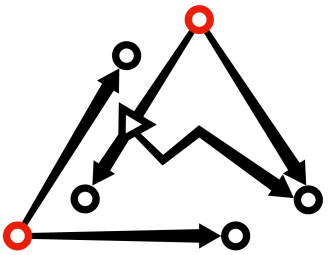

Least Authority

PRIVACY MATTERS

Token-2022 Confidential Transfer

Security Audit Report

Anza Technology

Final Audit Report: 10 November 2025

Table of Contents

Overview

Background

Solana Foundation has requested that Least Authority perform a security audit of their Confidential Transfer component of the Solana Token-2022 program. Two primary components enable confidential transfers: the ZK ElGamal Proof program and the on-chain Token-2022 program.

Project Dates

- **August 28, 2025 - September 17, 2025:** Initial Code Review *(Completed)*
- **September 19, 2025:** Delivery of Initial Audit Report *(Completed)*
- **November 10:** Verification Review *(Completed)*
- **November 10, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Confidential Transfer component of the Solana Token-2022 program followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- zk-sdk: https://github.com/solana-program/zk-elgamal-proof/tree/main/zk-sdk
- Token-2022 program: https://github.com/solana-program/token-2022/tree/main/program
- ZK ElGamal Proof Program: https://github.com/anza-xyz/agave/tree/master/programs/zk-elgamal-proof

Specifically, we examined the following Git revisions for our initial review:

- Zk-sdk: `2e45d33cf231ae5eb816b7a7a1f526d8c34c841d`
- Token-2022 program: `3986e684a115590c91cd476b4f503e6ecf4de82c`
- ZK ElGamal Proof Program: `703da254d7891aeafe085ce343b5048f80886a41`

For the verification, we examined the following Git revisions:

- Zk-sdk: `981504bb18add323e3368d35c7b0d67b1d7146a7`
- Token-2022 program: `08692efe0e84c6740780ed8b4da2bbe3efd34307`
- ZK ElGamal Proof Program: `2d407495d518293186f29408bf22783535cd14aa`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- anza-xyz-agave:
  https://github.com/LeastAuthority/anza-xyz-agave
- solana-program-token-2022:
  https://github.com/LeastAuthority/solana-program-token-2022
- solana-program-zk-elgamal-proof:
  https://github.com/LeastAuthority/solana-program-zk-elgamal-proof

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Websites:
  - https://solana.org
  - https://www.anza.xyz
- ZK ElGamal Proof Program:
  https://edge.docs.anza.xyz/runtime/zk-elgamal-proof
- Token-2022 Program:
  https://www.solana-program.com/docs/confidential-balances
- Previous audits:
  https://github.com/anza-xyz/security-audits
- Previous security advisories for the ElGamal program:
  - https://solana.com/tr/news/post-mortem-may-2-2025
  - https://solana.com/tr/news/post-mortem-june-25-2025

In addition, this audit report references the following documents:

- D. Boneh and V. Shoup, "A Graduate Course in Applied Cryptography." *toc*, 2023, [BS23].
- B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, et al., "Bulletproofs: Short Proofs for Confidential Transactions and More." *IACR Cryptology ePrint Archive*, 2017, [BBB+17]
- Y. Chen, X. Ma, C. Tang, and M. H. Au, "PGC: Decentralized Confidential Payment System with Auditability." *IACR Cryptology ePrint Archive*, 2019, [CMT+19]
- Q. Dao, J. Miller, O. Wright, and P. Grubbs, "Weak Fiat-Shamir Attacks on Modern Proof Systems." *IACR Cryptology ePrint Archive*, 2023, [DMW+23]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;

- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of the Confidential Transfer component of the Solana Token-2022 program. The project, combining the Token-2022 extensions with the `zk-elGamal/zk-sdk`, delivers privacy-preserving token operations while keeping balances and transaction validity cryptographically verifiable via homomorphic twisted ElGamal and Bulletproofs-based range proofs.

We examined the confidential transfer component of the Solana Token-2022 program and found the system to be designed with a strong emphasis on security, as demonstrated by confidential transfers, supported by high-quality cryptographic implementations, and multiple independent audits. In addition, the Anza team has provided security proofs for components of the system.

We audited the `zk-elGamal/zk-sdk` implementation against best practices. Our review included a comparison with the published specifications ([BBB+17] and [CMT+19]) and a detailed examination of the Fiat–Shamir heuristic [DMW+23]. We generally found the cryptographic implementation to be robust, largely adhering to cryptographic best practices, including zeroizing secrets. However, we identified some minor issues (Issue C and Issue D) and recommend two improvements (Suggestion 10 and Suggestion 11). We identified no deviations from the referenced protocol specifications. We also observed no deviations from strong Fiat–Shamir practice, but we recommend starting the transcript from a customizable global domain separator to mitigate cross-protocol risks (Suggestion 9). Based on these findings, we consider the `zk-sdk` a mature codebase.

