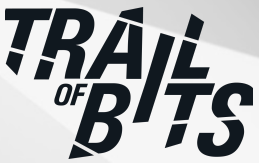

Succinct Labs Telepathy

Security Assessment

March 16, 2023

Prepared for:
Uma Roy, John Guibas
Succinct Labs

Prepared by: **Tjaden Hess, Marc Ilunga, and Joe Doyle**

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

Succinct Labs engaged Trail of Bits to review the security of its Telepathy Circom circuits and Solidity smart contracts. From January 9 to February 6, 2023, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and tests for the Circom circuits and Solidity smart contracts, as well as associated documentation and build scripts. We performed static and dynamic automated and manual testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The Telepathy smart contracts and Circom circuits are clear and well-constructed overall but lack sufficient unit testing and systematic documentation of internal subcircuit assumptions and postconditions. Using Circom's signal tags to uniformly handle value restrictions at compile time would significantly improve confidence in the soundness of the Circom circuits.

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	Count
High	5
Medium	1
Low	3
Informational	5

CATEGORY BREAKDOWN

Category	Count
Access Controls	1
Authentication	1
Cryptography	2
Data Validation	8
Error Reporting	1
Testing	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-SUCCINCT-1**
  During the sync committee rotation process, a malicious prover can supply ill-formed BigInts, corrupting the resulting commitment and preventing the `LightClient` from validating new block headers. The inability of the `LightClient` to progress may cause user funds on bridges relying on the `LightClient` block header data to become stuck.

- **TOB-SUCCINCT-2**
  During the sync committee rotation process, a malicious prover can supply valid BigInts that are not reduced modulo the BLS12-381 field prime, corrupting the resulting commitment and preventing the `LightClient` from making further progress. As a result, user funds on bridges relying on the `LightClient` block header data may become stuck.

- **TOB-SUCCINCT-7**
  When verifying transaction receipts in the `executeMessageFromLog` function, the `TargetAMB` does not verify that the provided Merkle proof is of the correct length. Malicious provers may therefore convince the contract that a value appearing elsewhere in the block Merkle tree is the transaction receipt root. As some of these

alternative values are potentially user-controlled, a malicious prover may be able to execute a transaction on the `TargetAMB` that was never sent to the `SourceAMB`, potentially stealing funds from the higher-level protocols relying on the bridge.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Joe Doyle, Consultant
joe.doyle@trailofbits.com

Project Timeline

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

Date	Event
January 05, 2023	Pre-project kickoff call
January 13, 2023	Status update meeting #1
January 20, 2023	Status update meeting #2
January 27, 2023	Status update meeting #3
February 07, 2023	Delivery of report draft
February 07, 2023	Report readout meeting
February 23, 2023	Delivery of fix review draft
March 16, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Succinct Labs Telepathy zero-knowledge ETH2.0 light client circuits and smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the Circom circuits sound in the presence of a malicious prover?
- Are arithmetic operations such as BigInt, field arithmetic, elliptic curve arithmetic, and bilinear pairings implemented correctly?
- Is the Poseidon-based commitment scheme to the sync committee cryptographically binding?
- Is the SHA-256-based public input commitment scheme cryptographically binding?
- Can the light client become stuck and unable to validate new block headers?
- Can the light client reflect an incorrect chain state under a ⅔ honest validator assumption?
- Can a malicious user execute a message on the target message bridge without a corresponding message on the source message bridge?

Project Targets

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

Telepathy Circom Circuits

Repository	https://github.com/succinctlabs/telepathy
Version	`3d6271a5c492cbb4d6be95d569983910d861fedc`
Type	Circom
Platform	Groth16 / R1CS

Telepathy Smart Contracts

Repository	https://github.com/succinctlabs/telepathy
Version	`b6a61ce90e748fc7a4a76cd0b04c0a8798db977d`
Type	Solidity
Platform	Ethereum

Project Coverage

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

- **Safety and correctness of Circom circuits.** We used static analysis tools on the Circom circuits to identify known vulnerabilities. We manually reviewed the circuits, paying particular attention to proper constraint specifications and arithmetic issues. Our manual review covered all Telepathy circuits and dependencies except Circomlib standard library templates. Review circuits include the BigInt, finite field, and elliptic curve arithmetic implementations, pairing computations, SHA-256, and Poseidon, as well as the top-level light client `Step` and `Rotate` circuits.
- **Security of SNARK-friendly commitments.** We reviewed the design and implementation of Poseidon-based commitments, focusing on whether they provide appropriate binding guarantees.
- **Smart contract security.** We manually reviewed the `LightClient`, `SourceAMB`, and `TargetAMB` implementations, in addition to analysis using Slither. We also reviewed the core Solidity contracts and the RLP and Merkle Patricia trie dependencies.
- **Overall ETH2.0 light client correctness.** We reviewed the light client implementations both in circuits and smart contracts in comparison to the ETH2.0 Beacon Chain light client/sync committee specification to ensure that the Telepathy protocol matches the security guarantees of the idealized light client protocol.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Front-end UI application
- Off-chain relayer, operator, monitoring, and SDK implementations
- Smart contract deployment procedures
- Example contracts in the `contracts/examples` directory
- Subgraph GraphQL manifests

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
Slither	A Solidity static analysis framework that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance code comprehension, and quickly prototype custom analyses.	Appendix D.1
Picus	Picus is a tool for checking output and witness determinacy of R1CS circuits using a combination of propagation and SMT solving.	Appendix D.2
Circomspect	Circomspect is a static analyzer and linter for the Circom programming language. It runs several analysis passes that can identify potential issues in Circom.	Appendix D.3

Areas of Focus

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

- Safety and reliability of Solidity smart contracts
- Determinacy of Circom sub-circuits
- Correctness and soundness of Circom circuits

Test Results

The results of this focused testing are detailed below.

Light client and message bridge contracts. We used Slither to detect common issues and antipatterns in the on-chain components of Telepathy.

Property	Tool	Result
Known security issues and unidiomatic code patterns	Slither	TOB-SUCCINCT-12

Circom circuits. We used Picus to check the determinacy of several subcircuits and Circomspect to check for best practices and common security issues.

Property	Tool	Result
Known security issues and unidiomatic code patterns	Circomspect	**Passed**
Strong and weak circuit determinacy	Picus	**Passed**

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	We discovered several issues caused by unsafe implementations of arithmetic functions. In particular, we found a lack of effective validation of BigInt and field element input in TOB-SUCCINCT-1, TOB-SUCCINCT-2, and TOB-SUCCINCT-14. We also identified mishandling of elliptic curve points at infinity in TOB-SUCCINCT-3 and TOB-SUCCINCT-4.	**Weak**
Auditing	The `LightClient` and Arbitrary Message Bridges emit events for critical changes, and the Telepathy repository contains monitoring tools to track `LightClient` updates and AMB messages.	**Satisfactory**
Authentication / Access Controls	Currently, the Telepathy protocol includes no privileged roles or user authentication mechanisms. However, some functionalities warrant additional restrictions, such as the update forcing described in TOB-SUCCINCT-8 and the transaction version allowlisting described in TOB-SUCCINCT-10.	**Moderate**
Complexity Management	The circuits and smart contracts are well-structured and laid out in a modular manner. While the circuits contain informative inline comments, there is no systematic documentation or validation of sub-circuit assumptions, making it easy to neglect or duplicate essential checks.	**Moderate**
Cryptography and Key Management	The application builds on sound cryptographic primitives like the BLS signature scheme and secure commitment schemes. However, the Poseidon permutation is misused	**Satisfactory**

	in a Merkle-Damgård construction rather than a sponge construction, contrary to the recommended usage (TOB-SUCCINCT-6).	
Decentralization	The Telepathy protocol has no privileged roles or centralized point of failure. However, the high cost of generating proofs for `LightClient` updates may prevent individual users from participating and reduce the overall number of provers. This high cost could lead to the centralization of denial-of-service (DoS) targets and extended liveness failures.	**Satisfactory**
Documentation	The light client implementation is accompanied by adequate documentation, including comments and technical specifications. However, several Circom templates do not explicitly state assumptions made on their inputs. As a consequence, it is challenging to know whether appropriate validation on inputs is missing or has already been performed elsewhere.	**Strong**
Front-Running Resistance	No specific front-running protections are present; however, there is currently no incentive to front-run transactions.	**Not Applicable**
Low-Level Manipulation	We found one instance (TOB-SUCCINCT-12) in which a low-level call does not properly handle recipients without code at the address.	**Moderate**
Testing and Verification	The Circom codebase currently lacks sufficient testing and provides only expensive end-to-end tests. Additional circuit unit tests would be beneficial (TOB-SUCCINCT-5).	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Prover can lock user funds by including ill-formed BigInts in public key commitment	Data Validation	High
2	Prover can lock user funds by supplying non-reduced Y values to G1BigIntToSignFlag	Data Validation	High
3	Incorrect handling of point doubling can allow signature forgery	Data Validation	High
4	EllipticCurveAdd mishandles points at infinity	Data Validation	Informational
5	Circom circuits lack adequate testing framework	Testing	Informational
6	Poseidon commitment uses a non-standard hash construction	Cryptography	Informational
7	Merkle root reconstruction is vulnerable to forgery via proofs of incorrect length	Cryptography	High
8	LightClient forced finalization could allow bad updates in case of a DoS	Access Controls	High
9	G1AddMany does not check for the point at infinity	Data Validation	Informational
10	TargetAMB receipt proof may behave unexpectedly on future transaction types	Data Validation	Informational
11	RLPReader library does not validate proper RLP encoding	Data Validation	Low

12	TargetAMB _executeMessage lacks contract existence checks	Error Reporting	**Low**
13	LightClient is unable to verify some block headers	Authentication	**Medium**
14	OptSimpleSWU2 Y-coordinate output is underconstrained	Data Validation	**Low**

Detailed Findings

1. Prover can lock user funds by including ill-formed BigInts in public key commitment

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SUCCINCT-1
Target: `circuits/circuits/rotate.circom`	

Description

The `Rotate` circuit does not check for the validity of BigInts included in `pubkeysBigIntY`. A malicious prover can lock user funds by carefully selecting malformed public keys and using the `Rotate` function, which will prevent future provers from using the default witness generator to make new proofs.

