



Anza Token-2022 Confidential Transfer, Blockchain

Security Assessment

January 30, 2026

Prepared for:

Sam Kim

Anza

Prepared by: **Samuel Moelius and Quan Nguyen**

Table of Contents

Table of Contents	1
Project Summary	2
Executive Summary	3
Project Goals	5
Project Targets	6
Project Coverage	7
Automated Testing	8
Codebase Maturity Evaluation	9
Summary of Findings	11
Detailed Findings	12
1. Unused commitments are not verified to be zero	12
2. BatchedRangeProofContext TryInto assumes all used commitments are nonzero	14
3. VecPoly1::eval can panic on malformed structs	16
4. Auditor pubkey validation differs between confidential mint/burn and transfer operations	18
5. from_bytes functions lack length checks	20
6. verify_mint_proof and verify_burn_proof do not handle mixed-mode calls correctly	22
A. Vulnerability Categories	26
B. Code Maturity Categories	28
C. Non-Security-Related Recommendations	30
D. Rust Functions That Were Fuzzed	36
About Trail of Bits	42
Notices and Remarks	43

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:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Samuel Moelius, Consultant **Quan Nguyen**, Consultant
samuel.moelius@trailofbits.com quan.nguyen@trailofbits.com

Project Timeline

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

Date	Event
January 5, 2026	Pre-project kickoff call
January 9, 2026	Status update meeting #1
January 15, 2026	Delivery of report draft
January 20, 2026	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 the token-2022 confidential transfer extensions.

The extensions have three main components:

- The native ZK ElGamal Proof Program performs ZK proof validation for on-chain Solana programs. The ZK ElGamal Proof Program resides in the `anza-xyz/agave` repository.
- The extensions themselves allow confidential minting, burning, and transferring of tokens. To do so, the extensions must be enabled when the token's mint account is created. The extensions reside in the `solana-program/token-2022` repository.
- The ZK-SDK contains cryptographic code used by the previous two repositories. The ZK-SDK resides in the `solana-program/zk-elgamal-proof` repository.

A team of two consultants conducted the review from January 5 to January 14, 2026, for a total of three engineer-weeks of effort. Our testing efforts focused on identifying ways that the token-2022 confidential transfer extensions could cause a loss of funds. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The token-2022 extensions and SDK feature many tests covering different scenarios. However, we uncovered edge cases where the code appears to misbehave. These include more commitments than are required, commitments that are unintentionally zero, unexpected auditor keys, and stray bytes passed to `from_bytes` functions. We recognize that a developer cannot be expected to foresee all possible edge cases. Hence, Anza might consider fuzzing as an additional means of uncovering bugs in the extensions.

The documentation does not provide all the information a developer needs to interact with a confidential-transfer-enabled token. For example, the documentation does not mention instruction introspection, one of the two ways of providing proofs to the extensions. Furthermore, several nonobvious choices were made during the implementation of the extensions. Such choices should be explained in inline comments. This will help auditors reviewing the code and developers who wish to understand how the code works.

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.
- **Expand the project's test.** As explained above, the code behaves incorrectly on edge cases uncovered during the review. We recommend correcting the behavior and expanding the tests to ensure the corrected behavior is preserved. Finally, consider adding fuzzing to your testing strategy to uncover additional misbehaviors.
- **Expand the project's documentation.** Expand the project's external documentation so that arbitrary users can learn how the extensions work. Also, expand the project's inline comments to help auditors and developers understand why the code behaves as it does.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

<i>Severity</i>	<i>Count</i>
High	0
Medium	0
Low	2
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

<i>Category</i>	<i>Count</i>
Data Validation	6

Project Goals

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

- Can the confidential transfer extensions be used to manipulate memory or state in an Agave validator?
- Can the extensions be used to cause loss of funds?
- Can the extensions be used to crash the validator (i.e., cause loss of availability)?
- Does the ZK ElGamal native program respect memory alignment?
- Does the ZK ElGamal native program provide access to memory that has already been cleared, possibly leading to loss of consensus?
- Can any of the above cause gas exhaustion?
- Can cryptographic guarantees be broken at API boundaries?

Project Targets

The engagement involved reviewing and testing the following targets.

agave

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

zk-elgamal-proof

Repository	https://github.com/solana-program/zk-elgamal-proof
Version	a57ab5f023d8af617f0bc2293f51e1a18e025efb
Type	Rust
Platform	POSIX/Solana

token-2022

Repository	https://github.com/solana-program/token-2022
Version	a4fd7036fb6b47df68fbda50671365066a90ded1
Type	Rust
Platform	Solana

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:

- **Documentation review.** We reviewed the **Confidential Transfer** documentation and rustdoc comments for accuracy and completeness.
- **Test coverage review.** We verified that each of the repository's tests passes. We also computed code coverage for the zk-sdk using **cargo-llvm-cov**.
- **Static analysis.** We ran **Clippy** over each of the three repositories with the pedantic lints enabled and reviewed the results.
- **Fuzzing.** We fuzzed many functions and methods in the zk-sdk and triaged the results.
- **Manual review.** We manually reviewed the code in each of the three repositories.

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:

- The scope was limited to the confidential features in token-2022 (confidential transfer, confidential transfer fee, confidential mint/burn, and related registry interactions) and the zk-sdk and zk-elgama1-proof native programs. Other parts of the token-2022 repository were not reviewed.
- This was a blockchain review. We did not perform a cryptographic security analysis of the underlying proofs (e.g., soundness or zero-knowledge properties).

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.

Fuzzing

We used `test-fuzz` and `cargo-afl` to fuzz many functions and methods in the `zk-sdk`. Our fuzzing efforts resulted in only one finding ([TOB-ACTBC-3](#)). Note that we did not report failures or panics that appeared uninteresting. Also, we would like to emphasize that panics observed in internal functions are not necessarily triggerable by external inputs.

Additional details can be found in [appendix D](#).

