ZKSECURITY

Solana Confidential Transfers

Date: September 16th, 2025

Introduction

On September 15th, Anza engaged zkSecurity to review its implementation of confidential transfers, as part of the Token-2022 standard. Two consultants spent 4 weeks to review the full stack, from the extension logic added to the Token-2022 program, down to the zero-knowledge proofs implementations.

The code was found to be extremely well-documented, thoroughly specified, as well as written in a heavily defensive style. The audit did not find any serious vulnerabilities. The Anza team was also exceptionally collaborative and responsive during the audit.

Scope

The scope focused on parts of the following three repositories:

zk-sdk and ZK ElGamal Proof Program. The solana-program/zk-elgamal-proof repository contains implementation for all proof systems (an inner-product argument (IPA) implementation using bulletproofs, a range proof implementation from the bulletproofs paper, and a number of sigma proofs) as well as the zk elgamal proof Solana program. We audited the codebase at commit `703da254d7891aeafe085ce343b5048f80886a41`.

ZK ElGamal Proof Built-In Program. We focused on the the ZK ElGamal Proof Program contained in the agave node codebase, which wraps and deploys the zk-elgamal-proof program implemented in the previous `zk-elgamal-proof` repository as a built-in program on the Solana blockchain. The audit was focused on commit `125487843ea46f3543683ac9a92a08cf5a7ce16c`.

Confidential Transfers as an Extension to the Token-2022 Solana Program. The logic for confidential transfers in Solana was implemented as part of the solana-program/token-2022 repository, where low-level helpers, client-side logic, as well as a helper Solana program "elgamal registry" were segregated in a `confidential-transfer/` folder, and the main confidential transfer logic was implemented as three main extensions in the `program/src/extension/confidential_*/` folders. The audit used commit `20a9e87885dec4935278fe6dfe4bc4d435a09f6b`.

Strategic Recommendations

We recommend addressing the following high-level issues:

Proofs APIs Are Insecure As Standalone Functions. All the proof APIs, including the bulletproofs, range proof, and sigma proofs APIs are insecure if used as standalone primitives. While this is not an issue in the current way they are used in the overall Token-2022 confidential transfer protocol, this could become an issue if used in other contexts. The insecurity of these APIs mostly stems from the fact that the verification logic is split across the SDK, the ZK ElGamal program and the confidential extensions to the Token-2022 program. Looking at the SDK as a standalone crate, the two main consequences of having split verification logic are that the public inputs are not properly constrained or computed (see Verifier Should Compute Expected Instance of the Fee Proof By Themselves and Unconstrained Fee in Standalone percentage_with_cap Proof) and the Fiat-Shamir implementations do not properly absorb instance parameters (see No Proofs Handle Fiat-Shamir Correctly In Standalone Versions, Fiat-Shamir Issue Leads To Colliding Transcripts In Range Proof and Fiat-Shamir of Public Parameters).

Unnecessary Determinism in Batch Verification. As pointed out in Unnecessary Deterministic Randomness in Batch Verification, the use of deterministic randomness (Fiat-Shamir style) has already been the source of two bugs, in exchange for debatable speed gains. Considering this, we recommend reconsidering whether batching verifier checks is necessary at all.

Code Duplication Is Error Prone. A lot of the Token-2022 confidential extensions logic involves code duplication across many (sub)instructions, which makes the code difficult to audit with respect to its global (implied) state machine. In Typed Abstraction Could Go a Long Way we ponder on the usefulness of typed abstractions, and on the trade-offs of refactoring at this point in time.

Solana Confidential Transfer Overview

The zk-sdk is the foundation of the confidential transfer protocol, as it provides the low-level zero-knowledge proof primitives. Previously part of the agave repository, it now exists as a standalone repository. The repository implements Authenticated Encryption (via AES-GCM-SIV), twisted ElGamal encryption, a number of zero-knowledge proof systems, as well as the building blocks for exposing all of these in a Solana program.

The implementation uses the **Ristretto group** (and the official dalek implementation), built on top of Curve25519. The abstraction provides a prime-order group while the implementation provides a safe-to-use API.

To implement Fiat-Shamir in the different proof systems, the codebase relies on the **Merlin** transcript framework, which builds on top of the Strobe protocol in order to safely absorb prover messages and the shape of the transcript in a duplex construction.