The `Rotate` circuit is designed to prove a translation between an SSZ commitment over a set of validator public keys produced by the Ethereum consensus protocol and a Poseidon commitment over an equivalent list. The SSZ commitment is over public keys serialized as 48-byte compressed BLS public keys, specifying an `X` coordinate and single sign bit, while the Poseidon commitment is over pairs `(X, Y)`, where `X` and `Y` are 7-limb, 55-bit BigInts. The prover specifies the `Y` coordinate for each public key as part of the witness; the `Rotate` circuit then uses `SubgroupCheckG1WithValidX` to constrain `Y` to be valid in the sense that `(X, Y)` is a point on the BLS12-381 elliptic curve.

However, `SubgroupCheckG1WithValidX` assumes that its input is a properly formed BigInt, with all limbs less than 2^{55}. This property is not validated anywhere in the `Rotate` circuit. By committing to a Poseidon root containing invalid BigInts, a malicious prover can prevent other provers from successfully proving a `Step` operation, bringing the light client to a halt and causing user funds to be stuck in the bridge.

Furthermore, the invalid elliptic curve points would then be usable in the `Step` circuit, where they are passed without validation to the `EllipticCurveAddUnequal` function. The behavior of this function on ill-formed inputs is not specified and could allow a malicious prover to forge `Step` proofs without a valid sync committee signature. Figure 1.1 shows where the untrusted `pubkeysBigIntY` value is passed to the `SubgroupCheckG1WithValidX` template.

```
/* VERIFY THAT THE WITNESSED Y-COORDINATES MAKE THE PUBKEYS LAY ON THE CURVE */
component isValidPoint[SYNC_COMMITTEE_SIZE];
for (var i = 0; i < SYNC_COMMITTEE_SIZE; i++) {
    isValidPoint[i] = SubgroupCheckG1WithValidX(N, K);
    for (var j = 0; j < K; j++) {
        isValidPoint[i].in[0][j] <== pubkeysBigIntX[i][j];
        isValidPoint[i].in[1][j] <== pubkeysBigIntY[i][j];
    }
}
```

Figure 1.1: `telepathy/circuits/circuits/rotate.circom#101–109`

Exploit Scenario

Alice, a malicious prover, uses a valid block header containing a sync committee update to generate a `Rotate` proof. Instead of using correctly formatted BigInts to represent the Y values of each public key point, she modifies the value by subtracting one from the most significant limb and adding 2^{55} to the second-most significant limb. She then posts the resulting proof to the `LightClient` contract via the rotate function, which updates the sync committee commitment to Alice's Poseidon commitment containing ill-formed Y coordinates. Future provers would then be unable to use the default witness generator to make new proofs, locking user funds in the bridge. Alice may be able to then exploit invalid assumptions in the `Step` circuit to forge `Step` proofs and steal bridge funds.

Recommendations

Short term, use a `Num2Bits` component to verify that each limb of the `pubkeysBigIntY` witness value is less than 2^{55}.

Long term, clearly document and validate the input assumptions of templates such as `SubgroupCheckG1WithValidX`. Consider adopting Circom signal tags to automate the checking of these assumptions.

2. Prover can lock user funds by supplying non-reduced Y values to G1BigIntToSignFlag

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SUCCINCT-2
Target: `circuits/circuits/rotate.circom`	

Description

The `G1BigIntToSignFlag` template does not check whether its input is a value properly reduced mod `p`. A malicious prover can lock user funds by carefully selecting malformed public keys and using the `Rotate` function, which will prevent future provers from using the default witness generator to make new proofs.

During the `Rotate` proof, when translating compressed public keys to full `(X, Y)` form, the prover must supply a Y value with a sign corresponding to the sign bit of the compressed public key. The circuit calculates the sign of Y by passing the Y coordinate (supplied by the prover and represented as a BigInt) to the `G1BigIntToSignFlag` component (figure 2.1). This component determines the sign of Y by checking if `2*Y >= p`. However, the correctness of this calculation depends on the Y value being less than p; otherwise, a positive, non-reduced value such as `p + 1` will be incorrectly interpreted as negative. A malicious prover could use this fact to commit to a non-reduced form of Y that differs in sign from the correct public key. This invalid commitment would prevent future provers from generating `Step` circuit proofs and thus halt the `LightClient`, trapping user funds in the `Bridge`.

```
template G1BigIntToSignFlag(N, K) {
    signal input in[K];
    signal output out;

    var P[K] = getBLS128381Prime();
    var LOG_K = log_ceil(K);
    component mul = BigMult(N, K);

    signal two[K];
    for (var i = 0; i < K; i++) {
        if (i == 0) {
            two[i] <== 2;
        } else {
            two[i] <== 0;
        }
    }
```

```
    for (var i = 0; i < K; i++) {
        mul.a[i] <== in[i];
        mul.b[i] <== two[i];
    }

    component lt = BigLessThan(N, K);
    for (var i = 0; i < K; i++) {
        lt.a[i] <== mul.out[i];
        lt.b[i] <== P[i];
    }

    out <== 1 - lt.out;
}
```

Figure 2.1: telepathy/circuits/circuits/bls.circom#197–226

Exploit Scenario

Alice, a malicious prover, uses a valid block header containing a sync committee update to generate a `Rotate` proof. When one of the new sync committee members' public key Y value has a negative sign, Alice substitutes it with `2P - Y`. This value is congruent to `-Y mod p`, and thus has positive sign; however, the `G1BigIntToSignFlag` component will determine that it has negative sign and validate the inclusion in the Poseidon commitment. Future provers will then be unable to generate proofs from this commitment since the committed public key set does not match the canonical sync committee.

Recommendations

Short term, constrain the `pubkeysBigIntY` values to be less than p using `BigLessThan`.

Long term, constrain all private witness values to be in canonical form before use. Consider adopting Circom signal tags to automate the checking of these assumptions.

3. Incorrect handling of point doubling can allow signature forgery

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCCINCT-3
Target: `telepathy/circuits/circuits/bls.circom`	

Description

When verifying the sync committee signature, individual public keys are aggregated into an overall public key by repeatedly calling `G1Add` in a tree structure. Due to the mishandling of elliptic curve point doublings, a minority of carefully selected public keys can cause the aggregation to result in an arbitrary, maliciously chosen public key, allowing signature forgeries and thus malicious light client updates.

When `bit1` and `bit2` of `G1Add` are both set, `G1Add` computes `out` by calling `EllipticCurveAddUnequal`:

```
template parallel G1Add(N, K) {
    var P[7] = getBLS128381Prime();

    signal input pubkey1[2][K];
    signal input pubkey2[2][K];
    signal input bit1;
    signal input bit2;

    /* COMPUTE BLS ADDITION */
    signal output out[2][K];
    signal output out_bit;
    out_bit <== bit1 + bit2 - bit1 * bit2;

    component adder = EllipticCurveAddUnequal(55, 7, P);
    for (var i = 0; i < 2; i++) {
        for (var j = 0; j < K; j++) {
            adder.a[i][j] <== pubkey1[i][j];
            adder.b[i][j] <== pubkey2[i][j];
        }
    }
```