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 found one instance where a call to <code>checked_add</code> is followed by a call to <code>unwrap</code> , where an error should likely be propagated to the caller. Otherwise, <code>checked</code> arithmetic seems to be used in appropriate places.	Satisfactory
Auditing	The code features few uses of <code>msg!</code> , which can make it hard to debug transaction failures. On the other hand, the <code>token-2022</code> code has fields (e.g., <code>actual_</code> and <code>expected_pending_balance_credit_counter</code>) to make monitoring easier.	Moderate
Authentication / Access Controls	This category is not applicable because authentication and access controls are handled by ZK proofs, which fell under the separate cryptography review.	Not Applicable
Complexity Management	The code is well structured and modular. The code has at least one outdated dependency (<code>merlin</code>). However, we have no alternatives to suggest.	Satisfactory
Cryptography and Key Management	The <code>zk-sdk</code> performs hashing. However, it does not feature any obvious cryptographic-related vulnerabilities (e.g., lack of domain separation).	Satisfactory
Decentralization	Decentralization is not applicable to the code under review, as it is a set of libraries that can be called by anyone.	Not Applicable
Documentation	The project's external documentation does not cover all aspects of the protocol that one needs to use it. For example, the documentation should mention that instruction introspection can be used to verify proofs.	Moderate

	<p>Furthermore, several aspects of the implementation are nonobvious and should be documented with inline comments. Examples include why auditor ciphertexts are duplicated in instruction data, what <code>decryptable_supply/decryptable_balance</code> represent, why they can lag behind the true supply and balance, and how the expected and actual pending balance credit counters help clients detect stale state. The lack of documentation for these aspects of the code makes audits and integrations harder than necessary.</p>	
Low-Level Manipulation	<p>Many of the findings are low-level in nature. For example, several findings involve improper zero checks or failure to check for unnecessary bytes at the end of input. Such bugs might be found through more thorough testing.</p>	Moderate
Testing and Verification	<p>The confidential transfer extensions and zk-sdk have many tests. However, some of the bugs found during this review might have been uncovered through more thorough testing. The code would also benefit from more advanced testing techniques, such as fuzzing.</p>	Moderate
Transaction Ordering	<p>Anza made deliberate design decisions to mitigate front-running risks where practical. For cases like withdrawing withheld tokens, where full front-running prevention would overly complicate the protocol, the documentation suggests alternative flows as mitigation. Our review identified no exploitable transaction ordering vulnerabilities.</p>	Satisfactory

Summary of Findings

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

ID	Title	Type	Severity
1	Unused commitments are not verified to be zero	Data Validation	Informational
2	BatchedRangeProofContext TryInto assumes all used commitments are nonzero	Data Validation	Low
3	VecPoly1::eval can panic on malformed structs	Data Validation	Informational
4	Auditor pubkey validation differs between confidential mint/burn and transfer operations	Data Validation	Informational
5	from_bytes functions lack length checks	Data Validation	Informational
6	verify_mint_proof and verify_burn_proof do not handle mixed-mode calls correctly	Data Validation	Low

Detailed Findings

1. Unused commitments are not verified to be zero

Severity: Informational

Difficulty: Undetermined

Type: Data Validation

Finding ID: TOB-ACTBC-1

Target: token-2022/confidential/proof-extraction/src/{mint.rs, burn.rs, transfer.rs}

Description

Several token-2022 instructions use `zip`, which considers only as many elements as the shorter of its two arguments. Therefore, the functions ignore subsequent elements. The functions should include additional validation to ensure those elements are zero.

An example of the problem with respect to the confidential mint instruction appears in figure 1.1. The problem also affects the burn and transfer operations. Note that the same pattern appears in `transfer_with_fee.rs`. However, all commitments there are used, so it is not vulnerable.

```
82     let expected_commitments = [  
83         *new_supply_commitment,  
84         mint_amount_commitment_lo,  
85         mint_amount_commitment_hi,  
86         // we don't care about the padding commitment, so ignore it  
87     ];  
88  
89     // range proof context always contains 8 commitments and therefore,  
90     // this check will verify equality of all expected commitments  
91     // (`zip` will not be short-circuited)  
92     if !range_proof_commitments  
93         .iter()  
94         .zip(expected_commitments.iter())  
95         .all(|(proof_commitment, expected_commitment)| proof_commitment ==  
96             expected_commitment)  
97     {  
98         return Err(TokenProofExtractionError::PedersenCommitmentMismatch);  
99     }
```

Figure 1.1: Excerpt of `verify_and_extract`. Since `expected_commitments` has length three, the function ignores all `range_proof_commitments` elements after the third.
(token-2022/confidential/proof-extraction/src/mint.rs#L82-L98)

Exploit Scenario

Alice writes code that accidentally appends the additional commitments to her mint instructions. The bug goes unnoticed because the on-chain code does not check that unused commitments are zero.

Recommendations

Short term, add code to `verify_and_extract` (figure 1.1) to check that unused commitments are zero. Do the same for the versions of these functions for `burn` and `transfer`. This will help to identify bugs in off-chain code.

Long term, test obscure edge cases such as instructions that contain commitments beyond the ones required. Doing so can help to reveal problems such as the one described here.

2. BatchedRangeProofContext TryInto assumes all used commitments are nonzero

Severity: Low

Difficulty: High

Type: Data Validation

Finding ID: TOB-ACTBC-2

Target: zk-sdk/src/zk_elgamal_proof_program/proof_data/
batched_range_proof/mod.rs

Description

BatchedRangeProofContext has a TryInto implementation to convert the struct into a vector of commitments and a vector of bit lengths. However, the implementation assumes that all used commitments are nonzero. If a BatchedRangeProofContext contains a zero commitment, the implementation will return vectors that are too short.

```
112     impl TryInto<(Vec<PedersenCommitment>, Vec<usize>)> for
BatchedRangeProofContext {
113         type Error = ProofVerificationError;
114
115         fn try_into(self) -> Result<(Vec<PedersenCommitment>, Vec<usize>),
Self::Error> {
116             let commitments = self
117                 .commitments
118                 .into_iter()
119                 .take_while(|commitment| *commitment !=
PodPedersenCommitment::zeroed())
120                 .map(|commitment| commitment.try_into())
121                 .collect:::<Result<Vec<PedersenCommitment>, _>>()
122                 .map_err(|_| ProofVerificationError::ProofContext)?;
123
124             let bit_lengths: Vec<_> = self
125                 .bit_lengths
126                 .into_iter()
127                 .take(commitments.len())
128                 .map(|bit_length| bit_length as usize)
129                 .collect();
130
131             Ok((commitments, bit_lengths))
132         }
133     }
```

