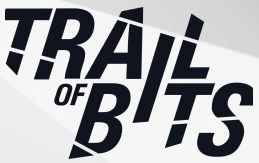

Anza Token-2022 Confidential Transfer, Cryptography

Security Assessment

January 30, 2026

Prepared for:
Will Hickey
Anza

Prepared by: **Joe Doyle and Marc Ilunga**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Tara Goodwin-Ruffus, Project Manager
tara.goodwin-ruffus@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

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

Date	Event
November 13, 2025	Pre-project kickoff call
December 18, 2025	Delivery of report draft; report readout meeting
January 30, 2026	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Anza engaged Trail of Bits to review the security of its Token-2022 confidential transfer implementation. The confidential transfer system is a set of extensions that enable transactions with confidential amounts on the Solana blockchain. The system uses the twisted ElGamal public key encryption scheme for confidentiality and a specialized suite of zero-knowledge proofs to ensure soundness and correctness of the system in a privacy-preserving manner.

A team of two consultants conducted the review from November 17 to December 16, 2025, for a total of seven engineer-weeks of effort. Our testing efforts focused on verifying that the zero-knowledge proofs of the confidential transfer system are implemented and used securely in accordance with the system specification and modern best practices. With full access to source code and documentation, we performed static and dynamic testing of the codebases, using automated and manual processes.

Observations and Impact

The implementations of confidential mint, burn, and transfer transactions, supported by the cryptographic proof implementations in the `zk-sdk` library, appear to be correct. We identified two implementation patterns (TOB-ZKELG-1, TOB-ZKELG-2) that add implementation complexity and increase the potential for error. We also identified a pattern of missing test coverage for invalid proofs (TOB-ZKELG-3). We did not identify any case where these patterns led to an exploitable issue.

Since the implementation did not appear to have exploitable issues, we used Lean and Tamarin to develop formal models of system components that are amenable to formal verification tooling. Although these models are simplified, they increase confidence that the cryptographic proofs correctly use the Fiat-Shamir transform, and that the confidential transaction logic guarantees amount-privacy without breaking the correctness of the existing system. Full details of these models are included in appendix E.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Anza take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.
- **Fill in remaining test coverage gaps in `zk-sdk`.** The overall coverage of the `zk-sdk` library is already quite high, and achieving full test coverage of the proof verification functions would provide greater confidence in their soundness.

- **Explore formal verification.** The proof systems and state machine logic in the Token-2022 confidential transfer extension are limited enough that formal verification may be feasible.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

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

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	2
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the Anza Token-2022 confidential transfer extension. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the Sigma proofs and the range proofs implemented correctly and securely?
- Are there any implementation issues with the ElGamal zero-knowledge proofs, such as weak Fiat-Shamir or insufficient input validation?
- Is batch validation implemented securely?
- Are there any implementation issues that could result in the forgery of proofs, improper minting of funds, or leaking of information about confidential transactions?
- Are the zero-knowledge proofs securely and correctly integrated into the Token-2022 program?

Project Targets

The engagement involved a review and testing of the following targets.

zk-sdk

Repository	https://github.com/solana-program/zk-elgamal-proof/
Version	`b43d723d6279c904f5503ce2da12deaea5da050f`
Type	Rust
Platform	Native

token-2022

Repository	https://github.com/solana-program/token-2022
Version	`08692efe0e84c6740780ed8b4da2bbe3efd34307`
Type	Rust
Platform	Agave

agave (zk-elgamal-proof)

Repository	https://github.com/anza-xyz/agave
Version	`e7af3564869e1487164b7795d0b46a822eefc10c`
Type	Rust
Platform	Native

Project Coverage

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

- Manual review of the proof and encryption implementations in `zk-sdk`
- Manual review of the transfer, mint, and burn components of the confidential transfer extension
- Modeling of Fiat-Shamir transcript usage in Lean
- Modeling of state machine logic in Tamarin
- Static analysis of the Rust code using the tools detailed in the Automated Testing section

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 table below for the automated testing phase of this project. Instructions for installing the tools can be found in appendix C.