Figure 3.1: telepathy/circuits/circuits/bls.circom#82–101

The results of `EllipticCurveAddUnequal` are constrained by equations that reduce to `0 = 0` if a and b are equal:

```
// constrain x_3 by CUBIC (x_1 + x_2 + x_3) * (x_2 - x_1)^2 - (y_2 - y_1)^2 = 0 mod
p

component dx_sq = BigMultShortLong(n, k, 2*n+LOGK+2); // 2k-1 registers abs val <
k*2^{2n}
component dy_sq = BigMultShortLong(n, k, 2*n+LOGK+2); // 2k-1 registers < k*2^{2n}
for(var i = 0; i < k; i++){
    dx_sq.a[i] <== b[0][i] - a[0][i];
    dx_sq.b[i] <== b[0][i] - a[0][i];

    dy_sq.a[i] <== b[1][i] - a[1][i];
    dy_sq.b[i] <== b[1][i] - a[1][i];
}
[...]
component cubic_mod = SignedCheckCarryModToZero(n, k, 4*n + LOGK3, p);
for(var i=0; i<k; i++)
    cubic_mod.in[i] <== cubic_red.out[i];
// END OF CONSTRAINING x3

// constrain y_3 by (y_1 + y_3) * (x_2 - x_1) = (y_2 - y_1)*(x_1 - x_3) mod p
component y_constraint = PointOnLine(n, k, p); // 2k-1 registers in [0, k*2^{2n+1})
for(var i = 0; i < k; i++)for(var j=0; j<2; j++){
    y_constraint.in[0][j][i] <== a[j][i];
    y_constraint.in[1][j][i] <== b[j][i];
    y_constraint.in[2][j][i] <== out[j][i];
}
// END OF CONSTRAINING y3
```

Figure 3.2: `telepathy/circuits/circuits/pairing/curve.circom#182–221`

If any inputs to `G1Add` are equal, a malicious prover can choose outputs to trigger this bug repeatedly and cause the signature to be checked with a public key of their choice. Each input to `G1Add` can be either a committee member public key or an intermediate aggregate key.

Exploit Scenario

Alice, a malicious prover, registers public keys `A = aG`, `B = bG` and `C = (a+b)G`. She waits for a sync committee selection in which `A`, `B` and `C` are all present and located in the same subtree of the sync committee participation array. By setting all other participation bits in that subtree to zero, Alice can force `EllipticCurveAddUnequal` to be called on `A+B` and `C`. Using the underconstrained point addition formula, she can prove that the sum of these points is equal to the next subtree root, and repeat until she has arbitrarily selected the aggregated public key. Alice can then forge a signature for an arbitrary block header and steal all user funds.

Recommendations

Short term, change `G1Add` to use `EllipticCurveAdd`, which correctly handles equal inputs.

Long term, review all uses of `EllipticCurveAddUnequal` to ensure that the inputs have different `X` components.

4. EllipticCurveAdd mishandles points at infinity

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCCINCT-4
Target: `circuits/circuits/{curve.circom, curve_fp2.circom}`	

Description

The `EllipticCurveAdd` and `EllipticCurveAddFp2` templates contain a logic bug when handling points at infinity, which can cause them to return incorrect results for specific sequences of elliptic curve point additions. The `EllipticCurveAdd` template is currently unused in the Telepathy codebase, while `EllipticCurveAddFp2` is used only in the context of cofactor clearing during the hash-to-curve process. Because the bug is triggered only in special sequences of operations, as described below, the random inputs generated by the hashing process are extremely unlikely to trigger the bug. However, if `EllipticCurveAdd` were to be used in the future (e.g., as we recommend in TOB-SUCCINCT-3), this bug may be exploitable using carefully chosen malicious inputs.

Figure 4.1 shows the logic, contained in `EllipticCurveAdd` and `EllipticCurveAddFp2`, that determines whether the point returned from `EllipticCurveAdd` is the point at infinity.

```
// If isInfinity = 1, replace `out` with `a` so if `a` was on curve, so is output

...

// out = 0 iff ( a = 0 AND b = 0 ) OR ( x_equal AND NOT y_equal )
signal ab0;
ab0 <== aIsInfinity * bIsInfinity;
signal anegb;
anegb <== x_equal.out - x_equal.out * y_equal.out;
isInfinity <== ab0 + anegb - ab0 * anegb; // OR gate
```

Figure 4.1: telepathy/circuits/circuits/pairing/curve.circom#344–349

When point `A` is the point at infinity, represented in projective coordinates as `(X, Y, 0)`, and `B` is the point `(X, -Y, 1)`, `EllipticCurveAdd` should return `B` unchanged and in particular should set `isInfinity` to zero. However, because the `x_equal AND NOT y_equal` clause is satisfied, the `EllipticCurveAdd` function returns `A` (a point at infinity).

This edge case is reachable by computing the sequence of additions `(A - A) - A`, for any point `A`, which will return `0` instead of the correct result of `-A`.

This edge case was first noted in a GitHub issue (#13) in the upstream yi-sun/circom-pairing library. We discovered that the edge case is in fact reachable via the sequence above and upstreamed a patch.

Exploit Scenario

The Succinct Labs developers modify the `G1Add` template as recommended in TOB-SUCCINCT-3. A malicious sync committee member then constructs public keys such that the `G1Add` process produces a sequence of additions that trigger the bug. Honest provers then cannot create proofs due to the miscomputed aggregate public key, causing funds to become stuck in the bridge.

Recommendations

Short term, update the pairing circuits to match the latest version of yi-sun/circom-pairing.

Long term, consider making the pairing and SHA-256 libraries npm dependencies or Git submodules so that developers can easily keep up to date with security fixes in the upstream libraries.

5. Circom circuits lack adequate testing framework

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-SUCCINCT-5
Target: `circuits/test`	

Description

The Telepathy Circom circuits do not have functioning unit tests or a systematic testing framework. Running the end-to-end circuit tests requires a large amount of RAM, disk space, and CPU time, and is infeasible on typical developer machines. Because it is difficult to rapidly develop and run tests, new code changes are likely to be insufficiently tested before deployment. The presence of a testing framework greatly aids security engineers, as it allows for the rapid adaptation of testing routines into security testing routines.

Recommendations

Short term, begin writing unit tests for each sub-circuit and requiring tests for all new code.

Long term, implement a framework for rapidly running all unit tests as well as end-to-end tests on scaled-down versions of the full circuits.

6. Poseidon commitment uses a non-standard hash construction

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SUCCINCT-6
Target: `circuits/circuits/poseidon.circom`	

Description

Telepathy commits to the set of sync committee public keys with a Poseidon-based hash function. This hash function uses a construction with poor theoretical properties. The hash is computed by `PoseidonFieldArray`, using a Merkle–Damgård construction with `circomlib`'s `Poseidon` template as the compression function:

```
template PoseidonFieldArray(LENGTH) {
    signal input in[LENGTH];
    signal output out;

    var POSEIDON_SIZE = 15;
    var NUM_HASHERS = (LENGTH \ POSEIDON_SIZE) + 1;
    component hashers[NUM_HASHERS];

    for (var i = 0; i < NUM_HASHERS; i++) {
        if (i > 0) {
            POSEIDON_SIZE = 16;
        }
        hashers[i] = Poseidon(POSEIDON_SIZE);
        for (var j = 0; j < 15; j++) {
            if (i * 15 + j >= LENGTH ) {
                hashers[i].inputs[j] <== 0;
            } else {
                hashers[i].inputs[j] <== in[i*15 + j];
            }
        }
        if (i > 0) {
            hashers[i].inputs[15] <== hashers[i- 1].out;
        }
    }

    out <== hashers[NUM_HASHERS-1].out;
}
```

Figure 6.1: `telepathy/circuits/circuits/poseidon.circom#25–51`

The Poseidon authors recommend using a sponge construction, which has better provable security properties than the MD construction. One could implement a sponge by using

PoseidonEx with nOuts = 1 for intermediate calls and nOuts = 2 for the final call. For each call, out[0] should be passed into the initialState of the next PoseidonEx component, and out[1] should be used for the final output. By maintaining out[0] as hidden capacity, the overall construction will closely approximate a pseudorandom function.

Although the MD construction offers sufficient protection against collision for the current commitment use case, hash functions constructed in this manner do not fully model random functions. Future uses of the PoseidonFieldArray circuit may expect stronger cryptographic properties, such as resistance to length extension.

Additionally, by utilizing the initialState input, as shown in figure 6.2, on each permutation call, 16 inputs can be compressed per template instantiation, as opposed to the current 15, without any additional cost per compression. This will reduce the number of compressions required and thus reduce the size of the circuit.

```
template PoseidonEx(nInputs, nOuts) {
    signal input inputs[nInputs];
    signal input initialState;
    signal output out[nOuts];
```

Figure 6.2: circomlib/circuits/poseidon.circom#67–70

Recommendations

Short term, convert PoseidonFieldArray to use a sponge construction, ensuring that out[0] is preserved as a hidden capacity value.

Long term, ensure that all hashing primitives are used in accordance with the published recommendations.

7. Merkle root reconstruction is vulnerable to forgery via proofs of incorrect length

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-SUCCINCT-7
Target: `contracts/src/libraries/SimpleSerialize.sol`	

Description

The `TargetAMB` contract accepts and verifies Merkle proofs that a particular smart contract event was issued in a particular Ethereum 2.0 beacon block. Because the proof validation depends on the length of the proof rather than the index of the value to be proved, Merkle proofs with invalid lengths can be used to mislead the verifier and forge proofs for nonexistent transactions.