Figure 2.1: TryInto implementation for BatchedRangeProofContext
(zk-elgamal-proof/zk-sdk/src/zk_elgamal_proof_program/proof_data/batched_
range_proof/mod.rs#L112-L133)

Exploit Scenario

Alice writes code to generate `BatchedRangeProofContext`. Her implementation constructs the structs directly rather than using `BatchedRangeProofContext::new`, thereby avoiding the function's validation. Alice's code contains a bug that causes some of her struct's commitments to be zero. When Alice calls the code in figure 2.2, the resulting vectors are too short.

Recommendations

Short term, use nonzero bit lengths to determine how long the resulting vectors should be. This will prevent problems caused by commitments that are unintentionally zero.

Long term, test obscure edge cases such as commitments that are unintentionally zero. Doing so can help to reveal problems such as the one described here.

3. VecPoly1::eval can panic on malformed structs

Severity: Informational

Difficulty: High

Type: Data Validation

Finding ID: TOB-ACTBC-3

Target: zk-sdk/src/range_proof/util.rs

Description

A VecPoly1 contains two vectors (figure 3.1). However, the eval function only considers the length of the first (figure 3.2). If eval is called on a VecPoly1 that is manually constructed (e.g., not the result of VecPoly1::new), eval could perform an out-of-bounds array access.

```
17 pub struct VecPoly1(pub Vec<Scalar>, pub Vec<Scalar>);
```

Figure 3.1: Definition of VecPoly1
(zk-elgamal-proof/zk-sdk/src/range_proof/util.rs#L17)

```
51 pub fn eval(&self, x: Scalar) -> Vec<Scalar> {  
52     let n = self.0.len();  
53     let mut out = vec![Scalar::ZERO; n];  
54     #[allow(clippy::needless_range_loop)]  
55     for i in 0..n {  
56         out[i] = self.0[i] + self.1[i] * x;  
57     }  
58     out  
59 }
```

Figure 3.2: If a VecPoly1 is constructed manually, its vectors could have different lengths, and access to the second vector could go out of bounds.

(zk-elgamal-proof/zk-sdk/src/range_proof/util.rs#L51-L59)

Exploit Scenario

Alice, an Anza developer, is working on changes to the range_proof module. Alice's changes require additional VecPoly1s that she constructs manually, thinking use of VecPoly1::new would be overkill. An obscure edge case in Alice's code constructs a VecPoly1 with vectors of different sizes. Calling eval on the VecPoly1 causes Alice's code to panic.

Recommendations

Short term, add validation to ensure that VecPoly1 vectors have the same size. This will help catch malformed VecPoly1s with vectors of different sizes.

Long term, consider adding fuzzing to your testing strategy. This bug was found with fuzzing.

4. Auditor pubkey validation differs between confidential mint/burn and transfer operations

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ACTBC-4

Target:

token-2022/program/src/extension/confidential_mint_burn/processor.rs

Description

The confidential operations in token-2022 handle auditor pubkey validation inconsistently. Confidential mint and burn operations only verify the auditor pubkey when the mint has one explicitly configured, while confidential transfer operations always enforce an exact match regardless of configuration state.

In the `process_confidential_mint` and `process_confidential_burn` functions, the auditor pubkey check is conditional. The code first converts the stored pubkey to an `Option` and performs validation only if the `Some` variant is present.

```
if let Some(auditor_pubkey) =  
Option::<PodElGamalPubkey>::from(auditor_elgamal_pubkey) {  
    if auditor_pubkey != proof_context.mint_pubkeys.auditor {  
        return Err(ProgramError::InvalidInstructionData);  
    }  
}
```

Figure 4.1: Conditional auditor check in confidential mint/burn
([confidential_mint_burn/processor.rs#L224-L228](#))

In contrast, the `process_transfer` function and the transfer-with-fee path use the `.equals()` method, which always enforces that the proof's auditor pubkey matches the mint's configured value.

```
if !confidential_transfer_mint  
    .auditor_elgamal_pubkey  
    .equals(&proof_context.transfer_pubkeys.auditor)  
{  
    return Err(TokenError::ConfidentialTransferElGamalPubkeyMismatch.into());  
}
```

Figure 4.2: Unconditional auditor check in confidential transfer
([confidential_transfer/processor.rs#L683-L688](#))

This behavioral difference means that when no auditor is configured on the mint, confidential mint/burn operations accept proofs generated with any arbitrary auditor pubkey, while confidential transfers require the proof to explicitly encode a zeroed/none auditor. Although this does not create an exploitable vulnerability, since the proof still binds to the pubkey used during generation, the inconsistency could lead to confusion for integrators or subtle interoperability issues when constructing proofs across different operation types.

Recommendations

Short term, update the auditor pubkey validation in `process_confidential_mint` and `process_confidential_burn` to use the same `.equals()` pattern as confidential transfer.

Long term, test obscure edge cases such as passing unexpected auditor keys. Doing so can help to reveal inconsistencies such as the one described here.