Tool	Description
`cargo-audit`	A Cargo subcommand that can be used to audit project dependencies for known vulnerabilities
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
Dylint	A tool for running Rust lints from dynamic libraries
Clippy	A Rust linter used to catch common mistakes and unidiomatic Rust code

Areas of Focus

Our automated testing and verification work focused on the following:

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

`cargo-audit`
`cargo-audit` identified several RUSTSEC advisories in the dependencies of the `token-2022` repository. We did not find any issues that affect the confidential transfer logic. We recommend running `cargo-audit` on every commit as part of CI.

cargo-llvm-cov

We ran `cargo-llvm-cov` in the `zk-sdk` library and obtained a test code coverage report. The results indicate that there are certain error cases that are not covered by the current test suite, but we did not identify issues with the logic in those cases.

Filename	Function Coverage	Line Coverage	Region Coverage
encryption/auth_encryption.rs	73.53% (25/34)	82.63% (195/236)	72.73% (72/99)
encryption/discrete_log.rs	89.47% (17/19)	86.55% (193/223)	88.89% (56/63)
encryption/elgamal.rs	79.09% (87/110)	86.16% (641/744)	69.12% (197/285)
encryption/grouped_elgamal.rs	81.82% (18/22)	90.65% (223/246)	88.71% (55/62)
encryption/macros.rs	55.56% (5/9)	55.56% (15/27)	55.56% (5/9)
encryption/pedersen.rs	88.89% (32/36)	92.72% (191/206)	91.67% (55/60)
encryption/pod/auth_encryption.rs	50.00% (3/6)	62.50% (15/24)	57.14% (4/7)
encryption/pod/elgamal.rs	40.91% (9/22)	60.87% (42/69)	45.83% (11/24)
encryption/pod/grouped_elgamal.rs	47.37% (9/19)	82.95% (107/129)	69.23% (27/39)
encryption/pod/pedersen.rs	28.57% (2/7)	35.29% (6/17)	28.57% (2/7)
pod.rs	27.27% (3/11)	50.00% (16/32)	47.62% (10/21)
range_proof/generators.rs	90.91% (10/11)	91.21% (83/91)	79.17% (19/24)
range_proof/inner_product.rs	100.00% (21/21)	97.38% (409/420)	83.82% (114/136)
range_proof/mod.rs	91.67% (22/24)	98.03% (447/456)	82.44% (108/131)
range_proof/pod.rs	70.00% (7/10)	82.09% (55/67)	68.42% (13/19)
range_proof/util.rs	92.86% (13/14)	91.76% (78/85)	84.62% (44/52)
sigma_proofs/batched_grouped_ciphertext_validity/handles_2.rs	100.00% (6/6)	100.00% (173/173)	100.00% (7/7)
sigma_proofs/batched_grouped_ciphertext_validity/handles_3.rs	100.00% (6/6)	100.00% (206/206)	100.00% (7/7)
sigma_proofs/ciphertext_ciphertext_equality.rs	100.00% (6/6)	100.00% (293/293)	72.73% (40/55)
sigma_proofs/ciphertext_commitment_equality.rs	100.00% (7/7)	100.00% (358/358)	77.55% (38/49)
sigma_proofs/errors.rs	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)
sigma_proofs/grouped_ciphertext_validity/handles_2.rs	100.00% (7/7)	99.70% (334/335)	76.74% (33/43)
sigma_proofs/grouped_ciphertext_validity/handles_3.rs	100.00% (7/7)	99.75% (397/398)	74.51% (38/51)
sigma_proofs/mod.rs	83.33% (5/6)	87.50% (21/24)	82.35% (14/17)
sigma_proofs/percentage_with_cap.rs	100.00% (15/15)	99.86% (690/691)	78.26% (54/69)
sigma_proofs/pod.rs	58.06% (18/31)	65.17% (58/89)	58.06% (18/31)
sigma_proofs/pubkey_validity.rs	100.00% (7/7)	99.25% (132/133)	82.76% (24/29)
sigma_proofs/zero_ciphertext.rs	100.00% (7/7)	100.00% (215/215)	82.86% (29/35)
transcript.rs	100.00% (13/13)	100.00% (53/53)	100.00% (16/16)
zk_elgamal_proof_program/errors.rs	20.00% (1/5)	20.00% (3/15)	20.00% (1/5)
zk_elgamal_proof_program/instruction.rs	0.00% (0/9)	0.00% (0/82)	0.00% (0/23)
zk_elgamal_proof_program/proof_data/batched_grouped_ciphertext_validity/handles_2.rs	40.00% (4/10)	93.50% (115/123)	63.33% (19/30)
zk_elgamal_proof_program/proof_data/batched_grouped_ciphertext_validity/handles_3.rs	40.00% (4/10)	94.44% (136/144)	63.64% (21/33)
zk_elgamal_proof_program/proof_data/batched_range_proof/batched_range_proof_u128.rs	70.00% (7/10)	95.74% (135/141)	74.00% (37/50)
zk_elgamal_proof_program/proof_data/batched_range_proof/batched_range_proof_u256.rs	75.00% (9/12)	94.97% (151/159)	74.14% (43/58)
zk_elgamal_proof_program/proof_data/batched_range_proof/batched_range_proof_u64.rs	70.00% (7/10)	95.71% (134/140)	74.00% (37/50)
zk_elgamal_proof_program/proof_data/batched_range_proof/mod.rs	60.00% (6/10)	90.16% (55/61)	78.95% (30/38)
zk_elgamal_proof_program/proof_data/ciphertext_ciphertext_equality.rs	40.00% (4/10)	93.89% (123/131)	64.52% (20/31)
zk_elgamal_proof_program/proof_data/ciphertext_commitment_equality.rs	40.00% (4/10)	88.06% (59/67)	61.54% (16/26)
zk_elgamal_proof_program/proof_data/grouped_ciphertext_validity/handles_2.rs	40.00% (4/10)	91.11% (82/90)	62.96% (17/27)
zk_elgamal_proof_program/proof_data/grouped_ciphertext_validity/handles_3.rs	40.00% (4/10)	92.45% (98/106)	63.33% (19/30)
zk_elgamal_proof_program/proof_data/mod.rs	0.00% (0/2)	0.00% (0/2)	0.00% (0/2)
zk_elgamal_proof_program/proof_data/percentage_with_cap.rs	40.00% (4/10)	94.41% (135/143)	64.29% (18/28)
zk_elgamal_proof_program/proof_data/pod.rs	0.00% (0/4)	0.00% (0/8)	0.00% (0/4)
zk_elgamal_proof_program/proof_data/pubkey_validity.rs	40.00% (4/10)	77.14% (27/35)	60.00% (12/20)
zk_elgamal_proof_program/proof_data/zero_ciphertext.rs	50.00% (5/10)	87.27% (48/55)	66.67% (16/24)
zk_elgamal_proof_program/state.rs	0.00% (0/7)	0.00% (0/24)	0.00% (0/9)
Totals	**69.68% (455/653)**	**91.82% (6950/7569)**	**73.19% (1417/1936)**