The SSZ.`restoreMerkleRoot` function reconstructs a Merkle root from the user-supplied transaction receipt and Merkle proof; the light client then compares the root against the known-good value stored in the `LightClient` contract. The `index` argument to `restoreMerkleRoot` determines the specific location in the block state tree at which the `leaf` node is expected to be found. The arguments `leaf` and `branch` are supplied by the prover, while the `index` argument is calculated by the smart contract verifier.

```
function restoreMerkleRoot(bytes32 leaf, uint256 index, bytes32[] memory branch)
    internal
    pure
    returns (bytes32)
{
    bytes32 value = leaf;
    for (uint256 i = 0; i < branch.length; i++) {
        if ((index / (2 ** i)) % 2 == 1) {
            value = sha256(bytes.concat(branch[i], value));
        } else {
            value = sha256(bytes.concat(value, branch[i]));
        }
    }
    return value;
}
```

Figure 7.1: telepathy/contracts/src/libraries/SimpleSerialize.sol#24–38

A malicious user may supply a proof (i.e., a `branch` list) that is longer or shorter than the number of bits in the `index`. In this case, the `leaf` value will not in fact correspond to the `receiptRoot` but to some other value in the tree. In particular, the user can convince the

smart contract that `receiptRoot` is the value at any generalized index given by truncating the leftmost bits of the true index or by extending the index by arbitrarily many zeroes following the leading set bit.

If one of these alternative indexes contains data controllable by the user, who may for example be the block proposer, then the user can forge a proof for a transaction that did not occur and thus steal funds from bridges relying on the `TargetAMB`.

Exploit Scenario

Alice, a malicious ETH2.0 validator, encodes a fake transaction receipt hash encoding a deposit to a cross-chain bridge into the `graffiti` field of a `BeaconBlock`. She then waits for the block to be added to the `HistoricalBlocks` tree and further for the generalized index of the historical block to coincide with an allowable index for the Merkle tree reconstruction. She then calls `executeMessageFromLog` with the transaction receipt, allowing her to withdraw from the bridge based on a forged proof of deposit and steal funds.

Recommendations

Short term, rewrite `restoreMerkleRoot` to loop over the bits of `index`, e.g. with a while loop terminating when `index = 1`.

Long term, ensure that proof verification routines do not use control flow determined by untrusted input. The verification routine for each statement to be proven should treat all possible proofs uniformly.

8. LightClient forced finalization could allow bad updates in case of a DoS

Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-SUCCINCT-8
Target: `contracts/src/lightclient/LightClient.sol`	

Description

Under periods of delayed finality, the `LightClient` may finalize block headers with few validators participating. If the Telepathy provers were targeted by a denial-of-service (DoS) attack, this condition could be triggered and used by a malicious validator to take control of the `LightClient` and finalize malicious block headers.

The `LightClient` contract typically considers a block header to be finalized if it is associated with a proof that more than two-thirds of sync committee participants have signed the header. Typically, the sync committee for the next period is determined from a finalized block in the current period. However, in the case that the end of the sync committee period is reached before any block containing a sync committee update is finalized, a user may call the `LightClient.force` function to apply the update with the most signatures, even if that update has less than a majority of signatures. A forced update may have as few as 10 participating signers, as determined by the constant `MIN_SYNC_COMMITTEE_PARTICIPANTS`.

```
/// @notice In the case there is no finalization for a sync committee rotation, this
method
///         is used to apply the rotate update with the most signatures throughout
the period.
/// @param period The period for which we are trying to apply the best rotate update
for.
function force(uint256 period) external {
    LightClientRotate memory update = bestUpdates[period];
    uint256 nextPeriod = period + 1;

    if (update.step.finalizedHeaderRoot == 0) {
        revert("Best update was never initialized");
    } else if (syncCommitteePoseidons[nextPeriod] != 0) {
        revert("Sync committee for next period already initialized.");
    } else if (getSyncCommitteePeriod(getCurrentSlot()) < nextPeriod) {
        revert("Must wait for current sync committee period to end.");
    }

    setSyncCommitteePoseidon(nextPeriod, update.syncCommitteePoseidon);
}
```

Figure 8.1: telepathy/contracts/src/lightclient/LightClient.sol#123–139

Proving sync committee updates via the rotate ZK circuit requires significant computational power; it is likely that there will be only a few provers online at any given time. In this case, a DoS attack against the active provers could cause the provers to be offline for a full sync committee period (~27 hours), allowing the attacker to force an update with a small minority of validator stake. The attacker would then gain full control of the light client and be able to steal funds from any systems dependent on the correctness of the light client.

Exploit Scenario

Alice, a malicious ETH2.0 validator, controls about 5% of the total validator stake, split across many public keys. She waits for a sync committee period, includes at least 10 of her public keys, then launches a DoS against the active Telepathy provers, using an attack such as that described in TOB-SUCCINCT-1 or an attack against the offchain prover/relayer client itself. Alice creates a forged beacon block with a new sync committee containing only her own public keys, then uses her 10 active committee keys to sign the block. She calls `LightClient.rotate` with this forged block and waits until the sync committee period ends, finally calling `LightClient.force` to gain control over all future light client updates.

Recommendations

Short term, consider removing the `LightClient.force` function, extending the waiting period before updates may be forced, or introducing a privileged role to mediate forced updates.

Long term, explicitly document expected liveness behavior and associated safety tradeoffs.

9. G1AddMany does not check for the point at infinity

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCCINCT-9
Target: `circuits/circuits/{sync_committee.circom, bls.circom}`	

Description

The `G1AddMany` circuit aggregates multiple public keys into a single public key before verifying the BLS signature. The outcome of the aggregation is used within `CoreVerifyPubkeyG1` as the public key. However, `G1AddMany` ignores the value of the final `out_bits`, and wrongly converts a point at infinity to a different point when all participation bits are zero.

```
template G1AddMany(SYNC_COMMITTEE_SIZE, LOG_2_SYNC_COMMITTEE_SIZE, N, K) {
    signal input pubkeys[SYNC_COMMITTEE_SIZE][2][K];
    signal input bits[SYNC_COMMITTEE_SIZE];
    signal output out[2][K];

    [...]
    for (var i = 0; i < 2; i++) {
        for (var j = 0; j < K; j++) {
            out[i][j] <== reducers[LOG_2_SYNC_COMMITTEE_SIZE-1].out[0][i][j];
        }
    }
}
```

Figure 9.1: BLS key aggregation without checks for all-zero participation bits (telepathy/circuits/circuits/bls.circom#16–48)

Recommendations

Short term, augment the `G1AddMany` template with an output signal that indicates whether the aggregated public key is the point at infinity. Check that the aggregated public key is non-zero in the calling circuit by verifying that the output of `G1AddMany` is not the point at infinity (for instance, in `VerifySyncCommitteeSignature`).

Long term, assert that all provided elliptic curve points are non-zero before converting them to affine form and using them where a non-zero point is expected.

10. TargetAMB receipt proof may behave unexpectedly on future transaction types

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCCINCT-10
Target: `contracts/src/libraries/StateProofHelper.sol`	

Description

The `TargetAMB` contract can relay transactions from the `SourceAMB` via events logged in transaction receipts. The contract currently ignores the version specifier in these receipts, which could cause unexpected behavior in future upgrade hard-forks.

To relay a transaction from a receipt, the user provides a Merkle proof that a particular transaction receipt is present in a specified block; the relevant event is then parsed from the transaction receipt by the `TargetAMB`.

```
function getEventTopic(...){
    ...
    bytes memory value =
        MerklePatriciaProofVerifier.extractProofValue(receiptRoot, key, proofAsRLP);
    RLPReader.RLPItem memory valueAsItem = value.toRlpItem();
    if (!valueAsItem.isList()) {
        // TODO: why do we do this ...
        valueAsItem.memPtr++;
        valueAsItem.len--;
    }
    RLPReader.RLPItem[] memory valueAsList = valueAsItem.toList();
    require(valueAsList.length == 4, "Invalid receipt length");
    // In the receipt, the 4th entry is the logs
    RLPReader.RLPItem[] memory logs = valueAsList[3].toList();
    require(logIndex < logs.length, "Log index out of bounds");
    RLPReader.RLPItem[] memory relevantLog = logs[logIndex].toList();
    ...
}
```

Figure 10.1: telepathy/contracts/src/libraries/StateProofHelper.sol#L44–L82

The logic in figure 10.1 checks if the transaction receipt is an RLP list; if it is not, the logic skips one byte of the receipt before continuing with parsing. This logic is required in order to properly handle legacy transaction receipts as defined in EIP-2718. Legacy transaction receipts directly contain the RLP-encoded list `rlp([status, cumulativeGasUsed, logsBloom, logs])`, whereas EIP-2718 wrapped transactions are of the form

`TransactionType||TransactionPayload`, where `TransactionType` is a one-byte indicator between `0x00` and `0x7f` and `TransactionPayload` may vary depending on the transaction type. Current valid transaction types are `0x01` and `0x02`. New transaction types may be added during routine Ethereum upgrade hard-forks.
The `TransactionPayload` field of type `0x01` and `0x02` transactions corresponds exactly to the `LegacyTransactionReceipt` format; thus, simply skipping the initial byte is sufficient to handle these cases. However, EIP-2718 does not guarantee this backward compatibility, and future hard-forks may introduce transaction types for which this parsing method gives incorrect results. Because the current implementation lacks explicit validation of the transaction type, this discrepancy may go unnoticed and lead to unexpected behavior.