5. from_bytes functions lack length checks

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-ACTBC-5

Target: zk-sdk/src/sigma_proofs/{ciphertext_ciphertext_equality.rs, ciphertext_commitment_equality.rs, pubkey_validity.rs, percentage_with_cap.rs, zero_ciphertext.rs, grouped_ciphertext_validity/{handles_2.rs, handles_3.rs}}

Description

The implementation of `CiphertextCiphertextEqualityProof::from_bytes` appears in figure 5.1. The function lacks a length check, allowing stray bytes to appear at the end of the input. The lack of a length check could allow bugs in off-chain code to go unnoticed.

```
280     pub fn from_bytes(bytes: &[u8]) -> Result<Self,  
EqualityProofVerificationError> {  
281         let mut chunks = bytes.chunks(UNIT_LEN);  
282  
283         let Y_0 = ristretto_point_from_optional_slice(chunks.next())?;  
284         let Y_1 = ristretto_point_from_optional_slice(chunks.next())?;  
285         let Y_2 = ristretto_point_from_optional_slice(chunks.next())?;  
286         let Y_3 = ristretto_point_from_optional_slice(chunks.next())?;  
287         let z_s = canonical_scalar_from_optional_slice(chunks.next())?;  
288         let z_x = canonical_scalar_from_optional_slice(chunks.next())?;  
289         let z_r = canonical_scalar_from_optional_slice(chunks.next())?;  
290  
291         Ok(CiphertextCiphertextEqualityProof {  
292             Y_0,  
293             Y_1,  
294             Y_2,  
295             Y_3,  
296             z_s,  
297             z_x,  
298             z_r,  
299         })  
300     }
```

Figure 5.1: Implementation of `CiphertextCiphertextEqualityProof::from_bytes`, which lacks a length check

(zk-elgamal-proof/zk-sdk/src/sigma_proofs/ciphertext_ciphertext_equality.rs#L280-L300)

The following are the seven `from_bytes` functions that lack length checks:

- `CiphertextCiphertextEqualityProof::from_bytes`

- `CiphertextCommitmentEqualityProof::from_bytes`
- `GroupedCiphertext2HandlesValidityProof::from_bytes`
- `GroupedCiphertext3HandlesValidityProof::from_bytes`
- `PubkeyValidityProof::from_bytes`
- `PercentageWithCapProof::from_bytes`
- `ZeroCiphertextProof::from_bytes`

Exploit Scenario

Alice writes code that accidentally appends unnecessary bytes to a slice she intends to convert to a `CiphertextCiphertextEqualityProof`. The bug goes unnoticed because the on-chain code does not check that the number of bytes passed is the number of bytes expected.

Recommendations

Short term, add length checks to each of the `from_bytes` implementations listed above. This will help to catch bugs in off-chain code that accidentally append unnecessary bytes to slices passed to those functions.

Long term, test obscure edge cases such as `from_bytes` inputs that contain more bytes than are needed. Doing so can help to reveal problems such as the one described here.

6. verify_mint_proof and verify_burn_proof do not handle mixed-mode calls correctly

Severity: Low

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-ACTBC-6

Target: program/src/extension/confidential_mint_burn/verify_proof.rs

Description

The `verify_mint_proof` and `verify_burn_proof` functions incorrectly handle mixed proof locations, where some proofs come from context state accounts and others use instruction introspection (figures 6.1 and 6.2). (Compare these functions to `verify_transfer_proof`, shown in figure 6.3, which does not exhibit the bug.)

```
18  /// Verify zero-knowledge proofs needed for a `ConfidentialMint` instruction
19  /// and return the corresponding proof context information.
20  #[cfg(feature = "zk-ops")]
21  pub fn verify_mint_proof(
22      account_info_iter: &mut Iter<'_, AccountInfo<'_>>,
23      equality_proof_instruction_offset: i8,
24      ciphertext_validity_proof_instruction_offset: i8,
25      range_proof_instruction_offset: i8,
26  ) -> Result<MintProofContext, ProgramError> {
27      let sysvar_account_info = if equality_proof_instruction_offset != 0 {
28          Some(next_account_info(account_info_iter)?)
29      } else {
30          None
31      };
```

Figure 6.1: The `verify_mint_proof` function, which checks only whether `equality_proof_instruction_offset` is nonzero
(`token-2022/program/src/extension/confidential_mint_burn/verify_proof.rs#L18-L31`)

```
66  /// Verify zero-knowledge proofs needed for a `ConfidentialBurn` instruction
67  /// and return the corresponding proof context information.
68  #[cfg(feature = "zk-ops")]
69  pub fn verify_burn_proof(
70      account_info_iter: &mut Iter<'_, AccountInfo<'_>>,
71      equality_proof_instruction_offset: i8,
72      ciphertext_validity_proof_instruction_offset: i8,
73      range_proof_instruction_offset: i8,
74  ) -> Result<BurnProofContext, ProgramError> {
75      let sysvar_account_info = if equality_proof_instruction_offset != 0 {
76          Some(next_account_info(account_info_iter)?)
```

```

77     } else {
78         None
79     };

```

Figure 6.2: The `verify_burn_proof` function, which exhibits the same bug as `verify_mint_proof`

([token-2022/program/src/extension/confidential_mint_burn/verify_proof.rs#L66-L79](#))

```

56    /// Verify zero-knowledge proof needed for a `Transfer` instruction without
fee
57    /// and return the corresponding proof context.
58    #[cfg(feature = "zk-ops")]
59    pub fn verify_transfer_proof(
60        account_info_iter: &mut Iter<AccountInfo>,
61        equality_proof_instruction_offset: i64,
62        ciphertext_validity_proof_instruction_offset: i64,
63        range_proof_instruction_offset: i64,
64    ) -> Result<TransferProofContext, ProgramError> {
65        let sysvar_account_info = if equality_proof_instruction_offset != 0
66            || ciphertext_validity_proof_instruction_offset != 0
67            || range_proof_instruction_offset != 0
68        {
69            Some(next_account_info(account_info_iter)?)
70        } else {
71            None
72        };

```

Figure 6.3: The `verify_transfer_proof` function, which does not exhibit the same bug as `verify_mint_proof` and `verify_burn_proof`

([token-2022/program/src/extension/confidential_transfer/verify_proof.rs#L56-L72](#))

The issue occurs when `equality_proof_instruction_offset` is zero but `ciphertext_validity_proof_instruction_offset` and `range_proof_instruction_offset` are nonzero. In that case, `verify_proof` will not attempt to fetch the sysvar account. However, `verify_and_extract_context` (figure 6.4) will do the following:

- For the equality proof, it will execute the then case, expecting the next account to be a context state account.
- For the ciphertext validity proof, it will execute the else case, expecting the next account to be a sysvar account.
- For the range proof, it will execute the else case, expecting the next account to be another sysvar account.