The Twisted ElGamal Encryption Scheme

Balances are encrypted using the twisted ElGamal encryption scheme. Since the scheme is additively homomorphic, balances and transfer amounts can be encrypted, and added/subtracted by the on-chain program.

To **generate a keypair**, a peer produces a random secret key s and a public key $P = s^{-1}H$ where H is a known generator.

To **asymmetrically encrypt** a message x to a peer's public key P, the sender computes a ciphertext as the pair (where G is a known generator as well):

$$C = xG + rH, \quad D = rP$$

Finally, to **decrypt**, the recipient can compute and subtract rH from C (without learning r) by computing

$$sD = srP = srs^{-1}H = rH$$

From there on, the recipient still has to figure out the discrete logarithm of xG, which they can do with the usual discrete logarithm algorithm if x is small.

In Solana's confidential transfer scheme, a few things are done to make this design realistic:

- a token's total supply is capped to a 64-bit value, and amounts (as well as some balances) are split between the lower 16-bit and the higher 32-bit (capped at 48 bits), allowing for easier solving of the discrete logarithm.
- a pending balance is used to accumulate transfers to an account, allowing the account to produce transfer proofs without having their proof being invalidated (as someone being able to modify their actual balance would invalidate the proof), and separating the management of the main balance from the decryption of transfers received.
- zero-knowledge proofs, including sigma protocols and range proofs, are used in a number of places to ensure that amounts are correct, or that balances are correctly added or subtracted to.

Furthermore, because a recipient never learns the random value r, many of the sigma proofs are used to prove correct re-encryption of encrypted values, where the new r' value used in the re-encryption is known, allowing the range proofs to be created.

Sigma Protocols Overview

Sigma protocols (sometimes "Σ-protocols") are a class of interactive zero-knowledge proofs. They are characterized by having 1 round (3 moves): the prover first sends a *commitment*, the verifier sends a *challenge*, and finally the prover responds with an *opening*. In comparison to zkSNARKs, Sigma protocols are not succinct; however, they are rarely applied to large problems and can be more performant than zkSNARKs for small, specialized tasks. As we will see below, Sigma protocols are particularly well-suited for proving statements about Pedersen commitments (and therefore Twisted ElGamal ciphertexts).

FS transform. These protocols can be made into non-interactive protocols using the Fiat-Shamir (FS) transform. In this section we give a reminder of the necessary security properties and outline how all the protocols used meet these.

Security properties

Necessary properties. For the overall system to be secure and confidential, it is necessary that the non-interactive proofs (resulting from the FS transform) have *knowledge soundness* and *zero-knowledge*. To guarantee these properties, we must analyze the security properties of the interactive protocols and ensure that the FS transform and non-interactive threat model do not incur a large security degradation.

Proving method. The standard way to show that the non-interactive proof is knowledge sound is to show that the interactive protocol has *special soundness*. To show that the non-interactive proof is zero-knowledge, it is enough to show that the interactive proof is *honest-verifier zero-knowledge* (HVZK). In some cases (*e.g.*, the OR composition), it will be helpful to show a slight variant known as *special HVZK*.

Sigma protocol for pre-image of a homomorphism

All the Sigma protocols in `zk-sdk` are instantiations of a more generic Sigma protocol to prove knowledge of the pre-image of a homomorphism. We refer readers to A Graduate Course in Applied Cryptography (Boneh and Shoup), chapter 19.5.4 for further details on this protocol. For convenience and as a means to introduce notation, we include the relevant excerpt below.

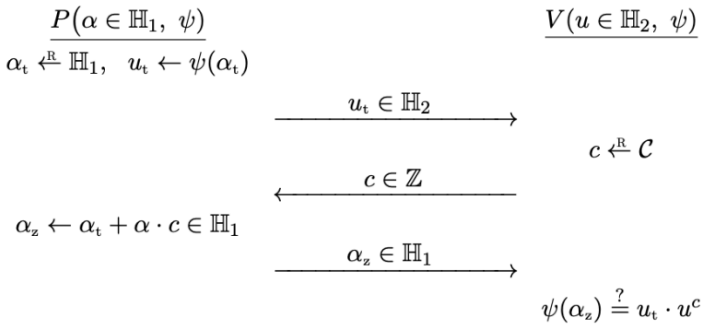

Figure 19.9: A Sigma protocol for the preimage of a homomorphism

19.5.4 A Sigma protocol for the pre-image of a homomorphism

All the Sigma protocols presented so far, including the general linear protocol, can be described more clearly and succinctly using the language of group homomorphisms. Let $\mathbb{H}_1$ and $\mathbb{H}_2$ be two finite abelian groups of known order and let $\psi : \mathbb{H}_1 \to \mathbb{H}_2$ be a group homomorphism. We will write the group operation in $\mathbb{H}_1$ additively and the group operation in $\mathbb{H}_2$ multiplicatively.