We additionally reviewed the six Sigma protocol specifications covering the zero-balance proof, public-key validity proof, ciphertext-and-commitment equality proof, ciphertext-and-ciphertext equality proof, percentage proof, and ciphertext validity proof. Our review focused on the soundness proofs of these protocols, where we identified a number of typographical errors and omissions (Suggestion 3, Suggestion 4, Suggestion 5, Suggestion 6, Suggestion 7, and Suggestion 8). In all of these proofs, the description of the rewinding lemma did not match the proving technique. We recommend updating the lemma and revising all proofs accordingly (Issue B). However, we note that implementation of these changes within the proofs is straightforward and does not impact the completeness, soundness, or zero-knowledge property of the Sigma protocols as described and implemented.

System Design

The Confidential Mint and Burn extension supports private minting and burning through six core instructions. The extension maintains two encrypted states: `confidential_supply`, which is encrypted under the ElGamal public key for the supply, and `pending_burn`, which accumulates burned tokens until they are applied to the supply.

For mint operations, the zero-knowledge proofs verify three properties:

- `CiphertextCommitmentEquality` verifies that the minted amount is consistently encrypted across different public keys;
- `BatchedGroupedCiphertext3HandlesValidity` verifies that the ciphertext is correctly formed under several necessary ElGamal public keys (destination, auditor, supply); and

- `BatchedRangeProofU128` verifies that minted amounts fall within permitted ranges.

The balance calculation uses homomorphic encryption via `ciphertext_arithmetic::add_with_lo_hi()` to update the encrypted supply without decryption. Burn operations follow a similar pattern when updating account balances and pending burns.

Confidential Transfer enables private token operations through eight primary instructions. Each confidential account maintains three encrypted balance states: `available_balance` (ready for use), `pending_balance_lo` and `pending_balance_hi` (awaiting application), along with a `decryptable_available_balance` encrypted with AES for authorized decryption.

For transfer operations, the zero-knowledge proofs verify three properties:

- `CiphertextCommitmentEquality` verifies that transfer amounts are consistently encrypted across sender, receiver, and auditor keys;
- `BatchedGroupedCiphertext3HandlesValidity` verifies proper ciphertext formation across multiple public keys; and
- `BatchedRangeProofU128` verifies that amounts fall within permitted ranges and prevents negative balances or overflows.

The balance calculations use ElGamal encryption's linear homomorphism property. Similar to minting operations, `ciphertext_arithmetic::add_to()` updates pending balance components when depositing or transferring tokens. Additionally, `ciphertext_arithmetic::add_with_lo_hi()` combines these values into the total pending balance and adds them to the account's available balance.

The Confidential Transfer Fee supports private fee collection on transfers through four core instructions. The extension maintains encrypted fee states both at the mint level (aggregated fees under the withdraw authority's ElGamal key) and at the account level (per-account fee accumulation via `withheld_amount`).

Fee calculations (deduct and credit) are performed using similar additive homomorphic encryption over ciphertexts. The `CiphertextCiphertextEquality` proof verifies that `withheld_amount` (under the authority's ElGamal key) matches the amount credited to the destination account (under the recipient's public key), thereby preventing malicious fee extraction.

Dependencies

We examined the dependencies implemented in the codebase and identified several instances of vulnerable dependencies. We recommend improving dependency management (Issue E).

Code Quality

We performed a manual review of the repositories in scope and found the code to be well organized, of high quality, and closely aligned with development best practices for cryptography.

Tests

The Token-2022 program contains sufficient tests under program/tests; however, overall coverage was not measured.

Documentation and Code Comments

The project documentation provided by the Anza team clearly outlines the Token-2022 program's purpose and sufficiently describes the system's intended functionality. In particular, project documentation includes clearly written custom Sigma protocols, with explicit statements of desired security properties such as completeness, soundness, and zero-knowledge, along with the underlying assumptions. The

inclusion of proofs was especially valuable for assessing the security of the system. Overall, the codebase demonstrates a strong emphasis on maintainability, clarity, and adherence to cryptographic best practices.

Additionally, the codebase includes descriptive comments, which aid in understanding the intended behavior of the relevant components.

