold setting, as the shares of $\tilde{d}$ can be computed from the shares of d and d' by each individual share owner, without any interaction, by computing $\tilde{d}_{(i)} = d_{(i)} + e \cdot d'_{(i)}$.

This approach prevents an attacker that obtains a single child private key from obtaining the master private keys or any other sibling private key. However, any attacker that obtains two distinct child private keys $\tilde{d}_1 = d + e_1 \cdot d'$ and $\tilde{d}_2 = d + e_2 \cdot d'$ for known tweaks e_1 and e_2 can recover the master private keys and hence all other sibling keys by computing the following:

$$d' = (\tilde{d}_1 - \tilde{d}_2)/(e_1 - e_2) \bmod q$$

$$d = \tilde{d}_1 - e_1 d' \bmod q$$

Therefore, it is recommended to immediately rotate the private master keys (and all corresponding child keys) if even a single child key is compromised.

Similarly, this approach will prevent attackers from recovering the keys if a nonce is reused across two different child keys. However, reusing the same nonce across three different child keys leads to the recovery of the private part of these child keys (and therefore, to the recovery of the master private keys as described above). The reason is that this nonce reuse creates a system of three linear equations with three unknowns (d, d', and nonce k), which can be solved by Gaussian elimination.

As a result, Lit Protocol is considering generating a larger number of root keys and deriving child keys from a tweak by multiplying each key by a different power of the tweak modulo

the group order. For example, for 10 keys d_0, d_1, ..., d_9 and tweak e, the corresponding child key would be computed as $\tilde{d} = d_0 + d_1 \cdot e + d_2 \cdot e^2 + ... + d_9 \cdot e^9 \bmod q$. The root keys are essentially used as coefficients of a polynomial that is evaluated at the tweak value. Given evaluations of this polynomial at 9 different tweaks, it is not possible to determine the evaluation at a different tweak.

This is because the evaluation at a different tweak could plausibly be any scalar in the group. Indeed, picking any scalar for the tenth evaluation point leads to a unique polynomial that can be computed using Lagrange interpolation. This means that an attacker would have to recover 10 child keys for different tweaks, or equivalently, that an attacker with nine child keys would not be able to recover the root keys or any other child keys. Therefore, the complexity for users increases linearly with the number of root keys, as does the complexity for an attacker recovering child keys.

Finally, adding root keys does not significantly affect the proof of Theorem 7 in the paper. Therefore, this scheme provides the same security guarantees against forgeries as homogeneous key derivation with two root keys.

Rerandomized presignatures

The same paper also describes a (theoretical) attack on ECDSA with presignatures and defines a mitigation against this attack called rerandomized presignatures. The following description of this attack and the mitigation uses the notation defined in the Cait-Sith specification.

The definition of a presignature in the paper is (R, k^{-1}), where $R = k^{-1}G$. Let r be the x-coordinate of R. An attacker with access to a presignature can compute $\tilde{R} = cR$, which has x-coordinate $\tilde{r}$. If the attacker can find two messages m and $\tilde{m}$ that hash to h and $\tilde{h}$, respectively, such that $\tilde{h} = h \cdot (\tilde{r}/r)$, and can request a signature (r, s) for m, then they can compute a forgery $(\tilde{r}, \tilde{s})$ on message $\tilde{m}$ by computing $\tilde{s} = (\tilde{r} \cdot s)/(r \cdot c)$.

This attack is considered theoretical because finding messages with corresponding hash values satisfying the required relation seems infeasible. However, it shows that using raw signing oracles with presignatures is unsafe, as it is trivial to compute the required h after choosing a single $\tilde{m}$ with corresponding $\tilde{h}$, given r and $\tilde{r}$.

As a mitigation, the paper proposes rerandomizing presignatures in the signing phase by generating an unpredictable δ after the message is fixed and updating the presignature to $(R', (k')^{-1}) = (R + \delta G, k^{-1} + \delta)$. Now, given the message, a signature can be produced by computing $s = k' \cdot (Hash(m) + r' \cdot x)$, where r' is the x-coordinate of R'. The paper further provides a security proof showing that this type of rerandomization results in a signature scheme with an equivalent security level to plain ECDSA.

The computation of the rerandomized value $k' = (k^{-1} + \delta)^{-1}$ using modular inversion happens during the signing stage using the definitions in the paper. However, in Cait-Sith, a presignature is of the form (R, k, xk), which means that the modular inversion happens during the presigning stage. Therefore, this additive rerandomization of the ephemeral key would be costly in the threshold context of Cait-Sith.

Instead, Lit Protocol is considering using multiplicative randomization, by defining $k' = \delta \cdot k$. This means that the presignature can be updated to $(\delta^{-1}R, \delta k, \delta xk)$, which is straightforward in the threshold setting as well. However, it is unclear whether a similar security proof can be given for the security of this proposal.

One significant difference between additive and multiplicative rerandomization is that in the additive case, the attacker will not know the discrete logarithm of R' with respect to R because $R' = k(k^{-1} + \delta)R = (1 + \delta k)R$. In the multiplicative case, the attacker knows that $R' = \delta^{-1}R$. This does not allow an attacker to perform the theoretical attack, however. The key insight is that the value r' is not revealed to the attacker before the message m or corresponding hash h is fixed. It seems hard for an attacker to compute a message $\tilde{m}$ with corresponding hash $\tilde{h}$ such that $\tilde{h} = h \cdot (\tilde{r}/r')$ since r' is not known until h is fixed. However, this observation does not constitute a proof of security. We recommend investigating whether the security proof of the paper can be modified to capture the multiplicative rerandomization case.