Importantly, this behavior does not match what `verify_transfer_proof` would do (i.e., expect a single sysvar account followed by a context state account).

```

71     /// Verify zero-knowledge proof and return the corresponding proof context.
72     pub fn verify_and_extract_context<'a, T: Pod + ZkProofData<U>, U: Pod>(
73         account_info_iter: &mut Iter<'_, AccountInfo<'a>>,
74         proof_instruction_offset: i64,
75         sysvar_account_info: Option<&'_ AccountInfo<'a>>,
76     ) -> Result<U, ProgramError> {
77         if proof_instruction_offset == 0 {
78             // interpret `account_info` as a context state account
79             let context_state_account_info =
next_account_info(account_info_iter)?;
80
check_zk_elgamal_proof_program_account(context_state_account_info.owner)?;
81             let context_state_account_data =
context_state_account_info.data.borrow();
82             let context_state =
pod_from_bytes::<ProofContextState<U>>(&context_state_account_data)?;
83
84             if context_state.proof_type != T::PROOF_TYPE.into() {
85                 return Err(ProgramError::InvalidInstructionData);
86             }
87
88             Ok(context_state.proof_context)
89         } else {
90             // if sysvar account is not provided, then get the sysvar account
91             let sysvar_account_info = if let Some(sysvar_account_info) =
sysvar_account_info {
92                 sysvar_account_info
93             } else {
94                 next_account_info(account_info_iter)?
95             };
96             let zkp_instruction =
97                 get_instruction_relative(proof_instruction_offset,
sysvar_account_info)?;
98             let expected_proof_type =
zkp_proof_type_to_instruction(T::PROOF_TYPE)?;
99             Ok(decode_proof_instruction_context::<T, U>(
100                 expected_proof_type,
101                 &zkp_instruction,
102             )?)
103         }
104     }

```

Figure 6.4: The `verify_and_extract_context` function, which is affected by the bug in `verify_proof`

([token-2022/confidential/proof-extraction/src/instruction.rs#L71-L104](#))

Exploit Scenario

Alice writes code to interact with tokens that use confidential transfer extensions. Her code works fine when she tests it with confidential transfers. However, when her code is used on

real deployments with confidential mints, the code behaves incorrectly because it expects accounts that differ from the ones passed. Eventually, the bug is fixed. However, following the fix, the accounts that were passed before the fix no longer work.

Recommendations

Short term, update the offset checks in figures 6.1 and 6.2 to check whether any of the offsets are nonzero. If any of them are, fetch the sysvar account at the start of the instruction. Doing so will save developers from confusion over account orders.

Long term, fetch all accounts at the start of an instruction, rather than conditionally fetching accounts in nested functions. Doing so will avoid problems where it becomes unclear whether an account has already been fetched.

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
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
Transaction Ordering	The system's resistance to transaction-ordering attacks

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 does not apply 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. Non-Security-Related 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.

- Have `add_vec` (figure C.1) return an error rather than panic.

```
105    /// Computes the component-wise sum of two vectors of scalars.
106    /// Panics if the lengths of the vectors two do not match.
107    pub fn add_vec(a: &[Scalar], b: &[Scalar]) -> Vec<Scalar> {
108        if a.len() != b.len() {
109            panic!("lengths of vectors don't match for vector addition");
110        }
```

Figure C.1: Excerpt of `add_vec`. The function panics rather than return an error.
([zk-elgamal-proof/zk-sdk/src/range_proof/util.rs#L105-L110](#))

- Have the functions shown in figures C.2 and C.3 return an error when `s` is zero.

```
58    pub fn new(elgamal_keypair: &ElGamalKeypair, transcript: &mut Transcript) ->
Self {
59        ...
65        assert!(s != &Scalar::ZERO);
```

Figure C.2: Assertion that `s` is not zero
([zk-elgamal-proof/zk-sdk/src/sigma_proofs/pubkey_validity.rs#L58-L65](#))

```
345    pub fn new(secret: &ElGamalSecretKey) -> Self {
346        let s = &secret.0;
347        assert!(s != &Scalar::ZERO);
```

Figure C.3: Assertion that `s` is not zero
([zk-elgamal-proof/zk-sdk/src/encryption/elgamal.rs#L345-L347](#))

- Eliminate the unnecessary braces on lines 170 and 197 in figure C.4.

```
169    if let Some(num_threads) = self.num_threads {
170        {
171            let mut starting_point = self.target;
172            ...
192            handles
193                .into_iter()
194                .map_while(|h| h.join().ok())
195                .find(|x| x.is_some())
196                .flatten()
197        }
198    } else {
```

Figure C.4: Code with unnecessary braces
(zk-elgamal-proof/zk-sdk/src/encryption/discrete_log.rs#L169-L198)

- Eliminate the typographical errors shown in figures C.5 through C.9.

```
95    /// This function exists for applications where a user may not wish to
maintain a Solana signer
96    /// and an authenticated encryption key separately. Instead, a user can
derive the ElGamal
97    /// keypair on-the-fly whenever encryption/decryption is needed.
```

Figure C.5: “encyrtion” should be “encryption”.
(zk-elgamal-proof/zk-sdk/src/encryption/auth_encryption.rs#L95-L97)

```
218    let second_keyapir = ElGamalKeypair::new_rand();
219    let second_pubkey = second_keyapir.pubkey();
```

Figure C.6: “keyapir” should be “keypair”.
(zk-elgamal-proof/zk-sdk/src/sigma_proofs/batched_grouped_ciphertext_validity/handles_3.rs#L218-L219)

```
265    // homomorphically compute the openings for A + x*S
```

Figure C.7: “compute” should be “compute”
(zk-elgamal-proof/zk-sdk/src/range_proof/mod.rs#L265)

```
136    // if a payer account for reallocation is provided, then reallocate
```