Scope

The scope of this review was sufficient and included all security-critical components.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Mismatch Between Actual and Expected Upper Bound for pending_balance_hi When Applying the Pending Balance	Unresolved
Issue B: Soundness Proofs Incorrect Due to Rewinding Lemma Definition	Partially Resolved
Issue C: Deriving Serialize/Deserialize/Debug for PedersenOpening Risks Leakage of Secret Openings via Unzeroized Heap/Log Copies	Resolved
Issue D: Secrets Not Zeroized in Range Proof	Resolved
Issue E : Vulnerable Dependencies	Unresolved
Suggestion 1: Improve Description of Error Messages	Resolved
Suggestion 2: Improve Code Quality	Partially Resolved
Suggestion 3: Correct Zero-Balance Proof	Resolved
Suggestion 4: Correct Public Key Validity Proof	Resolved
Suggestion 5: Correct Ciphertext-Ciphertext Equality Proof	Partially Resolved
Suggestion 6: Correct Ciphertext-Commitment Equality Proof	Partially Resolved
Suggestion 7: Correct Ciphertext Validity Proof	Resolved
Suggestion 8: Correct Percentage Proof	Partially Resolved
Suggestion 9: Customize Domain Separators	Unresolved
Suggestion 10: Implement a Public Key Consistency Check in ElGamalKeypair::try_from	Resolved
Suggestion 11: Expand Security-Critical Comment on Extra Hashing	Resolved

Rationale	

Issue A: Mismatch Between Actual and Expected Upper Bound for pending_balance_hi When Applying the Pending Balance

Location

`program/src/extension/confidential_transfer/account_info.rs#L111`

Synopsis

The Token-2022 Confidential Transfer design permits `pending_balance_hi` to accumulate up to 48 bits of value across several incoming transfers. The program, however, decrypts `pending_balance_hi` as a 32-bit integer when constructing an `ApplyPendingBalance` instruction. Once the accumulated high part exceeds 32 bits, `decrypt_u32` fails, causing `ApplyPendingBalance` to abort with `AccountDecryption` for an otherwise valid account state.

Impact

Medium.

Affected accounts might be unable to apply their pending balance and, as a result, update their available balance (that is, their spendable balance) unless an alternative is provided. Since the `ApplyPendingBalance` instruction is the standard way to update the available balance, this can lead to a denial of service in certain scenarios.

Feasibility

Medium.

Since any third party is allowed to transfer tokens to the target account, and the default configuration allows up to 2^{16} transfers before applying and resetting the pending balance, a repeated number of high-value incoming transfers can result in a `pending_balance_hi` value greater than the value of $2^{32}-1$ supported by `decrypt_u32`.

Severity

Medium.

Preconditions

The target account must have the Confidential Transfer extension enabled and receive enough incoming transfers until the accumulated high component of the pending balance overflows 32 bits.

Technical Details

The `ApplyPendingBalanceAccountInfo` data type stores the information necessary to create an `ApplyPendingBalance` instruction. This includes the `pending_balance_lo`, `pending_balance_hi`, and `decryptable_available_balance` values. To update the available balance, the two pending balance components are combined and added to the current available balance. Each pending balance component is decrypted using the `decrypt_u32` function, which fails if the plaintext exceeds 32 bits. The specification permits `pending_balance_hi` to accumulate values of up to 48 bits, so in certain scenarios, the `decrypt_u32` function may fail for valid states and return None instead of the expected plaintext value.

Mitigation

We suggest the following measures to mitigate the issue:

- Configure a lower `maximum_pending_balance_credit_counter` value to limit the number of consecutive incoming transfers before updating the available balance.
- Apply the pending balance more frequently to prevent `pending_balance_hi` from storing values that can trigger the error.

Remediation

We recommend the following remediation strategies:

- Implement a decrypt function that supports up to 48-bit plaintext values and use it to decrypt `pending_balance_hi`.
- Alternatively, if decryption efficiency or performance is the main concern, limit `pending_balance_hi` to 32 bits and update the documentation accordingly.
- Add invariant checks for the result of the decryption operations, both for `pending_balance_lo` ($\leq 2^{16}-1$) and `pending_balance_hi` ($\leq 2^{48}-1$).

Status

The Anza team determined that this issue does not represent a practically feasible vulnerability, although, in theory, the pending balance can overflow and there is no protocol-level mechanism to prevent it.

The standard procedure for decrypting the pending balance involves retrieving all incoming transfer transactions associated with an account, decrypting the ciphertexts corresponding to these transactions, and summing the resulting values to compute the pending balance. The encrypted values in these ciphertexts are limited to 16 and 32 bits, and within this range, decryption (discrete log) is fast.