Let $u \in \mathbb{H}_2$. Fig. 19.9 gives a Sigma protocol that allows a prover to convince a verifier that it "knows" a preimage of u under ψ. Specifically, the protocol is a Sigma protocol for the relation

$$\mathcal{R} := \big\{(\alpha, (u, \psi)) \in \mathbb{H}_1 \times (\mathbb{H}_2 \times \mathcal{F}) :\ \psi(\alpha) = u\big\}. \tag{19.15}$$

Here $\alpha \in \mathbb{H}_1$ is the preimage under ψ for $u \in \mathbb{H}_2$. The prover in Fig. 19.9 has the witness $\alpha \in \mathbb{H}_1$, the verifier has the image $u \in \mathbb{H}_2$, and both parties have $(\mathbb{H}_1, \mathbb{H}_2, \psi)$. The challenge space $\mathcal{C}$ is $\{0, 1, \ldots, N-1\} \subseteq \mathbb{Z}$ for some integer N.

Theorem 19.12. *The protocol in Fig. 19.9 is a Sigma protocol for the relation $\mathcal{R}$ defined in (19.15). Moreover, it is special HVZK, and provides special soundness whenever the smallest prime factor of $|\mathbb{H}_1| \times |\mathbb{H}_2|$ is at least $|\mathcal{C}|$.*

Security of `zk-sdk` Sigma protocols

To prove security of the Sigma protocols in `zk-sdk` it suffices to show how these protocols map to the generic pre-image protocol above. In all cases it is easy to check that ψ is a group homomorphism and that the conditions are met to apply Theorem 19.12 of Boneh and Shoup.

Public key validity

Given a public key P, the public key validity protocol proves knowledge of a secret key s such that $\psi(s^{-1}) = P$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism ψ: $\mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := \mathbb{Z}_p, \quad \mathbb{H}_2 := \mathbb{G}, \quad \psi(s^{-1}) := s^{-1} \cdot H, \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameter H.

Zero check

Given a system parameter H and a Pedersen commitment C, the zero check protocol proves knowledge of a secret key s such that $\psi(s) = (H, C)$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism ψ: $\mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := \mathbb{Z}_p, \quad \mathbb{H}_2 := \mathbb{G}^2, \quad \psi(s) := (s \cdot P, s \cdot D), \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the public key P and decryption handle D.

Grouped ciphertext validity

Given a Pedersen commitment C and decryption handles $D_1, \ldots, D_\ell$, the grouped ciphertext validity protocol proves knowledge of a message x and opening r such that $\psi(r, x) = (C, [D]_{i=1}^{\ell})$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism ψ: $\mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$

are defined as:

$$\mathbb{H}_1 := (\mathbb{Z}_p)^2, \quad \mathbb{H}_2 := \mathbb{G}^{\ell+1}, \quad \psi(r, x) := (r \cdot H + x \cdot G, [r \cdot P_i]_{i=1}^{\ell}), \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameters G, H and the public keys $P_1, \ldots, P_\ell$.

Batched version of the Grouped ciphertext validity

Later on in the confidential payment protocol, the system handles commitments that have been split into two parts C_{lo} and C_{hi}. We can reuse the previous sigma proof by using by considering the `lo` and `hi` ciphertexts as two separate instances of the grouped ciphertext validity proof.

Instead of proving and verifying them individually, the `zk-sdk` batches the instances using a verifier challenge. Concretely, we set

$$C = C_{\text{lo}} + t \cdot C_{\text{hi}} = x_{\text{lo}} G + r_{\text{lo}} H + t \cdot (x_{\text{hi}} G + r_{\text{hi}} H)$$

where t is a verifier challenge (sampled after observing the independent commitments), and using the same aggregation trick on the decryption handles as well

$$D_i = D_{\text{lo},i} + t \cdot D_{\text{hi},i} = r_{\text{lo}} P_i + t \cdot r_{\text{hi}} P_i.$$

The values C and D_i are then proven using the grouped ciphertext validity proof as above. We discuss the security implications of this type of batching in A note on batching.