Exploit Scenario
An Ethereum upgrade fork introduces a new transaction type with a corresponding transaction receipt format that differs from the legacy format. If the new format has the same number of fields but with different semantics in the fourth slot, it may be possible for a malicious user to insert into that slot a value that parses as an event log for a transaction that did not take place, thus forging an arbitrary bridge message.

Recommendations
Short term, check the first byte of `valueAsItem` against a list of allowlisted transaction types, and revert if the transaction type is invalid.

Long term, plan for future incompatibilities due to upgrade forks; for example, consider adding a semi-trusted role responsible for adding new transaction type identifiers to an allowlist.

11. RLPReader library does not validate proper RLP encoding

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SUCCINCT-11
Target: `contracts/lib/Solidity-RLP/RLPReader.sol`	

Description

The `TargetAMB` uses the external `RLPReader` dependency to parse RLP-encoded nodes in the Ethereum state trie, including those provided by the user as part of a Merkle proof.

When parsing a byte string as an `RLPItem`, the library does not check that the encoded payload length of the `RLPitem` matches the length of the underlying bytes.

```
/*
 * @param item RLP encoded bytes
 */
function toRlpItem(bytes memory item) internal pure returns (RLPItem memory) {
    uint256 memPtr;
    assembly {
        memPtr := add(item, 0x20)
    }

    return RLPItem(item.length, memPtr);
}
```

Figure 11.1: Solidity-RLP/contracts/RLPReader.sol#51–61

If the encoded byte length of the `RLPitem` is too long or too short, future operations on the `RLPItem` may access memory before or after the bounds of the underlying buffer. More generally, because the Merkle trie verifier assumes that all input is in the form of valid RLP-encoded data, it is important to check that potentially malicious data is properly encoded.

While we did not identify any way to convert improperly encoded proof data into a proof forgery, it is simple to give an example of an out-of-bounds read that could possibly lead in other contexts to unexpected behavior. In figure 11.2, the result of `items[0].toBytes()` contains many bytes read from memory beyond the bounds allocated in the initial byte string.

```
RLPReader.RLPItem memory item = RLPReader.toRlpItem('\xc3\xd0');
RLPReader.RLPItem[] memory items = item.toList();
assert(items[0].toBytes().length == 16);
```

Figure 11.2: Out-of-of-bounds read due to invalid RLP encoding

In this example, `RLPReader.toRLPItem` should revert because the encoded length of three bytes is longer than the payload length of the string; similarly, the call to `toList()` should fail because the nested `RLPItem` encodes a length of 16, again more than the underlying buffer.

To prevent such ill-constructed nested `RLPItems`, the internal `numItems` function should revert if `currPtr` is not exactly equal to `endPtr` at the end of the loop shown in figure 11.3.

```
// @return number of payload items inside an encoded list.
function numItems(RLPItem memory item) private pure returns (uint256) {
    if (item.len == 0) return 0;

    uint256 count = 0;
    uint256 currPtr = item.memPtr + _payloadOffset(item.memPtr);
    uint256 endPtr = item.memPtr + item.len;
    while (currPtr < endPtr) {
        currPtr = currPtr + _itemLength(currPtr); // skip over an item
        count++;
    }

    return count;
}
```

Figure 11.3: Solidity-RLP/contracts/RLPReader.sol#256–269

Recommendations

Short term, add a check in `RLPReader.toRLPItem` that validates that the length of the argument exactly matches the expected length of prefix + payload based on the encoded prefix. Similarly, add a check in `RLPReader.numItems`, checking that the sum of the encoded lengths of sub-objects matches the total length of the RLP list.

Long term, treat any length values or pointers in untrusted data as potentially malicious and carefully check that they are within the expected bounds.

12. TargetAMB _executeMessage lacks contract existence checks

Severity: **Low**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-SUCCINCT-12
Target: `contracts/src/amb/TargetAMB.sol`	

Description

When relaying messages on the target chain, the `TargetAMB` records the success or failure of the external contract call so that off-chain clients can track the success of their messages. However, if the recipient of the call is an externally owned-account or is otherwise empty, the `handleTelepathy` call will appear to have succeeded when it was not processed by any recipient.

```
bytes memory recieveCall = abi.encodeWithSelector(
    ITelepathyHandler.handleTelepathy.selector,
    message.sourceChainId,
    message.senderAddress,
    message.data
);
address recipient = TypeCasts.bytes32ToAddress(message.recipientAddress);
(status,) = recipient.call(recieveCall);

if (status) {
    messageStatus[messageRoot] = MessageStatus.EXECUTION_SUCCEEDED;
} else {
    messageStatus[messageRoot] = MessageStatus.EXECUTION_FAILED;
}
```

Figure 12.1: `telepathy/contracts/src/amb/TargetAMB.sol#150–164`

Exploit Scenario

A user accidentally sends a transaction to the wrong address or an address that does not exist on the target chain. The UI displays the transaction as successful, possibly confusing the user further.

Recommendations

Short term, change the `handleTelepathy` interface to expect a return value and check that the return value is some magic constant, such as the four-byte ABI selector. See OpenZeppelin's `safeTransferFrom / IERC721Reciever` pattern for an example.

Long term, ensure that all low-level calls behave as expected when handling externally owned accounts.

13. LightClient is unable to verify some block headers

Severity: **Medium**	Difficulty: **Low**
Type: Authentication	Finding ID: TOB-SUCCINCT-13
Target: `contracts/src/lightclient/LightClient.sol`	

This issue was discovered and relayed to the audit team by the Telepathy developers. We include it here for completeness and to provide our fix recommendations.

Description

The `LightClient` contract expects beacon block headers produced in a period prior to the period in which they are finalized to be signed by the wrong sync committee; those blocks will not be validated by the `LightClient`, and AMB transactions in these blocks may be delayed.

The Telepathy light client tracks only block headers that are "finalized," as defined by the ETH2.0 Casper finality mechanism. Newly proposed, unfinalized beacon blocks contain a `finalized_checkpoint` field with the most recently finalized block hash.

The `Step` circuit currently exports only the slot number of this nested, finalized block as a public input. The `LightClient` contract uses this slot number to determine which sync committee it expects to sign the update. However, the correct slot number for this use is in fact that of the wrapping, unfinalized block. In some cases, such as near the edge of a sync committee period or during periods of delayed finalization, the two slots may not belong to the same sync committee period. In this case, the signature will fail to verify, and the `LightClient` will become unable to validate the block header.

Exploit Scenario

A user sends an AMB message using the `SourceAMB.sendViaLog` function. The beacon block in which this execution block is included is late within a sync committee period and is not finalized on the beacon chain until the next period. The new sync committee signs the block, but this signature is rejected by the light client because it expects a signature from the old committee. Because this header cannot be finalized in the light client, the `TargetAMB` cannot relay the message until some future block in the new sync committee period is finalized in the light client, causing delivery delays.

Recommendations

Short term, Include `attestedSlot` in the public input commitment to the `Step` circuit. This can be achieved at no extra cost by packing the eight-byte `attestedSlot` value alongside the eight-byte `finalizedSlot` value, which currently is padded to 32 bytes.

Long term, add additional unit and end-to-end tests to focus on cases where blocks are near the edges of epochs and sync committee periods. Further, reduce gas usage and circuit complexity by packing all public inputs to the step function into a single byte array that is hashed in one pass, rather than chaining successive calls to SHA-256, which reduces the effective rate by half and incurs overhead due to the additional external precompile calls.