Semgrep
Semgrep did not identify any relevant findings in the codebase.

Clippy
Clippy did not identify any relevant findings in the codebase.

Dylint
Dylint did not identify any relevant findings in the codebase.

Summary of Findings

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

ID	Title	Type	Severity
1	Batch verification challenges do not need to be generated from the transcript	Cryptography	**Informational**
2	Transcript usage pattern is error-prone	Cryptography	**Informational**
3	Missing negative tests for proof verification functions	Testing	**Informational**

Detailed Findings

1. Batch verification challenges do not need to be generated from the transcript

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-ZKELG-1
Target: `zk-sdk/src/sigma-proofs/`	

Description

When verifying Sigma proofs, several equations must be checked homomorphically. To verify several such equations in a single multi-scalar-exponentiation call, the `zk-sdk` implementation includes batch verification, where the equations are added together, weighted by powers of a random value. For this to be sound, the prover must not be able to predict what this random value is. In the current implementation, the random value is computed as a transcript challenge after the prover's final messages are added to the transcript, as shown in figure 1.1.

```
transcript.append_scalar(b"z_s", &self.z_s);
transcript.append_scalar(b"z_x", &self.z_x);
transcript.append_scalar(b"z_r", &self.z_r);
let w = transcript.challenge_scalar(b"w"); // w used for batch verification
```

Figure 1.1: The mixing value for batch verification is a transcript challenge. (zk-elgamal-proof/zk-sdk/src/sigma_proofs/ciphertext_commitment_equality.rs#162–165)

However, since the prover does not send any additional messages after this challenge is generated, it does not need to be computed from the transcript at all, and could be generated privately by the verifier from any secure random number generator. While it is secure to derive it from the transcript, doing so introduces additional complexity. In particular, if the values `z_s`, `z_x`, and `z_r` were not included in the transcript, generating `w` randomly would still be secure, but deriving it from the transcript would not be.

Recommendations

Short term, consider replacing the batch-verification challenges with calls to `Scalar::random()`.

Long term, document which random values used by the verifier should be publicly generated and which can be privately generated, and use transcript challenges or randomly chosen values as appropriate.