For typical use cases, several practical optimizations are implemented. A mint can be configured with a cap on the maximum number of credits an account may receive before an `ApplyPendingBalance` instruction must be invoked on the account. This value is typically set to `2^16`. Consequently, `pending_balance_lo` (which can encrypt up to `2^16`) is capped at `2^32` and the `pending_balance_hi` (which can encrypt up to `2^32`) is capped at `2^48`. Under these conditions, the `pending_balance_lo` will always be decryptable. While the `pending_balance_hi` may exceed `2^32`, based on our measurements, computing a discrete log of approximately `2^40` requires only a few seconds. It is also unlikely that an average user would transfer amounts of this magnitude frequently enough to cause the balance to exceed `2^40`, as doing so would require receiving `2^32` credits for `2^8` times without invoking `ApplyPendingBalance`.

Verification

Unresolved.

Issue B: Soundness Proofs Incorrect Due to Rewinding Lemma Definition

Location

`runtime/zk-docs/zero_proof.pdf`

`runtime/zk-docs/percentage_with_cap.pdf`

`runtime/zk-docs/ciphertext_validity.pdf`

`runtime/zk-docs/pubkey_proof.pdf`

runtime/zk-docs/ciphertext_ciphertext_equality.pdf

runtime/zk-docs/ciphertext_commitment_equality.pdf

Synopsis

Lemma 1.2, the “Rewinding Lemma,” as used in the proofs, is stated with mutual independence of X, Y, Y', Z, and Z'. In the protocol, however, Z and Z' (the responses) are functions of (X, Y) and (X, Y'), and therefore are not independent of the challenges.

Impact

Low. This cannot be exploited by an adversary since the theorems still hold true.

Feasibility

Low.

Severity

Low.

Technical Details

In a Sigma protocol, the response Z is a function of the first message and the challenge, for example, $Z=g(X, Y)$. It is not independent of Y. Lemma 1.2, as written, requires Z, Z' to be mutually independent of X, Y, Y', which does not hold in the protocol and makes the lemma inapplicable to the interactive setting. The correct form used in rewinding analyses quantifies a predicate over (X, Y) (or treats Z as a deterministic function of (X, Y)), not over independent Z and Z'.

Remediation

We recommend correcting Lemma 1.2 and replacing the statement with a version appropriate for public-coin Sigma protocols (for example, the version in Boneh's book, p. 758, Lemma 19.2: “Rewinding Lemma,” [BS23]).

Status

The Anza team has partially resolved the issue, with a typographical error remaining in the probability equation across all relevant documents.

Verification

Partially Resolved.

Issue C: Deriving Serialize/Deserialize/Debug for PedersenOpening Risks Leakage of Secret Openings via Unzeroized Heap/Log Copies

Location

zk-sdk/src/encryption/elgamal.rs

Synopsis

`PedersenOpening` represents the blinding factor `r` used in Pedersen commitments and in the twisted ElGamal construction. The type currently derives `Serialize`, `Deserialize`, and `Debug`. While the struct itself is annotated `#[zeroize(drop)]`, serialization and debugging create unprotected heap or string copies (for example, `Vec<u8>`, `String`, JSON) that Zeroize will not clear, and these copies can

persist in process memory, logs, crash dumps, or files. This increases the risk that `r` is recovered postmortem or via telemetry or logging.

Impact

High.

Leakage of the opening `r` undermines the hiding of Pedersen commitments and twisted ElGamal ciphertexts. With `r` and a public commitment `C`, an adversary can compute $v \cdot G$ and then recover the clear value for u32-bounded amounts using the built-in discrete log decoder. This breaks confidentiality of encrypted or committed amounts for common ranges.

Feasibility

Low.

Exploitation requires access to memory or artifacts where serialized or debugged values are written (for example, logs, telemetry, crash dumps, or temporary files) or execution of code paths that serialize openings (as demonstrated in tests).

Severity

Low.

Preconditions

For this issue to occur, the code paths in the SDK consumer or in tests and tools must serialize or deserialize `PedersenOpening` (for example, JSON or bincode), or print or format it via `Debug`, or otherwise copy it into unprotected buffers. In addition, an attacker or post-incident analyst must have access to the process memory, logs, crash or core dumps, swap or page files, or persisted artifacts.

Remediation

We recommend removing the automatic derives `Serialize`, `Deserialize`, and `Debug` from `PedersenOpening` and instead, providing explicit conversion methods such as `to_bytes()` and `from_bytes()`.

Status

The Anza team has resolved the issue as recommended.

Verification

Resolved.

Issue D: Secrets Not Zeroized in Range Proof

Location

`src/range_proof/mod.rs#L168`

Synopsis

Private scalars, such as blinding factors used during Bulletproofs range-proof construction, are not zeroized after use, leaving sensitive material resident in process memory until reclaimed by the allocator.

Impact

Low.

Leakage of ephemeral witnesses or randomness can aid post-compromise analysis but does not, by itself, enable proof forgery or immediate value recovery.