Figure C.8: “realcation” should be “reallocation”.
(token-2022/program/src/extension/confidential_transfer/processor.rs#L136)

```
726    let fee_ciphertext_validity_proof_insruccion_offset =
```

Figure C.9: “insruccion” should be “instruction”.
(token-2022/program/src/extension/confidential_transfer/processor.rs#L726)

- In figure C.10, change second_pubkey to first_pubkey.

```
412    let first_handle = second_pubkey.decrypt_handle(&opening);
```

Figure C.10: Given that the variable assigned to is first_handle, the pubkey used should likely be first_pubkey.
(zk-elgamal-proof/zk-sdk/src/sigma_proofs/grouped_ciphertext_validity/handles_3.rs#L412)

- Change the comment in figure C.11 to a Base64-encoded Pedersen commitment, rather than an ElGamal public key.

```
18    /// Maximum length of a base64 encoded ElGamal public key
```



```
19     const PEDERSEN_COMMITMENT_MAX_BASE64_LEN: usize = 44;
```

Figure C.11: Since the constant concerns Pedersen commitments, the comment likely should as well. ([zk-elgamal-proof/zk-sdk/src/encryption/pod/pedersen.rs#L18-L19](#))

- **Generalize the comment shown in figure C.12 so that it is not specific to two-handle ciphertexts.** The macro in which the comment appears is applied to three-handle ciphertexts as well.

```
30     // `GROUPED_ELGAMAL_CIPHERTEXT_2_HANDLES` guaranteed to be at least
    `PEDERSEN_COMMITMENT_LEN`
31     let commitment = self.0[..PEDERSEN_COMMITMENT_LEN].try_into().unwrap();
```

Figure C.12: Comment that seems to incorrectly refer to two-handle ciphertexts ([zk-elgamal-proof/zk-sdk/src/encryption/pod/grouped_elgamal.rs#L30-L31](#))

- **In figure C.13, change “vectors two” to “two vectors”.**

```
106     /// Panics if the lengths of the vectors two do not match.
```

Figure C.13: “vectors two” should be “two vectors”.
([zk-elgamal-proof/zk-sdk/src/range_proof/util.rs#L106](#))

- **In figure C.14, change `commitment_addition` to `commitment_subtraction`.** Note that there is a `commitment_addition` variable used earlier in the file. Hence, this appears to be a copy-and-paste error.

```
306     let commitment_addition = Pedersen::with(amount_0 - amount_1, &(opening_0 -
opening_1));
```

Figure C.14: Since the value assigned to the variable results from subtraction, subtraction should likely be in the variable’s name.

([zk-elgamal-proof/zk-sdk/src/encryption/pedersen.rs#L306](#))

- **Add a `ConfidentialMintBurn` extension check to `process_burn` to match the existing check in `process_mint_to` (figure C.15).** This will ensure symmetric handling of supply changes when confidential mint/burn is enabled.

```
1026     if mint.get_extension::<ConfidentialMintBurn>().is_ok() {
1027         return Err(TokenError::IllegalMintBurnConversion.into());
1028     }
```

Figure C.15: The check present in the `process_mint_to` function ([token-2022/program/src/processor.rs#1026-1028](#)) should also be added to `process_burn`.

- **Rename the doc comment section labels (L423-L455) in `TransferWithFee` from “Transfer without fee” and “Transfer with fee” to “Single owner/delegate” and “Multisignature owner/delegate”.** The current labels are

misleading because both sections describe the `TransferWithFee` instruction, whereas the actual distinction lies in the authority type, not the presence of a fee.

- In figure C.16, change `VerifyPercentageWithFee` to `VerifyPercentageWithCap`. The proof instruction is named `VerifyPercentageWithCap` in `zk-sdk`, and the code uses `PercentageWithCapProofData / ProofInstruction::VerifyPercentageWithCap`.

```
411    /// - `VerifyPercentageWithFee`
```

Figure C.16: Incorrect proof instruction name in `TransferWithFee` documentation
([token-2022/interface/src/extension/confidential_transfer/instruction.rs#L411](#))

- In figure C.17, fix the logical error in `try_combine_lo_hi_u64`. On line 62, `checked_add(amount_hi)` should be `checked_add(amount_lo)`. The function is currently deprecated and unused, but should be corrected if it is ever re-enabled.

```
55    pub fn try_combine_lo_hi_u64(amount_lo: u64, amount_hi: u64, bit_length:
usize) -> Option<u64> {
56        match bit_length {
57            0 => Some(amount_hi),
58            1..=63 => {
59                // shifts are safe as long as `bit_length` < 64
60                amount_hi
61                    .checked_shl(bit_length as u32)?
62                    .checked_add(amount_hi)
63            }
64            64 => Some(amount_lo),
65            _ => None,
66        }
67    }
```

Figure C.17: `amount_hi` incorrectly used in place of `amount_lo`
([token-2022/confidential/proof-generation/src/lib.rs#L55-L67](#))

- Correct the comment shown in figure C.18. “Panics” should be “Errors”.

```
115    /// # Panics
116    /// This function will panic if the `openings` vector does not contain the
same number
117    /// of elements as the `amounts` and `bit_lengths` vectors.
```

Figure C.18: “Panics” should be “Errors”.
([zk-elgamal-proof/zk-sdk/src/range_proof/mod.rs#L115-L117](#))

- Remove the `n` argument from the `G` and `H` functions, shown in figure C.19. The argument is not checked against the length of the vector being iterated over. Thus,

passing in too large a value could result in an out-of-bounds array access. Also, limiting to the first n elements can be achieved with an iterator's `take` method.

```
137    /// Returns an iterator over the first `n` **G** generators.
138    #[allow(non_snake_case)]
139    pub(crate) fn G(&self, n: usize) -> impl Iterator<Item = &RistrettoPoint> {
140        GensIter {
141            array: &self.G_vec,
142            n,
143            gen_idx: 0,
144        }
145    }
146
147    /// Returns an iterator over the first `n` **H** generators.
148    #[allow(non_snake_case)]
149    pub(crate) fn H(&self, n: usize) -> impl Iterator<Item = &RistrettoPoint> {
150        GensIter {
151            array: &self.H_vec,
152            n,
153            gen_idx: 0,
154        }
155    }
```

Figure C.19: Two functions with an argument n that should be removed
(zk-elgamal-proof/zk-sdk/src/range_proof/generators.rs#L137–L155)

- **Correct the comment shown in figure C.20.** “supply” should be “balance”.

```
212    /// Compute the new decryptable supply.
213    pub fn new_decryptable_balance(
```

Figure C.20: “supply” should be “balance”.