2. Transcript usage pattern is error-prone

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-ZKELG-2
Target: `zk-sdk`	

Description

The general implementation pattern for Sigma protocols in the `zk-sdk` library assumes that the public inputs have already been included in the transcript. For certain proofs, there is documentation of this design pattern, but not all. For example, the `PercentageWithCapProof` type does not document this assumption, but `ZeroCiphertextProof` has a documentation comment, highlighted in figure 2.1.

```
impl ZeroCiphertextProof {
    /// Creates a zero-ciphertext proof.
    ///
    /// The function does *not* hash the public key and ciphertext into the
transcript. For
    /// security, the caller (the main protocol) should hash these public components
prior to
    /// invoking this constructor.
    ///
    /// This function is randomized. It uses `OsRng` internally to generate random
scalars.
    ///
    /// * `elgamal_keypair` - The ElGamal keypair associated with the ciphertext to
be proved
    /// * `ciphertext` - The main ElGamal ciphertext to be proved
    /// * `transcript` - The transcript that does the bookkeeping for the
Fiat-Shamir heuristic
    pub fn new(
```

Figure 2.1: The documentation comment describing what the calling code is required to do (zk-elgamal-proof/zk-sdk/src/sigma_proofs/zero_ciphertext.rs#49–61)

It appears that these proofs are used in a pattern that does include these public inputs in the proof, via `new_transcript` methods such as the one shown in figure 2.2. Nonetheless, this appears to be an unnecessary implementation risk, since failing to include one of these inputs could easily lead to a proof-forgery vulnerability.

```
impl GroupedCiphertext3HandlesValidityProofContext {
    fn new_transcript(&self) -> Transcript {
        let mut transcript =
```

```
Transcript::new(b"GroupedCiphertext3HandlesValidityProof");

        transcript.append_pubkey(b"source-pubkey", &self.source_pubkey);
        transcript.append_pubkey(b"destination-pubkey", &self.destination_pubkey);
        transcript.append_pubkey(b"auditor-pubkey", &self.auditor_pubkey);
        transcript
            .append_grouped_ciphertext_3_handles(b"grouped-ciphertext",
&self.grouped_ciphertext);

        transcript
    }
}
```

Figure 2.2: An example `new_transcript` implementation (agave/zk-token-sdk/src/instruction/grouped_ciphertext_validity/handles_3.rs#138–150)

Recommendations

Short term, consider moving these transcript append operations into the proof implementations themselves.

Long term, ensure that all uses of the Fiat-Shamir heuristic are done in a misuse-resistant manner.

3. Missing negative tests for proof verification functions

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Testing	Finding ID: TOB-ZKELG-3
Target: `zk-sdk`	

Description

The `verify()` functions of several proofs do not have any tests exercising the branch where a proof fails the final verification equation. Although all of them perform the check correctly, the existing test suite would not be able to detect a trivial implementation that skips these checks. The proofs missing these tests are the following:

- `InnerProductProof`
- `GroupedCiphertext2HandlesValidityProof`
- `GroupedCiphertext3HandlesValidityProof`
- `PercentageWithCapProof`
- `PubkeyValidityProof`

Recommendations

Short term, add negative tests for the uncovered proofs.

Long term, ensure that testing covers both success and failure cases.

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**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
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 Quality Recommendations

This appendix contains recommendations that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **Verify the length of inputs while deserializing Sigma proofs.** Various Sigma proof implementations have a `from_bytes` function to deserialize proofs that are provided as byte arrays. The functions truncate the inputs to the expected length and proceed to deserialize that input, as figure B.1 shows. As a consequence, different byte arrays can deserialize to the same proof. In the confidential transfer codebases, `from_bytes` is guaranteed to be called with the byte array of the right type. However, stand-alone use of the Sigma proofs may not be aware of this peculiarity.

```
pub fn from_bytes(bytes: &[u8]) -> Result<Self,
PubkeyValidityProofVerificationError> {
    let mut chunks = bytes.chunks(UNIT_LEN);
    let Y = ristretto_point_from_optional_slice(chunks.next())?;
    let z = canonical_scalar_from_optional_slice(chunks.next())?;
          Ok(PubkeyValidityProof { Y, z })
}
```

Figure B.1: Public key validity proof is deserialized without checking the length. (zk-sdk/src/sigma_proofs/pubkey_validity.rs)