14. OptSimpleSWU2 Y-coordinate output is underconstrained

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCCINCT-14
Target: `circuits/circuits/pairing/bls12_381_hash_to_G2.circom`	

Description

The `OptSimpleSWU2` circuit does not check that its Y-coordinate output is a properly formatted BigInt. This violates the canonicity assumptions of the `Fp2Sgn0` circuit and other downstream components, possibly leading to unexpected nondeterminism in the sign of the output of `MapToG2`. Incorrect results from `MapToG2` would cause the circuit to verify the provided signature against a message different from that in the public input. While this does not allow malicious provers to forge signatures on arbitrary messages, this additional degree of freedom in witness generation could interact negatively with future changes to the codebase or instantiations of this circuit.

```
var Y[2][50];
...
component Y_sq = Fp2Multiply(n, k, p);
// Y^2 == g(X)
for(var i=0; i<2; i++)for(var idx=0; idx<k; idx++){
    out[1][i][idx] <-- Y[i][idx];
    Y_sq.a[i][idx] <== out[1][i][idx];
    Y_sq.b[i][idx] <== out[1][i][idx];
}
for(var i=0; i<2; i++)for(var idx=0; idx<k; idx++){
    Y_sq.out[i][idx] === isSquare * (gX0.out[i][idx] - gX1.out[i][idx]) +
gX1.out[i][idx];
}

// sgn0(Y) == sgn0(t)
component sgn_Y = Fp2Sgn0(n, k, p);
for(var i=0; i<2; i++)for(var idx=0; idx<k; idx++)
    sgn_Y.in[i][idx] <== out[1][i][idx];

sgn_Y.out === sgn_in.out;
```

Figure 14.1: telepathy/circuits/circuits/pairing/bls12_381_hash_to_G2.circom#199–226

A malicious prover can generate a witness where that Y-coordinate is not in its canonical representation. `OptSimpleSWU2` calls `Fp2Sgn0`, which in turn calls `FpSgn0`. Although

FpSgn0 checks that its input is less than p using BigLessThan, that is not sufficient to guarantee that its input is canonical. BigLessThan allows limbs of its inputs to be greater than or equal to 2^n, so long as the difference a[i]-b[i] is in $[-2^n, 2^n)$. Violating FpSgn0's assumption that the limbs are all in $[0, 2^n)$ could lead to unexpected behavior in some cases—for example, if FpSgn0's output is incorrect, MapToG2 could return a point with an incorrect sign.

```
template FpSgn0(n, k, p){
    signal input in[k];
    signal output out;

    // constrain in < p
    component lt = BigLessThan(n, k);
    for(var i=0; i<k; i++){
        lt.a[i] <== in[i];
        lt.b[i] <== p[i];
    }
    lt.out === 1;

    // note we only need in[0] !
    var r = in[0] % 2;
    var q = (in[0] - r) / 2;
    out <-- r;
    signal div;
    div <-- q;
    out * (1 - out) === 0;
    in[0] === 2 * div + out;
}
```

Figure 14.2: telepathy/circuits/circuits/pairing/fp.circom#229–249

Exploit Scenario

A malicious prover can use a malformed BigInt to change the computed sign of the out signal of OptSimpleSWU2 at proving time. That changed sign will cause the rest of the circuit to accept certain incorrect values in place of H(m) when checking the BLS signature. The prover could then successfully generate a proof with certain malformed signatures.

Recommendations

Short term, use Num2Bits to constrain the Y-coordinate output of OptSimpleSWU2 to be a well-formed BigInt.

Long term, clearly document and validate the representation of the output values in templates such as OptSimpleSWU2.

Summary of Recommendations

The Succinct Labs Telepathy circuits and smart contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that Succinct Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Document assumptions on all inputs and enforce these assumptions early in the circuit. Consider adopting Circom signal tags as a way to clearly document assumptions and prevent missing or duplicate checks. For example, TOB-SUCCINCT-1 could have been prevented by adding a check as follows, where `maxbits` is a tag only ever added after constraining the maximum bits of a value.

```
template G1BigIntToSignFlag(N, K) {
    signal input {maxbits} in[K];
    signal output out;

    assert(in.maxbits <= 55);
    ...
}
```

- Routinely check for security bug fixes in hard-copied dependencies such as `circom-pairing`.
- Use proper chaining modes for Poseidon and SHA-256 commitments, for both performance and security benefits.
- Aggressively validate dynamically-sized and structured user input, such as RLP-encoded values and Merkle proofs.
- Consider adopting the Optimism Bedrock RLP and Merkle Patricia trie libraries. Do not use the current production Optimism libraries, as the MPT verifier is susceptible to a proof forgery attack.

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
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.

Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists code quality findings that we identified throughout our review.

- Several files do not correctly import dependencies needed. For instance, `sha256.circom` uses templates from `circomlib/bitify.circom`, but does not import the file. The missing imports do not cause build issues because the higher-level files import the functions, but they prevent directly testing the circuit with unit tests. Other instances are `hash_to_field.circom`, which should import `bls.circom`, and `bls.circom`, which should import `constants.circom`.
- The public input hashing in the `SerializeLightClientStepInputs` template uses a sequence of three chained 64-byte SHA-256 calls instead of one 128-byte SHA-256 call, which induces two extra calls to the SHA-256 compression function.
- `VerifySyncCommitteeSignature` checks that the witnessed Y coordinates lie on the curve. This check is redundant and is already implemented in the `Rotate` circuit when the Y values were added to the commitment.

```
component isValidPoint[SYNC_COMMITTEE_SIZE];
for (var i = 0; i < SYNC_COMMITTEE_SIZE; i++) {
    isValidPoint[i] = SubgroupCheckG1WithValidX(N, K);
    for (var j = 0; j < 2; j++) {
        for (var k = 0; k < K; k++) {
            isValidPoint[i].in[j][k] <== pubkeys[i][j][k];
        }
    }
}
```

Figure C.1: Point validation within `VerifySyncCommitteeSignature` (`telepathy/circuits/circuits/sync_committee.circom#54–62`)

- The `CoreVerifyPubkeyG1` template checks that public keys belong to the appropriate subgroup. This check appears unnecessary and should be delegated to the `Rotate` circuit.

```
component pubkey_valid = SubgroupCheckG1(n, k);
for(var i=0; i<2; i++)for(var idx=0; idx<k; idx++)
    pubkey_valid.in[i][idx] <== pubkey[i][idx];
```

Figure C.2: Subgroup membership check (`telepathy/circuits/circuits/pairing/bls_signature.circom#112–114`)

- The template `SubgroupCheckG1WithValidX` does not check that the given point belongs to the subgroup generated by the generator G1, and checks only that the point lies on the curve. We recommend specifying this in the comment and adopting a name that does not imply a subgroup constraint, such as `IsPointOnBLSCurve`.

- The `HashToField` template is parametrized with a `COUNT` value specifying the number of field elements the `hash_to_field` function returns. The underlying `hash_to_field` function uses a hash function to derive an intermediary uniform random byte string whose length depends on the number of field elements desired. However, the `HashToField` template hard-codes the number of field elements to two, which is insufficient if a larger `COUNT` value was requested; the ultimate results are compilation errors. The `COUNT` parameter can be omitted given that only two field elements are required throughout the codebase. Alternatively, `BYTES_LEN` should be computed accordingly and assigned to `COUNT * M * L`.

```
template HashToField(MSG_LEN, COUNT) {
    signal input msg[MSG_LEN];

    var DST[43] = getDomainSeperatorTag();
    var P[7] = getBLS128381Prime();
    var DST_LEN = 43;
    var LOG2P = 381;
    var M = 2;
    var L = 64;
    var BYTES_LEN = 256;
```

Figure C.3: `BYTES_LEN` is independent of `COUNT` and is hard-coded to 256 (`telepathy/circuits/circuits/hash_to_field.circom#L11–L20`)

- The variables `S256S_0_INPUT_BYTE_LEN` and `S256S_INPUT_BYTE_LEN` in the template `HashToField` duplicate the computation of the same value `B_IN_BYTES + 1 + DST_LEN + 1`. This deduplication makes the code slightly harder to read and can introduce potential errors. For instance, if the last `+ 1` addition was forgotten in `S256S_INPUT_BYTE_LEN`, the circuits would run fine, but the results would no longer conform to the specification. We stress that although this does not apply to the codebase, a single computed value would increase readability and make potential errors easier to spot.

- Comparisons with literal `true` on lines 173 and 193 of `LightClient.sol` are unnecessary:

```
require(verifyProofRotate(proof.a, proof.b, proof.c, inputs) == true);
```

Figure C.4: `telepathy/contracts/src/lightclient/LightClient.sol#193`

- `TargetAMB.lightclient` and `TargetAMB.broadcaster` are set only by the constructor and thus may be marked immutable for efficiency.

- The following comment should read “Sets the current slot…”

```
/// @notice Gets the current slot for the chain the light client is reflecting.
```

```
function setHead(uint256 slot, bytes32 root) internal {
```

Figure C.5: telepathy/contracts/src/lightclient/LightClient.sol#206–207

- The Poseidon sync committee commitment is passed to the `Rotate` circuit as a field element, but stored in the `Step` circuit SHA-256 input commitment as a little-endian byte string. This necessitates an additional on-chain byte-reversal operation for each sync committee rotation, which could be avoided if the commitment was uniformly treated as big-endian.

```
inputs[64] = uint256(SSZ.toLittleEndian(uint256(update.syncCommitteePoseidon)));
```

Figure C.6: telepathy/contracts/src/lightclient/LightClient.sol#191

- The `SSZ.toLittleEndian` function can be made significantly more gas-efficient by avoiding loops and using a log-time reversal algorithm. See https://ethereum.stackexchange.com/a/83627. Additionally, eight-byte values, such as Slot numbers, should be reversed before padding to 32 bytes instead of afterwards in order to save gas.

- Input serialization to the `Step` function is done by chaining calls to SHA-256. This halves the effective rate of the hash function and, because of the significant external call overhead for precompile use, is very inefficient compared to serializing all input data and calling SHA-256 just once.

- Template `SSZRestoreMerkleRoot` accepts a depth parameter and an index parameter. The template functions correctly only if `2^(depth + 1) > index`; an assertion to this effect should be added to the template.