Feasibility

Low.

An adversary with host or process introspection (for example, core dumps, swap, crash reporting, or memory forensics) can retrieve remnants from the heap or stack.

Severity

Low.

Preconditions

Attackers must be able to read process memory or artifacts (core dumps, swap, crash logs) from a system running range-proof generation with this library.

Technical Details

The range proof module inherits behavior from dalek-bulletproofs, which does not zeroize private variables. As a result, temporary scalars and witness material are dropped without explicit clearing. This differs from other components such as ElGamal ciphertext and Sigma proofs, which were written to zeroize for safety.

Remediation

We recommend the following steps to remediate this issue:

- Wrap secret scalars and points in `zeroize::Zeroizing`; and
- Enable zeroization features in dependencies where available.

Status

The Anza team has resolved the issue as recommended.

Verification

Resolved.

Issue E: Vulnerable Dependencies

Synopsis

Analyzing the project's dependencies with `cargo audit` reveals four vulnerable crates:

- `curve25519-dalek v3.2.0`: timing variability as described in this security advisory.
- `ed25519-dalek v1.0.1`: double public key signing function oracle attack as described in this security advisory.
- `idna v0.1.5`: improper parsing of Punycode labels as described in this security advisory.
- `tracing-subscriber v0.3.19`: log injection with user-controlled ANSI escape sequences as described in this security advisory.

Impact

Consult the listed advisories on a case-by-case basis.

Feasibility

Consult the listed advisories on a case-by-case basis.

Severity

Consult the listed advisories on a case-by-case basis.

Technical Details

The Token-2022 program includes dependencies with known security issues that have since been resolved in updated versions of these dependencies.

Remediation

We recommend updating these crates and following a process that emphasizes secure crate usage to avoid introducing vulnerabilities into the Token-2022 program and to mitigate supply-chain attacks. This process includes:

- Manually reviewing and assessing currently used crates;
- Upgrading crates with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained crates with secure and battle-tested alternatives, if possible;
- Pinning crates to specific versions, including pinning build-level crates in the `Cargo.toml` file to a specific version;
- Only upgrading crates upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including Automated Dependency auditing reports in the project's CI/CD workflow.

Status

At the time of the verification, the issue had not been resolved.

Verification

Unresolved.

Suggestions

Suggestion 1: Improve Description of Error Messages

Location

`program/src/extension/confidential_mint_burn/account_info.rs`

`program/src/extension/confidential_transfer_fee/account_info.rs`

`program/src/extension/confidential_transfer_fee/processor.rs`

`program/src/extension/confidential_transfer/account_info.rs`

`program/src/extension/confidential_transfer/processor.rs`

Synopsis

Some of the error messages displayed when an intended operation fails do not reflect the corresponding error with full clarity. This can obscure the actual reason for the failure.

Mitigation

We recommend improving the error messages to make them more descriptive and contextual. Below, we provide a non-exhaustive list of suggested improvements:

- Use `CiphertextDecryption` (or similar) instead of `MalformedCiphertext` here, here, and here.
- Use `CiphertextConversion` (or similar) instead of `MalformedCiphertext` here, here, here, here, here, here, here, here, here, and here.
- Replace the incorrect message here with `CiphertextConversion`.
- Use `TokenError::CiphertextArithmeticFailed` instead of `ProgramError::InvalidInstructionData` here, here, here, and here.
- Use `TokenError::ConfidentialTransferBalanceMismatch` instead of `ProgramError::InvalidInstructionData` here.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 2: Improve Code Quality

Location

- Code comments mismatch: `program/src/extension/confidential_transfer/account_info.rs#L191-L192`
- Redundant logic: `program/src/extension/confidential_transfer/account_info.rs#L131-L135`
- Variable naming inconsistency: `program/src/extension/confidential_transfer/verify_proof.rs#L48-L49`
- Performance inefficiency: `program/src/extension/confidential_transfer_fee/processor.rs#L224-L255`

Synopsis

During our extensive review of the codebase, our team identified practices that impact its quality, readability, and maintainability. To illustrate, the following is a non-exhaustive list of examples:

- Correct the comment to specify the appropriate instruction type when creating a `Withdraw` instruction.
- Remove the implementation that calls to `get_pending_balance` to reduce code footprint and prevent inconsistencies if the implementation changes.
- Update the return variable naming in `verify_withdraw_proof` to accurately represent the proof type.
- Perform the validity checks before computing `aggregate_withheld_amount` to improve performance and avoid unnecessary computation.

Mitigation

We recommend addressing the items listed above to improve overall code quality, and using them as a baseline for identifying and remediating similar issues across the codebase.