- **Avoid panic code paths in functions that return a `Result`.** Several functions include a code path that may potentially induce a panic. In particular, figure B.2 shows that the function `verify_and_split_deposit_amount` returns a `Result` but nevertheless calls the `unwrap` function, which may panic. Consider strengthening the robustness of the implementation by restricting code paths that panic and improving explicit error handing.

```rust
/// Verifies that a deposit amount is a 48-bit number and returns the least
/// significant 16 bits and most significant 32 bits of the amount.
#[cfg(feature = "zk-ops")]
pub fn verify_and_split_deposit_amount(amount: u64) -> Result<(u64, u64),
TokenError> {
    if amount > MAXIMUM_DEPOSIT_TRANSFER_AMOUNT {
        return Err(TokenError::MaximumDepositAmountExceeded);
    }
    let deposit_amount_lo = amount & (u16::MAX as u64);
    let deposit_amount_hi = amount.checked_shr(u16::BITS).unwrap();
    Ok((deposit_amount_lo, deposit_amount_hi))
   }
```

Figure B.2: Potential panic code path in a function that returns a result (token-2022/program/src/extension/confidential_transfer/processor.rs)

C. Automated Testing

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

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 repository's 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 repository's root directory. The `--html` flag can be used to output the resulting report as HTML.

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. To run Clippy in pedantic mode, use the command `cargo clippy -- -W clippy::pedantic`.

Semgrep

The community version of 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 ruleset hosted on the Semgrep registry. To use the default configuration, use the command-line argument `--config auto`. Trail of Bits' public ruleset can be used by running `semgrep --config p/trailofbits`.

E. Formal Modeling

During the engagement, we used formal verification tools to model different aspects of the Token-2022 confidential transfer system. We used Tamarin to model an abstract representation of the state machines for confidential transfer, confidential mint and burn, and confidential transfer with fees. We also used Lean to model binding properties of various Fiat-Shamir transforms in Sigma proofs and batch verification procedures. The work and the results obtained are presented in the sections below. The artifacts we generated were produced with the support of an LLM to expedite development during the limited time period.

Modeling with Lean

Lean is an interactive theorem prover and functional programming language. It enables writing precise mathematical statements and their proofs, which are machine-verified.

In this audit, we used Lean as a component in a pipeline to verify the correctness and binding of Fiat-Shamir computations and of batch verification through multiscalar multiplication (MSM). Specifically, we developed a small pipeline that extracts, for each Sigma proof verifier, 1) a transcript trace: the ordered list of transcript operations (domain separators, absorbs, and challenge squeezes), labeled exactly as in the Rust source code; 2) MSM equation structure: the list of scalar expressions (as an AST) and point labels used in the verifier's MSM check; and 3) provenance context: local bindings and proof/public inputs needed to interpret scalar expressions and attribute them to proof fields versus challenges.

Lean then checks—by exhaustive evaluation over these finite extracted structures—that the verifier does not use prover-controlled values "too late" (i.e., after a challenge is derived, in a way that could violate Fiat–Shamir binding). When the check fails (or is intentionally over-approximating), Lean produces violation reports to pinpoint which labels/fields appear late relative to which challenge.

Installing and Running Lean

Follow these instructions to install Lean. An up-to-date version of Rust is required. To extract Rust to Lean, navigate to the extractor code folder and run this command:

```
cargo run --release -- --backend lean --input PATH/TO/sigma_proofs
--output-lean ../path/Generated
```

Then, to run the Lean check, navigate to the "Generated" folder and run `lake build Proofs.AutoScan`.

Rust Extraction Pipeline

At a high level, the Rust extractor parses the Sigma-proof verifier implementations, derives a proof-specific specification from the AST (proof fields, verifier inputs, transcript operations, challenges, bindings, and MSM structure), and emits a Lean file per proof under

`lean/Generated/`…. The extractor is structure-preserving: it aims to serialize the verifier's behavior into concrete Lean definitions that can be checked.

Figure D.1 shows the high-level "spec from AST" shape used by the extractor.

Once extracted, each proof gets a generated Lean module (e.g., `Generated/ZeroCiphertext.lean`) containing the proof/public input structures, `transcriptTrace`, `challengeLabels`, and MSM-related lists.