Ciphertext-commitment equality

Given a system parameter H and Pedersen commitments $C_{\mathsf{EG}}, C_{\mathsf{Ped}}$, the ciphertext-commitment equality protocol proves knowledge of a secret key s, a message x and an opening r such that $\psi(s, x, r) = (H, C_{\mathsf{EG}}, C_{\mathsf{Ped}})$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism $\psi : \mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := (\mathbb{Z}_p)^3, \quad \mathbb{H}_2 := \mathbb{G}^3, \quad \psi(s, x, r) := (s \cdot P_{\mathsf{EG}}, x \cdot G + s \cdot D_{\mathsf{EG}}, x \cdot G + r \cdot H), \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameters G, H, the public key P_{EG} and the decryption handle D_{EG}.

Ciphertext-ciphertext equality

Given a system parameter H, Pedersen commitments C_0, C_1 and a decryption handle D_0, the ciphertext-commitment equality protocol proves knowledge of a secret key s, a message x and an opening r such that $\psi(s, x, r) = (H, C_0, C_1, D_1)$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism $\psi : \mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := (\mathbb{Z}_p)^3, \quad \mathbb{H}_2 := \mathbb{G}^4, \quad \psi(s, x, r) := (s \cdot P_0, x \cdot G + s \cdot D_0, x \cdot G + r \cdot H, r \cdot P_1), \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameters G, H, the public keys P_0, P_1 and the decryption handle D_0.

Percentage with cap

The percentage with cap protocol is a composition of two Sigma protocols, where the verifier asserts that the prover knew a satisfying witness to *either* instance. We once again refer readers to Boneh and Shoup chapter 19.7.2 for further exposition. Special-soundness and special HVZK of the OR-composition follows from special-soundness and special HVZK of the individual components.

Fee is a percentage. Given a fee commitment C_{fee}, an amount commitment C_{amt}, a basis point parameter bp and a claim C_{claimed}, the first branch of the percentage with cap protocol defines

$$C_{\mathsf{delta}} := C_{\mathsf{fee}} \cdot 10000 - C_{\mathsf{amt}} \cdot \mathsf{bp},$$

and proves knowledge of a message x and openings $r_{\mathsf{delta}}, r_{\mathsf{claimed}}$ such that $\psi(x, r_{\mathsf{delta}}, r_{\mathsf{claimed}}) = (C_{\mathsf{delta}}, C_{\mathsf{claimed}})$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism $\psi : \mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := (\mathbb{Z}_p)^3, \quad \mathbb{H}_2 := \mathbb{G}^2, \quad \psi(x, r_{\mathsf{delta}}, r_{\mathsf{claimed}}) := (x \cdot G + r_{\mathsf{delta}} \cdot H, x \cdot G + r_{\mathsf{claimed}} \cdot H), \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameters G, H.

Fee is capped. Given a fee commitment C_{fee}, a maximum fee $\mathsf{max}_{\mathsf{fee}}$ and a system parameter G, the second branch of the percentage with cap protocol defines $D := C_{\mathsf{fee}} - \mathsf{max}_{\mathsf{fee}} \cdot G$ and proves knowledge of an opening r such that $\psi(r) = D$ where the groups $\mathbb{H}_1$, $\mathbb{H}_2$, the homomorphism $\psi : \mathbb{H}_1 \to \mathbb{H}_2$ and challenge set $\mathscr{C}$ are defined as:

$$\mathbb{H}_1 := \mathbb{Z}_p, \quad \mathbb{H}_2 := \mathbb{G}, \quad \psi(r) := r \cdot H, \quad \mathscr{C} := \mathbb{Z}_p.$$

Note that ψ is defined with respect to the system parameters H.

A note on batching

It is a common strategy to reduce proof size or verifier work to employ *randomized batching*, either of instances or of the verifier's equations. The `zk-sdk` codebase employs both forms: it batches instances of the grouped ciphertext validity proof (see Batched version of the grouped ciphertext validity proof) and batches verifier checks for all sigma protocols that assert more than one equality.