- `srcSlotTxSlotPack` is implemented using a variable-length byte array, requiring 32 bytes of calldata for length in addition to the 16 bytes of payload. Consider packing the two eight-byte values into a single `uint32` and extracting the slots via bitwise operations.

```
(uint64 srcSlot, uint64 txSlot) = abi.decode(srcSlotTxSlotPack, (uint64, uint64));
```

Figure C.7: telepathy/contracts/src/amb/TargetAMB.sol#98

- An explicit check for a zero return value should be added when fetching the `executionStateRoot`. A clear revert message will allow better error reporting.

```
bytes32 executionStateRoot =
    lightClients[message.sourceChainId].executionStateRoots(slot);
```

Figure C.8: telepathy/contracts/src/amb/TargetAMB.sol#57–58

- The `_checkPreconditions` function of `TargetAMB` should ensure that the source chain ID is initialized before proceeding with processing the message. In particular, `lightClients[message.sourceChainId]` and `broadcasters[message.sourceChainId]` should be required to be nonzero.

D. Automated Analysis Tool Configuration

D.1 Slither

We used Slither to detect common issues and anti-patterns. Low-severity issues may be filtered with the `--exclude-informational` and `--exclude-low` flags. Slither discovered a number of minor code quality issues and potential optimizations. Integrating Slither into your testing environment can help improve the overall quality of your smart contract code.

```
slither --exclude-informational --exclude-low --filter-paths
src/LightClient,src/amb/mocks,script,examples,lib/forge-std,lib/openzeppelin-contrac
ts,lib/v3-core,test .
```

Figure D.1: Example Slither configuration

D.2 Picus

We used Picus to check for determinacy of several critical subcircuits. We were able to verify the strong (witness) determinacy of the following subcircuits when instantiated with the parameters used in the overall circuit design.

```
test_bigmult_shortlong_2d.circom
test_bigmult_shortlong_2d_unequal.circom
test_elliptic_curve_add_unequal.circom
test_elliptic_curve_add_unequal_fp2.circom
test_elliptic_curve_double.circom
test_elliptic_curve_double_fp2.circom
test_fp2_invert.circom
test_fp_sgn0.circom
test_signed_check_carry_mod_to_zero.circom
test_signed_fp2_divide.circom
test_signed_fp_carry_mod_p.circom
```

Figure D.2: Circuits tested for determinacy

We ran Picus on each circuit with the following script:

```
import subprocess
import pathlib
from pqdm.processes import pqdm

def compile(filename, build_dir):
    cmd = f"circom --r1cs --sym --O0 {str(filename)} -o {str(build_dir)}"
    subprocess.run(cmd.split(),capture_output=True)

def picus(filename, outfile):
    cmd = f"racket ~/Picus/test-v3-uniqueness.rkt --solver z3 --r1cs {filename}"
    with open(outfile, 'w') as f:
```

```
        subprocess.run(cmd.split(), stdout=f)

if __name__ == "__main__":
    wd = pathlib.Path("./")
    circom_files = wd.glob("*.circom")
    build_dir = wd.joinpath("build")
    build_dir.mkdir(parents=True, exist_ok=True)
    print("Compiling:")
    pqdm([(f, build_dir) for f in circom_files], compile, n_jobs=10,
argument_type="args")
    r1cs_files = build_dir.glob("*.r1cs")
    print("Solving:")
    pqdm([(f, f.with_suffix('.picus')) for f in r1cs_files], picus, n_jobs=10,
argument_type="args")
```

Figure D.3: `picus.py`

D.3 Circomspect

We used Circomspect to detect known vulnerabilities in the Circom code. The analysis did not identify any issues.

```
import subprocess
import pathlib
from pqdm.processes import pqdm

circuits_path = pathlib.Path("./circuits/circuits/")
circtuis_files = circuits_path.glob("**/*.circom")
circomspect_path = pathlib.Path("./circomspect/")

def analyze(filename):
    output_file = circomspect_path.joinpath(filename).with_suffix(".sarif")
    output_file.parent.mkdir(parents=True, exist_ok=True)
    _cmd = f"circomspect -s {str(output_file)} {str(filename)}"
    cmd = _cmd.split()
    subprocess.run(cmd,capture_output=True)

if __name__ == "__main__":
    circtuis_files = list(circuits_path.glob("**/*.circom"))
    results = pqdm(circtuis_files, analyze, n_jobs=10)
```

Figure D.4: `run-circomspect.py`

E. 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 February 21 to February 23, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Succinct Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

The primary commit for this fix review is 79b44f1. As of the time of writing, this corresponds to the head of the succinctlabs/telepathy/main branch.

In summary, Succinct Labs has resolved 13 of the issues described in this report and has partially resolved one issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Prover can lock user funds by including ill-formed BigInts in public key commitment	High	Resolved
2	Prover can lock user funds by supplying non-reduced Y values to G1BigIntToSignFlag	High	Resolved
3	Incorrect handling of point doubling can allow signature forgery	High	Resolved
4	EllipticCurveAdd mishandles points at infinity	Informational	Resolved
5	Circom circuits lack adequate testing framework	Informational	Partially Resolved
6	Poseidon commitment uses a non-standard hash construction	Informational	Resolved
7	Merkle root reconstruction is vulnerable to forgery via proofs of incorrect length	High	Resolved

8	LightClient forced finalization could allow bad updates in case of a DoS	High	Resolved
9	G1AddMany does not check for the point at infinity	Informational	Resolved
10	TargetAMB receipt proof may behave unexpectedly on future transaction types	Informational	Resolved
11	RLPReader library does not validate proper RLP encoding	Low	Resolved
12	TargetAMB _executeMessage lacks contract existence checks	Low	Resolved
13	LightClient is unable to verify some block headers	Medium	Resolved
14	OptSimpleSWU2 Y-coordinate output is underconstrained	Low	Resolved

Detailed Fix Review Results

TOB-SUCCINCT-1: Prover can lock user funds by including ill-formed BigInts in public key commitment
Resolved in commit 90888da. The client added additional constraints in the `Rotate` circuit requiring each prover-provided public key X and Y coordinate to be properly formed BigInts.

TOB-SUCCINCT-2: Prover can lock user funds by supplying non-reduced Y values to G1BigIntToSignFlag
Resolved in commits 14b7f7b and 90ed04d. The client added an instantiation of `BigLessThan`, constraining the prover-provided public key coordinates to be reduced modulo the BLS12-381 field prime.

TOB-SUCCINCT-3: Incorrect handling of point doubling can allow signature forgery
Resolved in commit d623f3c. The client substituted `EllipticCurveAdd` for `EllipticCurveAddUnequal` and now uses the logic built into `EllipticCurveAdd` for handling points at infinity.

TOB-SUCCINCT-4: `EllipticCurveAdd` mishandles points at infinity
Resolved in commit d623f3c. The client adopted the upstream logic fixing the handling of points at infinity in `EllipticCurveAdd`.

TOB-SUCCINCT-5: Circom circuits lack adequate testing framework
Partially resolved in PR #66. The client now has a functioning unit test framework for their circom circuits, but code coverage is currently low and missing entirely on many critical components. There is also no framework for negative testing, for example it is not currently possible to easily add regression tests for previously missing constraints. Finally, we recommend that unit testing be incorporated into a CI process to ensure that new modifications to the codebase do not introduce known bugs.

TOB-SUCCINCT-6: Poseidon commitment uses a non-standard hash construction
Resolved in commit 4124cc3. The client modified the Poseidon hash chaining to use a dedicated “capacity” field element separate from the eventual “rate” output and to make use of the `initialState` input to `PoseidonEx`.

TOB-SUCCINCT-7: Merkle root reconstruction is vulnerable to forgery via proofs of incorrect length
Resolved in commit f878751. The client modified the Merkle root reconstruction loop control flow to depend only on the contract-provided `index` and not on the user-provided `branch` array.

TOB-SUCCINCT-8: `LightClient` forced finalization could allow bad updates in case of a DoS
Resolved in commit 0e744fe. The client removed the `LightClient.force` function. There is a remaining comment above the `LightClient.rotate` function that refers to the removed "best optimistic update" functionality; this comment should be amended for clarity.

TOB-SUCCINCT-9: `G1AddMany` does not check for the point at infinity
Resolved in commit aab0156. `G1AddMany` now returns a bit indicating whether the result is infinite. `VerifySyncCommitteeSignature` now constrains the aggregate public key to be finite before proceeding with verification.

TOB-SUCCINCT-10: `TargetAMB` receipt proof may behave unexpectedly on future transaction types
Resolved in PR #78 and commit 6a9eb9e. The client added checks to ensure that the transaction type was one of the three known values, reverting otherwise.

TOB-SUCCINCT-11: RLPReader library does not validate proper RLP encoding
Resolved in PR #78. The client changed the underlying RLP deserializing library from hamdiallam/Solidity-RLP to optimism-bedrock/rlp, which includes more robust checks to validate encoded lengths before reading RLP lists and strings.