Figure D.2 shows an excerpt of such output.

```
impl SigmaProofSpec {
    /// Derive a complete spec from an impl block and its struct definition.
    ///
    /// This is the ONLY way to create a `SigmaProofSpec` - ensuring all data comes
from AST.
    pub fn from_impl_and_struct(imp: &ExtractedImpl, struct_def:
Option<&ExtractedStruct>) -> Option<Self> {
        // Only process proof types
        if !imp.target_type.ends_with("Proof") {
            return None;
        }
        // Find verify method
        let verify_method = imp.methods.iter().find(|m| m.name == "verify")?;
        let body = verify_method.body.as_ref()?;
[...]
        // Extract public inputs from verify method parameters
        let public_inputs = extract_public_inputs(&verify_method.params);
        // Extract transcript operations from method body
        let transcript_ops = extract_transcript_ops(body);
        // Analyze challenge structure from transcript ops
        let challenges = analyze_challenge_structure(&transcript_ops);
        // Extract local variable bindings from verify body
        let local_bindings = extract_local_bindings(body);
        // Extract MSM scalars/points (if present) from verify body
        let msm_info = extract_msm_info(body);
        Some(SigmaProofSpec {
            lean_name,
            proof_fields,
            public_inputs,
            challenges,
            transcript_ops,
            local_bindings,
            msm_info,
        })
    }
```

```
}
```

Figure D.1: Rust expression of a Sigma proof specification from the AST

```
structure ZeroCiphertextProof where
  Y_P : RistrettoPoint
  Y_D : RistrettoPoint
  z : Scalar

/-! ## Public inputs (from verify signature) -/

structure ZeroCiphertextPublicInputs where
  elgamal_pubkey : ElGamalPubkey
  ciphertext : ElGamalCiphertext

/-! ## Transcript trace (extracted from verify body) -/

def transcriptTrace : TranscriptTrace := [
  .domainSep "zero_ciphertext_proof_domain_separator",
  .absorb "Y_P",
  .absorb "Y_D",
  .squeeze "c",
  .squeeze "w"
]

/-! ## MSM point labels -/

def msmPointLabels : List String := [
  "P",
  "H",
  "Y_P",
  "D",
  "C",
  "Y_D"
]

def challengeLabels : List String := [
  "c",
  "w"
]
```

Figure D.2: Excerpt of `ZeroCiphertext.lean`

Lean Analysis: Binding Checks and Reports

The core check is a decidable predicate that enforces the following ordering rule (local provenance rule): if a prover-controlled value (a proof field, or a value derived from proof fields) and a challenge both appear within the same MSM scalar expression, then the prover-controlled value must have been absorbed into the transcript before that challenge was squeezed.

This produces a Boolean outcome per proof attesting to safe binding or violations. Separately, we provide optional report modes to aid debugging and triage. Figure D.3 illustrates our definition of safety, which requires that all fields in an MSM value verify the `isSafe` predicate. This predicate, in turn, necessitates that the field is absorbed before a challenge is generated from the transcript.

```
/-- All MSM scalars are safe -/
def allMSMScalarsSafe
    (ctx : ProvenanceContext)
    (msm : MSMExprSpec)
    (trace : TranscriptTrace)
    (challenges : List String) : Prop :=
  ∀ expr ∈ msm.scalarExprs,
    isSafe (computeProvenance ctx expr) trace challenges
```

Figure D.3 Lean definition of transcript safety

For failing proofs, our tool generates violation reports that identify late occurrences of labels that are actually absorbed in the transcript, including relevant local variable names that are absorbed (more directly aligned with "what was committed before challenge X"). For example, the dedicated report module for `PercentageWithCap` (`Proofs/ReportPercentageWithCap.lean`) generates per-challenge late lists for both the current and previous snapshots.

Cryptographic Modeling Scope

The Lean analysis in this appendix serves as a structural binding/provenance check, rather than a computational cryptographic proof. In particular, it does not implement concrete Ristretto arithmetic or hashing to determine whether a label was absorbed before a challenge, or whether an MSM scalar expression depends on prover-controlled data.

The Lean side then runs executable checks (proved via `native_decide`) over these concrete extracted lists. This yields two outputs developers can act on: 1) a pass/fail theorem that the extracted MSM scalar expressions satisfy the Fiat–Shamir ordering rule, and 2) optional, readable "challenge → violations" reports that point to the exact terms/labels requiring inspection.

Results

We ran the Lean binding checker on the extracted Fiat–Shamir transcript traces and MSM scalar expression structure for Token-2022 Sigma proofs. As a representative example, we ran a focused check for `ZeroCiphertext` on both the current code and the older version of the code before commit `8a085e`. And the corresponding build output (abridged) shows the violations: `[z] not absorbed before 'c' | [z] not absorbed before 'w'`.