In both cases, the resulting interactive protocol has one additional round and requires its own analysis. As noted by Attema, Fehr and Resch, these batching techniques do not always result in special-sound protocols (in the strict sense that the extractor must always succeed). Analyzing the resulting protocol and its behavior under the FS transform requires stronger tools than the ones used above. A formal security analysis is outside the scope of this report. Nonetheless, these techniques are widely deployed and we do not foresee security vulnerabilities associated with their application.

Range Proof Overview

A range proof is a protocol to show that a committed value v is in the range $[0, \mathsf{max})$. The confidential payments system uses this to ensure that balances, fee and transfer amounts are correctly capped. We give an overview of the concrete range proof protocol implemented in the zk-sdk.

At a high level, the protocol works by showing that the binary decomposition of v has a fixed length. Proving such a statement immediately gives a range proof when the value max is a power of 2. Our overview will first focus on this simple case. We then show how the protocol can be used for maxima that are not powers of two.

Generators Disambiguation

Note that in this section we use different generators, which we attempt to disambiguate here.

Pedersen Commitments. Pedersen commitments are obtained from a value x and blinding factor/opening r by computing $xG + rH$, with generator G (B in the Dalek library) and the blinding point H (resp. $\tilde{B}$). The base point G is the standard Curve25519 base point chosen as the point with coordinate $x = 9$:

```
/// The Ed25519 basepoint, as an `EdwardsPoint`.
///
/// This is called `_POINT` to distinguish it from
/// `ED25519_BASEPOINT_TABLE`, which should be used for scalar
/// multiplication (it's much faster).
pub const ED25519_BASEPOINT_POINT: EdwardsPoint = EdwardsPoint {
    X: FieldElement51::from_limbs([
        1738742601995546,
        1146398526822698,
        2070867633025821,
        562264141797630,
        587772402128613,
    ]),
    Y: FieldElement51::from_limbs([
        1801439850948184,
        1351079888211148,
        450359962737049,
        900719925474099,
        1801439850948198,
    ]),
    Z: FieldElement51::from_limbs([1, 0, 0, 0, 0]),
    T: FieldElement51::from_limbs([
        1841354044333475,
        16398895984059,
        755974180946558,
        900171276175154,
        1821297809914039,
    ]),
};

// ...

/// The Ristretto basepoint, as a `RistrettoPoint`.
///
/// This is called `_POINT` to distinguish it from `_TABLE`, which
/// provides fast scalar multiplication.
pub const RISTRETTO_BASEPOINT_POINT: RistrettoPoint = RistrettoPoint(ED25519_BASEPOINT_POINT);

// ...

/// Pedersen base point for encoding messages to be committed.
pub const G: RistrettoPoint = RISTRETTO_BASEPOINT_POINT;
```

while the point H is derived via another hash-to-curve based on a hash of the standard Ristretto basepoint:

```rust
/// The Ristretto basepoint, in `CompressedRistretto` format.
pub const RISTRETTO_BASEPOINT_COMPRESSED: CompressedRistretto = CompressedRistretto([
    0xe2, 0xf2, 0xae, 0x0a, 0x6a, 0xbc, 0x4e, 0x71, 0xa8, 0x84, 0xa9, 0x61, 0xc5, 0x00, 0x51, 0x5f,
    0x58, 0xe3, 0x0b, 0x6a, 0xa5, 0x82, 0xdd, 0x8d, 0xb6, 0xa6, 0x59, 0x45, 0xe0, 0x8d, 0x2d, 0x76,
]);

/// Pedersen base point for encoding the commitment openings.
pub static H: std::sync::LazyLock<RistrettoPoint> = std::sync::LazyLock::new(|| {
    RistrettoPoint::hash_from_bytes::<Sha3_512>(RISTRETTO_BASEPOINT_COMPRESSED.as_bytes())
});
```

IPA's output point Q. The IPA protocol produces a randomized basepoint to aggregate the result of the inner product into the commitment to the inputs of the inner product. This point Q is computed as $Q = wG$ (or wB in the Dalek documentation) with a random challenge w.

IPA generators $\mathbf{H}_{\text{IPA}}$. The IPA protocol has a second set of generators in order to commit (in a non-hiding way) to the inputs of the inner product. The left input vector uses the vector of points $\mathbf{G}_{\text{IPA}}$, while the right input vector uses the generators $\mathbf{H}_{\text{IPA}}$. All of these generators are independent and computed using a hash-to-curve algorithm.