Status

The Anza team has partially implemented the mitigation.

Verification

Partially Resolved.

Suggestion 3: Correct Zero-Balance Proof

Location

`runtime/zk-docs/zero_proof.pdf`

Synopsis

Several errors and omissions were identified in the soundness proof of the zero-balance proof:

- Page 6, "Witness validity": The final equality sign should be replaced with a multiplication symbol in "*This means that* $(z-z') \cdot P=(c-c') \cdot H$ *[…].*"
- Page 5, Section 4.2, "Description of extractor": The acceptance checks for transcripts with `c` and `z` are missing from the description of extractor E. The extractor should explicitly verify the two equations before computing `s`; otherwise, it may return an invalid witness.
- Page 5, 4.2, "Description of extractor": The extractor E is described as doing exactly two runs. For witness-extended emulation (Def. 2.3), the emulator must return a valid witness for (almost) every accepting transcript. That is, the probability that `tr` is an accepting transcript but E fails to extract must be negligible. To achieve this, E should *repeat* the "rewind with a fresh challenge" step a polynomially bounded number of times until it obtains two accepting transcripts with distinct challenges. With repetition, the failure probability becomes negligible. The current proof omits this step.
- Page 5f, "Abort probability": The probability statement for `abort` is inverted and requires the insertion of an additional "not." The rewinding bound (even when corrected according to Issue B) shows that the probability of obtaining two accepting transcripts with distinct challenges (that is, the extractor E succeeding) is at least $\varepsilon^2 - \varepsilon/p$. Then, when ε is non-negligible, the probability of E succeeding is also non-negligible. The non-negligible probability applies to E's success, not to its abort. The abort probability is at most $1 - (\varepsilon^2 - \varepsilon/p)$.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 4: Correct Public Key Validity Proof

Location

`runtime/zk-docs/pubkey_proof.pdf`

Synopsis

Several errors and omissions were identified in the public key validity proof:

- Page 4, Section 4.1: The last line should be "`z*H = [...] = c*P + Y`" and not "`z*H = c*P + y*Y`."
- Page 5, Section 4.2: The last bullet point "*return s as the witness*" should be "s^{-1}." To match the relation R, E should return $w=s=(s^{-1})^{-1}$."
- Page 5, Section 4.2, "Description of extractor": According to Definition 2.3 (witness-extended emulation (WEE)), the extractor E should return a transcript and a witness. The text has the emulator produce two runs and "*return s as the witness*," but it never explicitly specifies which

transcript it outputs to the environment in the WEE game (Definition 2.3). The simplest approach is to output the first transcript produced in the emulation and perform any rewinding or extraction after fixing that transcript.

- Page 5, 4.2, "Description of extractor": The acceptance checks for transcripts with c and z are not included in the description of extractor E. The extractor should explicitly verify those transcripts and repeat the rewind if they fail; otherwise, it may return an invalid witness.
- Page 5, 4.2, "Description of extractor": The extractor E is described as doing exactly two runs. For witness-extended emulation (Def. 2.3), the emulator must return a valid witness for (almost) every accepting transcript. That is, the probability that `tr` is an accepting transcript but E fails to extract must be negligible. To achieve this, E should *repeat* the "rewind with a fresh challenge" step a polynomially bounded number of times until it obtains two accepting transcripts with distinct challenges. With repetition, the failure probability becomes negligible. The current proof omits this step.
- Page 5f, "Abort probability": The probability statement for abort is inverted and requires the insertion of an additional "not." The rewinding bound (even when corrected according to Issue B) shows that the probability of obtaining two accepting transcripts with distinct challenges (that is, the extractor E succeeding) is at least $\varepsilon^2 - \varepsilon/p$. When ε is non-negligible, the probability of E succeeding is also non-negligible. The non-negligible probability applies to E's success, not to its abort. The abort probability is at most $1 - (\varepsilon^2 - \varepsilon/p)$.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 5: Correct Ciphertext-Ciphertext Equality Proof

Location

`runtime/zk-docs/ciphertext_ciphertext_equality.pdf`

Synopsis

Several errors and omissions were identified in the soundness proof of the ciphertext-ciphertext equality proof:

- Page 7, "Witness validity": "`z_s`" should be used instead of "`z_x`" in "`z_x * P_0 = c * H + Y_0`" and "`z'_x * P_0 = c' * H + Y_0`", as the verifier's first check is for "`z_s`."
- Page 5, 4.1 Proof of Theorem 3.1: "`D_1`" should be "`H`" in the third verifier equation in the completeness proof.
- Page 7, "Witness validity": "`Y_4`" should be "`Y_3`" since the protocol only defines "`Y_0 … Y_3.`"
- Page 6, 4.1 Proof of Theorem 3.2: "*two executions of the ciphertext equality protocol*" should replace "*two executions of the zero-balance protocol.*"
- Page 6, third bullet point: In the transcript (`Y_0, … Y_3, c, z'_s, z'_x, z'_r`) of the second execution after rewinding, "`c'`" should be used instead of "`c.`"
- Page 5f, "Abort probability": The probability statement for abort is inverted and requires the insertion of an additional "not." The rewinding bound (even when corrected according to Issue B) shows that the probability of obtaining two accepting transcripts with distinct challenges (that is, the extractor E succeeding) is at least $\varepsilon^2 - \varepsilon/p$. When ε is non-negligible, the probability of E

succeeding is also non-negligible. The non-negligible probability applies to E's success, not to its abort. The abort probability is at most $1 - (\varepsilon^2 - \varepsilon/p)$.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has partially implemented the recommended mitigation, with one outstanding item remaining. Specifically, on page 6, the third bullet point, in (`Y_0, … Y_3, c, z'_s, z'_x, z'_r`) of the second execution after rewinding, "`c'`" should be used instead of "`c.`"

Verification

Partially Resolved.

Suggestion 6: Correct Ciphertext-Commitment Equality Proof

Location

`runtime/zk-docs/ciphertext_commitment_equality.pdf`

Synopsis

Several errors and omissions were identified in the soundness proof of the ciphertext-commitment equality proof:

- Page 7, "Witness validity": "`z_s`" should be used instead of "`z_x`" in "`z_x * P_EG = c * H + Y_0`" and "`z'_x * P_EG = c' * H + Y_0`", as the verifier's first check is for "`z_s.`"
- Page 6, third bullet point: In the transcript (`Y_0, Y_1, Y_2, c, z'_s, z'_x, z'_r`) of the second execution after rewinding, "`c'`" should be used instead of "`c.`"
- Page 5, 4.1 Proof of Theorem 3.1: "`D_EG`" should be "`H`" in the third verifier equation in the completeness proof. Additionally, "`Y_0 = y_s * P_EG`" should be used instead of "`Y_0 = y_s * P`" in the definition of `Y_0`.
- Page 5, 4.1 Proof of Theorem 3.2: "*two executions of the ciphertext commitment equality protocol*" should replace "*two execution of the zero-balance protocol.*"
- Page 6, Section 4.2, "Description of extractor": According to Definition 2.3 (witness-extended emulation (WEE)), the extractor E should return a transcript and a witness. The text has the emulator produce two runs and return a witness "`(s,x,r),`" but it never explicitly specifies which transcript it outputs to the environment in the WEE game (Definition 2.3). The simplest approach is to output the first transcript produced in the emulation and perform any rewinding or extraction after fixing that transcript.
- Page 6, "Abort probability": The probability statement for abort is inverted and requires the insertion of an additional "not." The rewinding bound (even when corrected according to Issue B) shows that the probability of obtaining two accepting transcripts with distinct challenges (that is, the extractor E succeeding) is at least $\varepsilon^2 - \varepsilon/p$. When ε is non-negligible, the probability of E succeeding is also non-negligible. The non-negligible probability applies to E's success, not to its abort. The abort probability is at most $1 - (\varepsilon^2 - \varepsilon/p)$.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has partially implemented the recommended mitigation, with one outstanding item remaining. Specifically, on page 6, the third bullet point, in (`Y_0, … Y_3, c, z'_s, z'_x, z'_r`) of the second execution after rewinding, "`c'`" should be used instead of "`c`."

Verification

Partially Resolved.

Suggestion 7: Correct Ciphertext Validity Proof

Location

`runtime/zk-docs/ciphertext_validity.pdf`

Synopsis

Several errors and omissions were identified in the soundness proof of the ciphertext validity proof:

- Page 5f, Description of the emulator: According to Definition 2.3 (witness-extended emulation (WEE)), the extractor E should return a transcript and a witness. The text has the emulator produce two runs and return a witness "`(r, x)`," but it never explicitly specifies which transcript it outputs to `A_2` in the WEE game (Definition 2.3). The simplest approach is to output the first transcript produced in the emulation and perform any rewinding or extraction after fixing that transcript.
- Page 5f, Description of the emulator: The emulator E is described as doing exactly two runs and aborting if `c=c'` or the second run is not an accepting transcript. For witness-extended emulation (Def. 2.3), the emulator must return a valid witness for (almost) every accepting transcript. That is, the probability that `tr` is an accepting transcript but E fails to extract must be negligible. To achieve this, E should *repeat* the "rewind with a fresh challenge" step a polynomially bounded number of times until it obtains two accepting transcripts with distinct challenges. Repetition reduces the failure probability to negligible. The current proof omits this step.
- Page 5, 4.1 Proof of Theorem 3.2: "*two executions of the ciphertext validity protocol*" should replace "*four executions of the zero-balance protocol.*"
- Page 5f, "Abort probability": The probability statement for abort is inverted and requires the insertion of an additional "not." The rewinding bound (even when corrected according to Issue B) shows that the probability of obtaining two accepting transcripts with distinct challenges (that is, the extractor E succeeding) is at least $\varepsilon^2 - \varepsilon/p$. When ε is non-negligible, the probability of E succeeding is also non-negligible. The non-negligible probability applies to E's success, not to its abort. The abort probability is at most $1 - (\varepsilon^2 - \varepsilon/p)$.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 8: Correct Percentage Proof