These results demonstrate that the Lean pipeline can 1) automatically extract proof-specific traces and MSM structure, and 2) reliably surface ordering/binding anomalies with concrete, audit-friendly evidence.

Limitations and Discussion

Our theorem statements are too strong. A reported “late” value is an explicit signal to inspect whether the dependency is real and security-relevant, rather than an automatic proof of exploitability.

The results of our modeling guarantee the ordering/binding structure of Fiat–Shamir usage and MSM dependencies; however, they do not prove the zero-knowledge, completeness, or soundness of the underlying schemes.

The guarantees depend on the extractor faithfully capturing transcript operations, bindings, and MSM structure from the Rust source. The extractor is designed to be structure-preserving, but it has not been formally validated.

Modeling with Tamarin

Tamarin is a model checker that works in the symbolic model. Tamarin takes as input a specification of the protocol and desired security properties, and it builds a set of constraints that are consistent with the system specification but violate the desired properties. It enables modeling of security protocols as state machines and proving of safety properties under the Dolev-Yao adversary model.

In this audit, we used Tamarin to verify authorization and state-transition correctness properties of the Token-2022 confidential transfer extension family—specifically confidential transfer (CT), confidential transfer fee (CTF), and confidential mint/burn (CMB). The model encodes the program's instruction processing as state-transition rules that consume and produce state facts. In the following section, we discuss the result of a simpler model of the confidential transfer system. We provide a more comprehensive model of the system separately, which requires high computational resources to run.

Installing and Running Tamarin

To install Tamarin, refer to the Install Tamarin guide. On macOS and Linux systems with Homebrew, run `brew install tamarin-prover/tap/tamarin-prover`. Tamarin requires significant computational resources to run complex models. We recommend renting a virtual machine with at least 128 GB of RAM.

To run a Tamarin file (e.g., `model.sptyh`), run `tamarin-prover model.sptyh`. We provide additional guidance on running Tamarin on our bigger models in our additional deliverables.

Modeling Cryptographic Schemes

We use symbolic function symbols to represent cryptographic operations and protocol-level abstractions. These are uninterpreted functions whose security properties are captured through equational theories and restrictions rather than computational proofs. For example, the function symbol `pk/1` has arity one and represents a public key encryption scheme where the public key is derived from the input secret key. Since no other equation is provided, a public key is effectively "one-way" because the adversary cannot learn the secret key with the available derivation rules.

Figure D.4 shows a lightweight abstraction of homomorphic operations. For example, the rule `h_add(x, zero_ct(pk)) = x` encodes that adding a zero ciphertext to some arbitrary ciphertext `x` preserves the value of `x`.

```
equations:
  h_add(x, zero_ct(pk)) = x,
  h_add(zero_ct(pk), x) = x,
  h_sub(x, zero_ct(pk)) = x,
  h_sub(h_add(x, y), y) = x
```

Figure D.4: Homomorphic identities

Modeling State Machines

Tamarin models protocols as labeled transition systems where facts represent states and rules define transitions. A rule has the following form:

`[Premises] --[Actions]--> [Conclusions]`

A rule consumes premise facts, emits action facts (observable events), and produces conclusion facts. Tamarin supports linear and persistent facts. Linear facts (e.g., `Account(...)`) are consumed when used and must be explicitly reproduced to persist. Persistent facts (e.g., `!Mint(...)`) remain available indefinitely once created, modeling immutable configuration. In Tamarin, the adversary controls the network via `In(...)` and `Out(...)` facts under the Dolev-Yao model. Security properties are expressed as lemmas over action facts and timestamps. For example, states that every transfer event must coincide with an owner signature event.

`"Transfer(...) @ i ==> UserSigned(...) @ i"`

Tamarin exhaustively searches all possible rule interleavings to verify that these properties hold on all traces (for all-traces lemmas) or to find a witness trace (for exists-trace lemmas).