TOB-SUCCINCT-12: `TargetAMB _executeMessage` lacks contract existence checks
Resolved in PR #115. The `ITelepathyHandler` interface now returns a four byte "magic" value indicating successful receipt, and the `_executeMessage` function checks that this value is returned correctly before marking a transaction as successful.

TOB-SUCCINCT-13: `LightClient` is unable to verify some block headers
Resolved in commits ef128f8, 009ac66 and 2e1c080. The `Step` circuit now validates the `attestedHeader` field. The `LightClient` contract now computes sync committees relative to the `attestedHeader`.

TOB-SUCCINCT-14: `OptSimpleSWU2` Y-coordinate output is underconstrained
Resolved in commit ef128f8. The client added a `Num2Bits` instance constraining the output of `OptSimpleSWU2` to be a properly formed BigInt.

F. Upgradeability Review

Engagement Summary

After the culmination of the main security review and before the fix review, Succinct Labs modified the overall structure of the Telepathy smart contracts by enabling upgrades to the Telepathy message bridge and adding two new permissioned roles. During the fix review, we dedicated one engineer-day to reviewing these changes for implementation flaws and security pitfalls.

The target of this review is commit number 79b44f1 of the telepathy repository.

We found one flaw in the implementation of upgradeability. In addition, the new features inherently create security considerations and potential attack vectors not present in the original codebase.

Review Coverage

We manually reviewed all contracts in the `contracts/src/amb` directory, checking for issues with storage layout, proxy initialization, and access controls. Additionally, we checked for common upgradeability issues using the `slither-check-upgradeability` tool.

We did not review the implementation of any OpenZeppelin library contracts, multi-sig wallet implementations, key management protocols or contract upgrade procedures.

Summary of Changes

- The new `TelepathyRouter` contract implements UUPS upgradeability via the OpenZeppelin `UUPSUpgradeable` library and subsumes the `SourceAMB` and `TargetAMB` contracts.
- Two new permissioned roles have been added:
  - The “guardian” role is responsible for freezing/unfreezing source chains on the `TargetAMB` and enabling/disabling sending on the `SourceAMB`. The guardian cannot forge messages or otherwise steal funds from higher-level protocols but can delay operation indefinitely.
  - The “timelock” role is responsible for upgrading the smart contract. Succinct Labs intends to instantiate this role using an OpenZeppelin `TimelockController`, controlled by a multi-signature wallet. The timelock role can perform arbitrary actions, including forging messages from the bridge and stealing user funds from higher-level protocols.
- `TargetAMB` instances now may receive messages from many source chains, replacing the prior static association with a single `SourceAMB` instance. The timelock role manages the dynamic mapping from source chain IDs to `LightClient` instances.

Implementation Issues

- `TelepathyRouter` does not disable initializers in the constructor of its logic contract. Failure to disable initializers in the logic contract can result in malicious actors taking control of the logic contract, which may enable unexpected interactions between the logic contract and the proxy. A malicious actor may also convince unsuspecting users to interact directly with the logic contract rather than the proxy. The malicious actor would then fully control those users' views of the source chain states and could steal their funds.

Recommendations

- Succinct Labs must insert a call to `_disableInitializers` in the constructor of `TelepathyRouter`. Doing so will prevent the logic contract from being initialized and used without the overlying proxy. Refer to the OpenZeppelin "Writing Upgradeable Contracts" documentation for more information.
- Succinct Labs must ensure that the private keys authorized to perform smart contract upgrades are well-secured. Some mitigation is provided by a time-delay contract, providing users of the Telepathy protocol time to react to unexpected or malicious contract upgrade attempts. Succinct Labs has also indicated that they intend to use a multi-signature wallet to manage upgrades. Refer to Appendix G for best practices regarding the use of multi-sig wallets.
- Succinct Labs must take care when upgrading the Telepathy contracts to avoid pitfalls in `delegatecall`-based upgrades, such as storage layout changes. Refer to Appendix H for detailed guidance.
- Telepathy users should consistently monitor the `TimelockController` contract for unexpected events. Security-relevant events include changes to the `TimelockController` owner or `minDelay` and any calls to the Telepathy contracts.

G. Security Best Practices for the Use of a Multi-sig Wallet

The principles of a consensus requirement for a sensitive action, such as upgrading a smart contract, are to mitigate the risks of:

- Any one person's judgment overruling the others'
- Any one person's mistake causing a failure
- Any one person's credential compromise causing a failure

In the 2-of-3 multisignature smart contract upgrade example, the authority to execute an upgrade transaction requires a consensus of two individuals in possession of two of the wallet's three private keys. In order for this model to be useful, the guideline requirements are:

1. The private keys must be stored or held separately, and each must be respectively access-limited to separate individuals.
2. If the keys are physically held in the custody of a third-party (e.g., a bank), then multiple keys should not be stored with the same custodian (doing so would violate requirement #1).
3. The person asked to perform the second and final signature on the transaction (a.k.a. the co-signer) ought to refer to a policy—established beforehand—specifying the conditions for approving the transaction by signing it with his/her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first (forfeiting the decision responsibility back to the first signer, and defeating the security model). If the co-signer may refuse to approve the transaction for any reason, then the due-diligence conditions for approval may be unclear. That is why a policy for validating the transaction is needed. Example verification policy rules may include:

- A predetermined protocol for being asked to cosign (e.g., a half-signed transaction will only be accepted via an approved channel)
- Verification of contract binaries by independent recompilation
- A mandatory delay period and community outreach process

Requirement #4 mitigates the risk of a single stolen key. In a hypothetical example, an attacker somehow acquires the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction will reveal that the key has been stolen and that the transaction should not be co-signed. If under an active threat of violence, a "duress code" (a code word, phrase, or other system agreed upon in

advance) can be used as a covert way for one signatory to alert the others that the transaction is not being initiated willfully, without alerting the attacker.

H. Delegatecall Proxy Guidance

The `delegatecall` opcode is a very sharp tool that must be used carefully. Many high-profile exploits use little-known edge cases and counter-intuitive aspects of the `delegatecall` proxy pattern. This section outlines the most important risks to keep in mind while developing such smart contract systems. Trail of Bits developed the `slither-check-upgradeability` tool to aid in the development of secure `delegatecall` proxies; it performs safety checks relevant to both upgradeable and immutable `delegatecall` proxies.

- **Storage layout**: The storage layouts of the proxy and implementation contracts must be the same. Do not try to define the same state variables on each contract. Instead, both contracts should inherit all of their state variables from one shared base contract.
- **Inheritance**: If the base storage contract is split up, be aware that the order of inheritance impacts the final storage layout. For example, "`contract A is B,C`" and "`contract A is C,B`" will not yield the same storage layout if both `B` and `C` define state variables.
- **Initialization**: Make sure that the implementation is immediately initialized. Well-known disasters (and near disasters) have featured an uninitialized implementation contract. A factory pattern can help ensure that contracts are deployed and initialized correctly while also preventing front-running risks that might otherwise open up between contract deployment and initialization.
- **Function shadowing**: If the same method is defined in the proxy and the implementation, then the proxy's function will not be called. Be aware of `setOwner` and other administration functions that commonly exist in proxies.
- **Preventing direct implementation usage**: Consider configuring the implementation's state variables with values that prevent the implementation from being used directly. For example, a flag could be set during construction that disables the implementation and causes all methods to revert. This is particularly important if the implementation also performs `delegatecall` operations because such operations could result in the unintended self-destruction of the implementation.
- **Immutable and constant variables**: These variables are embedded into a contract's bytecode and could, therefore, become out of sync between the proxy and implementation. If the implementation has an incorrect immutable variable, this value may still be used even if the same variables are correctly set in the proxy's bytecode.

- **Contract existence checks**: All low-level calls, not just `delegatecall`, will return `true` on an address with empty bytecode. This return value could be misleading: the caller may interpret this return value to mean that the call successfully executed the operation, when it did not. This return value may also cause important safety checks to be silently skipped. Be aware that while a contract's constructor is running, the contract address's bytecode remains empty until the end of the constructor's execution. We recommend rigorously verifying that all low-level calls are properly protected against nonexistent contracts. Keep in mind that most proxy libraries (such as the one written by OpenZeppelin) do not perform contract existence checks automatically.

For more information regarding `delegatecall` proxies in general, refer to our blog posts and presentations on the subject:

- Contract upgrade anti-patterns: This blog post describes the difference between a downstream data contract and `delegatecall` proxies that use an upstream data contract and how these patterns impact upgradeability.
- How the Diamond standard falls short: This blog post dives deep into `delegatecall` risks that apply to all contracts, not only those that follow the Diamond standard.
- Breaking Aave upgradeability: This blog post describes a subtle problem that Trail of Bits discovered in Aave `AToken` contracts that resulted from the interplay between `delegatecall` proxies, contact existence checks, and unsafe initializations.
- Contract upgrade risks and recommendations: This Trail of Bits presentation describes best practices for developing upgradeable `delegatecall` proxies. The section starting at 5:49 describes some general risks that also apply to non-upgradeable proxies.