(token-2022/program/src/extension/confidential_mint_burn/account_info.rsL
#212–L213)

- **Change the unwrap shown in figure C.21 to `ok_or`, and pass some appropriate error.**

```
137    let new_decrypted_available_balance = current_available_balance
138        .checked_add(pending_balance)
139        .unwrap(); // total balance cannot exceed `u64`
```

Figure C.21: `unwrap` that should be an `ok_or(. . .)` for some appropriate error
(token-2022/program/src/extension/confidential_transfer/account_info.rs#L
137–L139)

- **Validate that `new_supply_elgamal_pubkey` matches on-chain behavior.** As shown in figure C.22, `RotateSupplyElGamalPubkeyData` includes `new_supply_elgamal_pubkey`, but `process_rotate_supply_elgamal_pubkey` ignores it and instead uses `proof_context.second_pubkey`.

```

216     pub struct RotateSupplyElGamalPubkeyData {
217         /// The new ElGamal pubkey for supply encryption
218         #[cfg_attr(feature = "serde", serde(with = "elgamalpubkey_fromstr"))]
219         pub new_supply_elgamal_pubkey: PodElGamalPubkey,
220         /// The location of the
221         /// `ProofInstruction::VerifyCiphertextCiphertextEquality` instruction
222         /// relative to the `RotateSupplyElGamalPubkey` instruction in the
transaction
223         pub proof_instruction_offset: i8,
224     }

```

Figure C.22: RotateSupplyElGamalPubkeyData includes new_supply_elgamal_pubkey.
(token-2022/interface/src/extension/confidential_mint_burn/instruction.rs
#L216–L224)

D. Rust Functions That Were Fuzzed

We used `test-fuzz` and `cargo-af1` to fuzz several zk-sdk functions and methods. This appendix contains a complete list of all functions that we fuzzed.

In total, we fuzzed 112 functions and found panics in 13 of them. The table below contains the list of functions that we fuzzed. Specifically, the first column gives the name of the function and the second gives the panic that resulted, if any. If the panic is mentioned in this report, we indicate its location. If the panic was considered uninteresting, no location is given.

To emphasize a point made earlier in the report, finding a panic in an intermediate function does not imply the function could panic from user-provided input.

Function	Panic(s) Found
<code>solana_zk_sdk::encryption::auth_encryption::ae_ciphertext_decrypt</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_ciphertext_from_bytes</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_ciphertext_to_bytes</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_decrypt</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_encrypt</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_from</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_from_seed</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_from_seed_phrase_and_passphrase</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_new_from_signature</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_new_rand</code>	
<code>solana_zk_sdk::encryption::auth_encryption::ae_key_seed_from_signature</code>	
<code>solana_zk_sdk::encryption::auth_encryption::authenticated_encryption_decrypt</code>	
<code>solana_zk_sdk::encryption::auth_encryption::authenticated_encryption_encrypt</code>	

solana_zk_sdk::encryption::auth_encryption::authenticated_encryption_keygen	
solana_zk_sdk::encryption::auth_encryption::from	
solana_zk_sdk::encryption::discrete_log::decode_u32_precomputation	
solana_zk_sdk::encryption::discrete_log::discrete_log_decode_range	
solana_zk_sdk::encryption::discrete_log::discrete_log_decode_u32	
solana_zk_sdk::encryption::discrete_log::discrete_log_new_for_g	
solana_zk_sdk::encryption::discrete_log::discrete_log_num_threads	
solana_zk_sdk::encryption::discrete_log::discrete_log_set_compression_batch_size	
solana_zk_sdk::encryption::discrete_log::ristretto_iterator_new	
solana_zk_sdk::encryption::elgamal::decrypt_handle_from_bytes	
solana_zk_sdk::encryption::elgamal::decrypt_handle_new	
solana_zk_sdk::encryption::elgamal::decrypt_handle_to_bytes	
solana_zk_sdk::encryption::elgamal::el_gamal_keypair_encodable_pubkey	
solana_zk_sdk::encryption::elgamal::el_gamal_keypair_from_seed	
solana_zk_sdk::encryption::elgamal::el_gamal_keypair_from_seed_phrase_and_passphrase	
solana_zk_sdk::encryption::elgamal::el_gamal_keypair_new_rand	
solana_zk_sdk::encryption::elgamal::el_gamal_keypair_pubkey_owned	
solana_zk_sdk::encryption::elgamal::el_gamal_pubkey_encrypt_u64	
solana_zk_sdk::encryption::elgamal::el_gamal_pubkey_encrypt_with_u64	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_ct_eq	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_decrypt	

solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_decrypt_u32	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_eq	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_from	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_from_seed	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_from_seed_phrase_and_passphrase	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_new_from_signature	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_new_random	
solana_zk_sdk::encryption::elgamal::el_gamal_secret_key_seed_from_signature	
solana_zk_sdk::encryption::elgamal::from	
solana_zk_sdk::encryption::elgamal::tests::tmp_file_path	
solana_zk_sdk::encryption::pedersen::pedersen_commitment_from_bytes	
solana_zk_sdk::encryption::pedersen::pedersen_commitment_new	
solana_zk_sdk::encryption::pedersen::pedersen_commitment_to_bytes	
solana_zk_sdk::encryption::pedersen::pedersen_opening_ct_eq	
solana_zk_sdk::encryption::pedersen::pedersen_opening_eq	
solana_zk_sdk::encryption::pedersen::pedersen_opening_from_bytes	
solana_zk_sdk::encryption::pedersen::pedersen_opening_new	
solana_zk_sdk::encryption::pedersen::pedersen_opening_new_random	
solana_zk_sdk::encryption::pedersen::pedersen_opening_to_bytes	
solana_zk_sdk::encryption::pod::auth_encryption::pod_ae_ciphertext_default	
solana_zk_sdk::encryption::pod::auth_encryption::pod_ae_ciphertext_from	