In our model of the Token-2022 system, we use rules that encode client actions and program actions. Clients are, in general, any entity that emits an instruction that the Token-2022 program processes. Client facts are denoted with a `Tx` prefix. For example, the rule `TxBuilder_BuildTransfer` generates a fresh amount, encrypts it for the source and destination, and outputs the ciphertexts for the adversary-controlled network.

```
rule TxBuilder_BuildTransfer:
  let
    amount = amt48(~amountRaw)
    srcCt = enc_amt(amount, ~r, srcPk)
    dstCt = enc_amt(amount, ~r, dstPk)
  in
  [ !UserKey($Src, srcSk), Account($Src, mintId, srcAccId, srcPk, srcAvail, srcPend,
'true'), Account($Dst, mintId, dstAccId, dstPk, dstAvail, dstPend, dstApproved) ,
Fr(~amountRaw), Fr(~r)
  ]
--[ TransferRequested($Src, $Dst, srcAccId, dstAccId) ]->
  [ !UserKey($Src, srcSk), Out(amount), Out(srcCt), Out(dstCt)
  ]
```

Figure D.5: Transfer builder rule

Program actions are modeled using rules prefixed with `ct`. They consume inputs from the network via `In(...)`, may check authorization via `Sig(...)` facts, and read and update on-chain state facts, emitting action facts that record what occurred.

For example, the `ct_process_transfer` rule in figure D.6 requires a signature, consumes the source `Account` fact, and produces updated account states.

```
rule ct_process_transfer:
  let
    newSrcAvail = h_sub(srcAvail, srcCt)
    newDstPend  = h_add(dstPend, dstCt)
  in
  [ !AccOwner(srcAccId, $Src)
  , !AccMint(srcAccId, mintId)
  , !AccOwner(dstAccId, $Dst)
  , !AccMint(dstAccId, mintId)
  , Account($Src, mintId, srcAccId, srcPk, srcAvail, srcPend, 'true')
  , Account($Dst, mintId, dstAccId, dstPk, dstAvail, dstPend, dstApproved)
  , Sig($Src)
  , In(amount)
  , In(srcCt)
  , In(dstCt)
  ]
--[ Transfer($Src, $Dst, srcAccId, dstAccId, amount)
  , UserSigned($Src)
  ]->
  [ Account($Src, mintId, srcAccId, srcPk, newSrcAvail, srcPend, 'true')
  , Account($Dst, mintId, dstAccId, dstPk, dstAvail, newDstPend, dstApproved)
  , PendingCreditToken(dstAccId)
  ]
```

Figure D.6: Transfer processing rule (`Token22_simple.spthy`)

Properties Verified

This section describes at a high level the lemmas we ran and verified. The full list of lemmas is available in the models we provide separately.

Confidentiality lemma: We verify confidentiality in the `CT_Amount_Confidentiality` lemma. This lemma is implemented as an observational equivalence lemma where an adversary must distinguish between observing an encryption of two different amounts. In spirit, the lemma aims to capture a property akin to the CPA-like confidentiality definition given in definition 5.6 of the confidential payment system specification. The lemma is weaker than the computational notion since Tamarin is a symbolic verifier.

```
// Ensure the model only offers a single challenge event (helps termination in diff
runs).
restriction One_CT_Confidentiality_Challenge:
  "All #i #j.
     CT_Confidentiality_Challenge() @ i
     & CT_Confidentiality_Challenge() @ j
     ==> #i = #j"

rule ct_small_confidentiality_challenge:
  let
    ct = enc_amt(diff(m1, m2), ~r, dstPk)
  in
  [ Fr(~r)
  , In(dstPk)
  , In(<m1, m2>)
  ]
--[ CT_Confidentiality_Challenge() ]->
  [ Out(ct) ]
```

Figure D.7: Confidentiality lemma. For efficiency, the model is restricted to a single challenge.

Reachability lemmas: We define several Lemmas that check the correctness of our models, confirming that the model can reach states of interest. The verified lemmas are `MintBurn_Init_Possible` and `FeeConfig_Init_Possible`.

Correctness and safety lemmas: We ran and verified lemmas that capture expected correctness and safety properties, such as the prevention of unauthorized transfer, in the lemma `No_Unauthorized_Transfer`.

Limitations and Discussions

The verified properties confirm that, under the symbolic Dolev-Yao model, only the account owner can perform operations on their account. The model is non-vacuous, and transfers are confidential in the sense of observational equivalence.

However, our model has the following limitation: cryptographic operations are modeled symbolically, rather than computationally. In particular, although our full model aims to symbolically implement zero-knowledge proofs (based on Fischlin's formalization), our models are not fine-grained enough to capture possible soundness issues.

Slice-based verification: Some properties are verified on minimal slices, not the full model. Using specific flags, the trace can be limited to specific rules. However, such restrictions may miss attacks arising from unexpected interactions between components.

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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn 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

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.