Lit Protocol is considering letting each participant contribute random values during the presigning phase and constructing δ during the signing phase by hashing these values together with the message and other context information. We recommend forcing the participants to commit to their random values and to wait for all commitments before broadcasting the values. Precommitted randomness prevents the class of attacks where the last participant to provide their random value can influence the result. Furthermore, the hash function should be used in a way to guarantee domain separation of the various inputs. Domain-separated hashes ensure that different inputs will not produce the same output.

H. 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 September 25 to September 27, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Lit Protocol team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 12 issues described in this report, Lit Protocol has resolved five issues, has partially resolved one issue, and has not resolved the remaining six issues. The supplied fixes address issues that pose the biggest risk to the safe use of Cait-Sith. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Correlated-OT-Extension does not properly use session ID in PRG	Resolved
2	Timing differences in hash_to_scalar implementation may disclose information to the sender in Random-OT-Extension	Resolved
3	Insufficient warnings or safeguards against reusing presignatures and triples	Unresolved
4	Cait-Sith does not time out if progress is not made	Unresolved
5	Sensitive data is not zeroized upon completion of subprotocols	Unresolved
6	Protocol implementation tells the user to wait after completion	Unresolved
7	Iterated extended oblivious transfer is not secure against a malicious receiver	Resolved
8	Caller responsibilities around aborts are unclear	Unresolved

9	Different participants in triple generation and triple setup causes deadlock	**Resolved**
10	Requirements on thresholds are unclear and inconsistently verified in the implementation	**Unresolved**
11	The receiver in Batch-Random-OT does not check that Y is nonzero	**Resolved**
12	Cait-Sith is implemented with outdated dependencies	**Partially Resolved**

Detailed Fix Review Results

TOB-LPCS-1: Correlated-OT-Extension does not properly use session ID in PRG
Resolved in commit `fa56723`. The `meow` object used to instantiate the PRG now includes the session ID in its state. Concretely, each invocation of the PRG in the protocol clones the initial state of the `meow` object, which includes the session ID.

TOB-LPCS-2: Timing differences in hash_to_scalar implementation may disclose information to the sender in Random-OT-Extension
Resolved in commit `bbede58`. The `hash_to_scalar` function no longer performs rejection sampling. Instead, the `CSCurve` trait requires that a compatible curve provides a function, `sample_scalar_constant_time`, which uses a (seed) PRG to generate a random field element. For the specific instantiation with the curve `Secp256k`, the `sample_scalar_constant_time` function performs a wide modular reduction on an initially sampled random 512-bit number.

TOB-LPCS-3: Insufficient warnings or safeguards against reusing presignatures and triples
Unresolved. This issue was not addressed in the fixes supplied by the Lit Protocol team.

TOB-LPCS-4: Cait-Sith does not time out if progress is not made
Unresolved. This issue was not addressed in the fixes supplied by the Lit Protocol team.

TOB-LPCS-5: Sensitive data is not zeroized upon completion of subprotocols
Unresolved. The Lit Protocol developers are working on ways to consistently delete sensitive data.

TOB-LPCS-6: Protocol implementation tells the user to wait after completion
Unresolved. This issue was not addressed in the fixes supplied by the Lit Protocol team.

TOB-LPCS-7: Iterated extended oblivious transfer is not secure against a malicious receiver
Resolved in commit `0d52f32`. The triple generation protocol has been reimplemented to no longer use a preexisting setup during triple generation. Instead, the protocol internally runs the Batch-Random-OT protocol to establish a fresh setup. Furthermore, all implementation details related to setups have been removed from the codebase by deleting the `triple_setup.rs` file.

TOB-LPCS-8: Caller responsibilities around aborts are unclear
Unresolved. This issue was not addressed in the fixes supplied by the Lit Protocol team.

TOB-LPCS-9: Different participants in triple generation and triple setup causes deadlock
Resolved in commit `0d52f32`. The triple generation protocol no longer uses any preexisting setup. Each run of the protocol triggers a new run of the Batch-Random-OT protocol.

Consequently, the multiplication function takes the current list of participants as a parameter, which is the same as the list of participants that participated in the setup. Therefore, the implementation prevents deadlock caused by a mismatch.

TOB-LPCS-10: Requirements on thresholds are unclear and inconsistently verified in the implementation
Unresolved. This issue was not addressed in the fixes supplied by the Lit Protocol team.

TOB-LPCS-11: The receiver in Batch-Random-OT does not check that Y is nonzero
Resolved in commit `46f175c`. The receiver in the Batch-Random-OT protocol now checks whether the Y point received is the identity element and returns an error if it is.

TOB-LPCS-12: Cait-Sith is implemented with outdated dependencies
Partially resolved in commit `ddf5288`. Several dependencies have been updated. However, Cait-Sith still uses a few outdated dependencies. Notably, the `criterion` and `structopt` crates have not been updated; consequently, the security advisories still need to be addressed.

I. 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.