Location

`runtime/zk-docs/percentage_with_cap.pdf`

Synopsis

Several errors and omissions were identified in the soundness proof of the percentage proof:

- Page 6, Protocol table and Page 7, section "Fee equal to max fee": "`y_max`" should be used instead of "`r_max`." This matches the prose underneath and is consistent with committing to its value. The prose underneath also needs refinement, since, for example, "`z_r`" is not defined.
- Page 5, Protocol table: "`y_x`" should be used instead of "`y_s`" in "`z_x \arrow c_equality * x + y_s`."
- Page 7, "Protocol": "we denote `D \in G` to denote the commitment `C_fee * G`" should be replaced with "`C_fee - max_fee * G`" instead.
- Page 7, "Fee equal to max fee", Step 1: "sample random challenge `c_max`" should be "sample random challenge `c_equality`" (since it is for the equality proof)
- Page 6, "Specification": The language description of `L^{fee}_{G, H, bp, maxfee}` could be rewritten for clarity, since this language should capture the existential statements that each Sigma protocol proves. By defining "`C_delta := C_fee * 10000 - bp * C_amt`", the witness statements can be expressed as: "There exist `x, r_delta, r_claimed` such that `C_delta = x * G + r_delta * H` and such that `C_claimed = x * G + r_claimed * H`" for the percentage part. (The original description is the group equality, which also fixes the random values). In addition, the statement "There exists a `r_max` such that `D = r_max * H`" should be included for the cap part of the proof.
- Page 8, Theorem 3.2: The description of the extractor is missing and should be added.

Mitigation

We recommend addressing and correcting the points described above.

Status

The Anza team has partially implemented the recommended mitigation, with one outstanding item remaining. Specifically, on page 7, under "`Fee equal to max fee`," the computation "`Y_max ← r_max*H`" is incorrect and should be "`Y_max ← y_max*H`."

Verification

Partially Resolved.

Suggestion 9: Customize Domain Separators

Location

`zk-elgamal-proof/main/zk-sdk`

Synopsis

Generator H is fixed across protocol instantiations by hashing only the basepoint, which creates cross-protocol context confusion when different systems reuse the same curve and parameters. Parameterizing H with a protocol-scoped domain separator would bind commitments to their intended context.

Mitigation

We suggest the following measures:

- Derive H via `hash_to_group(DOMAIN // basepoint_bytes)` with a configurable global `DOMAIN` per protocol instance.
- Keep the empty string as the default for backward compatibility, for example: `DOMAIN = "PROTOCOL_XYZ | curve25519 | solana zk-elgamal | v1 | G"`
- Require that operators treat and verify commitments as protocol-scoped and refrain from cross-domain artifact mixing until an update is released.

Status

The Anza team acknowledged the value of the suggestion but decided against implementing it, noting that doing so would break compatibility with previous versions.

Verification

Unresolved.

Suggestion 10: Implement a Public Key Consistency Check in ElGamalKeypair::try_from

Location

zk-sdk/src/encryption/elgamal.rs#L270

Synopsis

`ElGamalKeypair::try_from` accepts a keypair without verifying that the provided public key matches the public key derived from the secret key, which enables inconsistent key pairs.

Mitigation

We recommend adding a check in `try_from` that derives the public key from the secret key, compares it to the provided value, and returns an error on mismatch. We further recommend requiring callers to recompute and verify before use until an update is released.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 11: Expand Security-Critical Comment on Extra Hashing Rationale

Location

zk-sdk/src/range_proof/mod.rs#L277

Synopsis

The inline comment fails to explain the security rationale for the "extra hashing" introduced after a prior bug in which some scalar proof components were not hashed. This omission renders verification assumptions opaque and reduces auditability.

Mitigation

We recommend expanding security-critical comments to explain the rationale for additional hashing.

Status

The Anza team has implemented the mitigation as recommended.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.