IPA generators in the range proof protocol. As part of the range proof protocol, the base $\mathbf{H}_{\text{IPA}}$ is not used as is to commit to the second input of the inner product; instead the protocol uses the bases $\mathbf{H'}$ which are formed from shifts with the vector $\mathbf{y^n}$ of powers of y: $\mathbf{H'} = \mathbf{y^n} \circ \mathbf{H}_{\text{IPA}}$.

Aggregating Range Proofs Into a Single Check

The implementation aggregates multiple range proofs into a single proof. This aggregation technique is explained in section 4.3 "Aggregating Logarithmic Proofs" of the Bulletproofs paper. Since the implementation is built on top of the Dalek implementation, we reexplain the aggregation following Dalek's notation.

As a reminder, the vanilla equation to prove that one value v is in the range $[0, 2^n)$ is the following:

$$z^2 v = z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle$$

In the aggregated case, we prove m values $v_0, \ldots, v_{m-1}$ with bit-lengths $n_0, \ldots, n_{m-1}$. Each has a bit vector $\mathbf{a}_L^{(i)} \in \{0,1\}^{n_i}$ and $\mathbf{a}_R^{(i)} = \mathbf{a}_L^{(i)} - \mathbf{1}$.

The heart of the change is rather straightforward: instead of looking at different series of bits and independently performing the two bit-related constraints on them (check that $\mathbf{a}_R$ is computed correctly, check that each vector entry is 0 or 1), we can simply concatenate the vectors and check them as a single longer vector:

- $\mathbf{a}_L = \mathbf{a}_L^{(0)} \| \cdots \| \mathbf{a}_L^{(m-1)}$
- $\mathbf{a}_R = \mathbf{a}_R^{(0)} \| \cdots \| \mathbf{a}_R^{(m-1)}$

with total length $N = \sum_i n_i$.

We can similarly extend the vector $\mathbf{y}^N$ to cover the whole concatenated vector. The batched check becomes

$$\sum_{i=0}^{m-1} z^{i+2} v_i = \sum_{i=0}^{m-1} z^{i+2} \langle \mathbf{a}_L^{(i)}, \mathbf{2}^{n_i} \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^N \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^N \rangle.$$

Define the blockwise offset vector

$$\mathbf{Z}' = (z^2 \mathbf{2}^{n_0}) \,\|\, (z^3 \mathbf{2}^{n_1}) \,\|\, \cdots \,\|\, (z^{m+1} \mathbf{2}^{n_{m-1}}),$$

so that

$$\sum_{i=0}^{m-1} z^{i+2} \langle \mathbf{a}_L^{(i)}, \mathbf{2}^{n_i} \rangle = \langle \mathbf{a}_L, \mathbf{Z}' \rangle.$$

The check then becomes

$$\sum_{i=0}^{m-1} z^{i+2} v_i + z \langle \mathbf{1}, \mathbf{y}^N \rangle = \langle \mathbf{a}_L, \mathbf{Z}' + z \mathbf{y}^N \rangle - z \langle \mathbf{a}_R, \mathbf{y}^N \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^N \rangle.$$

Add $\langle -z\mathbf{1}, \mathbf{Z}' + z\mathbf{y}^N \rangle$ to both sides to complete the square:

$$\sum_{i=0}^{m-1} z^{i+2} v_i + (z - z^2) \langle \mathbf{1}, \mathbf{y}^N \rangle - \sum_{i=0}^{m-1} z^{i+3} \langle \mathbf{1}, \mathbf{2}^{n_i} \rangle = \langle \mathbf{a}_L - z\mathbf{1},\ \mathbf{y}^N \circ (\mathbf{a}_R + z\mathbf{1}) + \mathbf{Z}' \rangle.$$

Thus the unblinded vectors are

$$\mathbf{l} = \mathbf{a}_L - z\mathbf{1}, \qquad \mathbf{r} = \mathbf{y}^N \circ (\mathbf{a}_R + z\mathbf{1}) + (z^2 \mathbf{2}^{n_0} \| \cdots \| z^{m+1} \mathbf{2}^{n_{m-1}}),$$

with

$$\langle \mathbf{l}, \mathbf{r} \rangle = \sum_{i=0}^{m-1} z^{i+2} v_i + (z - z^2)\langle \mathbf{1}, \mathbf{y}^N \rangle - \sum_{i=0}^{m-1} z^{i+3} \langle \mathbf{1}, \mathbf{2}^{n_i} \rangle.$$