solana_zk_sdk::encryption::pod::elgamal::compressed_ristretto_from	
solana_zk_sdk::encryption::pod::elgamal::decrypt_handle_try_from	
solana_zk_sdk::encryption::pod::elgamal::el_gamal_pubkey_try_from	
solana_zk_sdk::encryption::pod::elgamal::pod_decrypt_handle_from	
solana_zk_sdk::encryption::pod::elgamal::pod_el_gamal_ciphertext_default	
solana_zk_sdk::encryption::pod::elgamal::pod_el_gamal_ciphertext_from	
solana_zk_sdk::encryption::pod::elgamal::pod_el_gamal_pubkey_from	
solana_zk_sdk::encryption::pod::grouped_elgamal::pod_grouped_el_gamal_ciphertext2_handles_default	
solana_zk_sdk::encryption::pod::grouped_elgamal::pod_grouped_el_gamal_ciphertext3_handles_default	
solana_zk_sdk::encryption::pod::pedersen::compressed_ristretto_from	
solana_zk_sdk::encryption::pod::pedersen::pedersen_commitment_try_from	
solana_zk_sdk::encryption::pod::pedersen::pod_pedersen_commitment_from	
solana_zk_sdk::pod::pod_u16_from	
solana_zk_sdk::pod::pod_u64_from	
solana_zk_sdk::pod::u16_from	
solana_zk_sdk::pod::u64_from	
solana_zk_sdk::range_proof::delta	Arithmetic overflow
solana_zk_sdk::range_proof::generators::generators_chain_default	
solana_zk_sdk::range_proof::generators::range_proof_gens_increase_capacity	Large allocation
solana_zk_sdk::range_proof::generators::range_proof_gens_new	Large allocation

<code>solana_zk_sdk::range_proof::pod::copy_range_proof_modulo_inner_product_proof</code>	Output buffer too small; slice size mismatch
<code>solana_zk_sdk::range_proof::pod::pod_range_proof_u128_try_from</code>	Slice size mismatch
<code>solana_zk_sdk::range_proof::pod::pod_range_proof_u256_try_from</code>	Slice size mismatch
<code>solana_zk_sdk::range_proof::pod::pod_range_proof_u64_try_from</code>	Slice size mismatch
<code>solana_zk_sdk::range_proof::range_proof_from_bytes</code>	
<code>solana_zk_sdk::range_proof::range_proof_to_bytes</code>	
<code>solana_zk_sdk::range_proof::util::add_vec</code>	Explicit panic
<code>solana_zk_sdk::range_proof::util::exp_iter</code>	
<code>solana_zk_sdk::range_proof::util::inner_product</code>	
<code>solana_zk_sdk::range_proof::util::poly2_eval</code>	
<code>solana_zk_sdk::range_proof::util::read32</code>	Input buffer too small
<code>solana_zk_sdk::range_proof::util::scalar_exp_size_hint</code>	
<code>solana_zk_sdk::range_proof::util::sum_of_powers</code>	
<code>solana_zk_sdk::range_proof::util::sum_of_powers_slow</code>	
<code>solana_zk_sdk::range_proof::util::vec_poly1_eval</code>	Index out of bounds (TOB-ACTBC-3)
<code>solana_zk_sdk::range_proof::util::vec_poly1_inner_product</code>	Explicit panic
<code>solana_zk_sdk::range_proof::util::vec_poly1_zero</code>	Large allocation
<code>solana_zk_sdk::sigma_proofs::pod::pod_batched_grouped_ciphertext2_handles_validity_proof_from</code>	
<code>solana_zk_sdk::sigma_proofs::pod::pod_batched_grouped_ciphertext3_handles_validity_proof_from</code>	
<code>solana_zk_sdk::sigma_proofs::pod::pod_ciphertext_ciphertext_equality_proof_from</code>	

solana_zk_sdk::sigma_proofs::pod::pod_ciphertext_commitment_equality_proof_from	
solana_zk_sdk::sigma_proofs::pod::pod_grouped_ciphertext2_handles_validity_proof_from	
solana_zk_sdk::sigma_proofs::pod::pod_grouped_ciphertext3_handles_validity_proof_from	
solana_zk_sdk::sigma_proofs::pod::pod_percentage_with_cap_proof_from	
solana_zk_sdk::sigma_proofs::pod::pod_pubkey_validity_proof_from	
solana_zk_sdk::sigma_proofs::pod::pod_zero_ciphertext_proof_from	
solana_zk_sdk::zk_elgamal_proof_program::errors::proof_verification_error_from	
solana_zk_sdk::zk_elgamal_proof_program::instruction::close_context_state	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::batched_range_proof::batched_range_proof_context_new_transcript	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::batched_range_proof::batched_range_proof_context_try_into	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::ciphertext_ciphertext_equality::ciphertext_ciphertext_equality_proof_data_new	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::ciphertext_commitment_equality::ciphertext_commitment_equality_proof_data_new	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::percentage_with_cap::percentage_with_cap_proof_data_new	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::pod::pod_proof_type_from	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::pod::proof_type_try_from	
solana_zk_sdk::zk_elgamal_proof_program::proof_data::pubkey_validity::pubkey_validity_proof_data_new	Assertion failure
solana_zk_sdk::zk_elgamal_proof_program::proof_data::zero_ciphertext::zero_ciphertext_proof_data_new	

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

© 2026 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

Trail of Bits considers this report to be business confidential information; it is licensed to Anza under the terms of the project statement of work and is intended solely for internal use by Anza. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

If published, the sole canonical source for Trail of Bits publications is the [Trail of Bits Publications page](#). Reports accessed through sources other than that page may have been modified and should not be considered authentic.

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.