Finally, to make it zero-knowledge, sample blinding vectors $\mathbf{s}_L, \mathbf{s}_R$ and define

$$\mathbf{l}(x) = (\mathbf{a}_L - z\mathbf{1}) + \mathbf{s}_L x, \qquad \mathbf{r}(x) = \mathbf{y}^N \circ (\mathbf{a}_R + z\mathbf{1} + \mathbf{s}_R x) + (z^2 \mathbf{2}^{n_0} \| \cdots \| z^{m+1} \mathbf{2}^{n_{m-1}}).$$

You can see that in the implementation:

```
// 4. Construct the blinded vector polynomials l(x) and r(x).
//    l(x) = (a_L - z*1) + s_L*x
//    r(x) = y^nm o (a_R + z*1 + s_R*x) + (z^2*2^n_1 || ... || z^{m+1}*2^n_m)
//    where `o` is the Hadamard product and `||` is vector concatenation.
let mut l_poly = util::VecPoly1::zero(nm);
let mut r_poly = util::VecPoly1::zero(nm);

let mut i = 0;
let mut exp_z = z * z;
let mut exp_y = Scalar::ONE;

for (amount_i, n_i) in amounts.iter().zip(bit_lengths.iter()) {
    let mut exp_2 = Scalar::ONE;

    for j in 0..(*n_i) {
        // `j` is guaranteed to be at most `u64::BITS` (a 6-bit number) and therefore,
        // casting is lossless and right shift can be safely unwrapped
        let a_L_j = Scalar::from(amount_i.checked_shr(j as u32).unwrap() & 1);
        let a_R_j = a_L_j - Scalar::ONE;

        l_poly.0[i] = a_L_j - z;
        l_poly.1[i] = s_L[i];
        r_poly.0[i] = exp_y * (a_R_j + z) + exp_z * exp_2;
        r_poly.1[i] = exp_y * s_R[i];

        exp_y *= y;
        exp_2 = exp_2 + exp_2;

        // `i` is capped by the sum of vectors in `bit_lengths`
        i = i.checked_add(1).unwrap();
    }
    exp_z *= z;
}
```

Merging The Range Proof With IPA

In the Dalek range-proof write-up (https://doc-internal.dalek.rs/bulletproofs/notes/range_proof/index.html#proving-that-mathbflx-mathbfrx-are-correct), the verifier constructs the IPA input as:

$$P \;=\; -\tilde{e}\,\tilde{B} \;+\; A \;+\; x\,S \;+\; \langle z\,\mathbf{y}^N + \mathbf{Z}',\, \mathbf{H}' \rangle \;-\; z\,\langle \mathbf{1}, \mathbf{G} \rangle,$$

which is algebraically the same as $\langle \mathbf{l}(x), \mathbf{G} \rangle + \langle \mathbf{r}(x), \mathbf{H}' \rangle$ for the aggregated case (with the blockwise offset $\mathbf{Z}'$).

The prover also provides the scalar evaluation $t(x)$, which presumably represents the result of the inner product.

Everything is evaluated at a challenge x: the vectors are polynomials in x but the IPA itself runs on the **evaluated** vectors $\mathbf{l}(x), \mathbf{r}(x)$ (i.e., after instantiating x via Fiat–Shamir). With these in hand, the verifier runs the **standard IPA verification** as we describe here:

1. We have the IPA input

$$P \;=\; \langle \ell(x),\, \mathbf{G} \rangle \;+\; \langle r(x),\, \mathbf{H}' \rangle.$$

2. Add the claimed inner-product output:

$$P \;+\; t(x)\,Q \;=\; \langle \ell(x), \mathbf{G} \rangle \;+\; \langle r(x), \mathbf{H}' \rangle \;+\; \langle \ell(x), r(x) \rangle Q.$$

3. Add the IPA folding terms (one round per i, with transcript-derived u_i) to produce a **reduced** commitment of the statement:

$$P \;+\; t(x)\,Q \;+\; \sum_i \left(u_i^2 L_i \;+\; u_i^{-2} R_i\right) \;=\; \langle \ell^\star,\, G^\star \rangle \;+\; \langle r^\star,\, H^\star \rangle \;+\; \langle \ell^\star, r^\star \rangle Q,$$

where after all folds we are down to scalars $\ell^\star = a$, $r^\star = b$ and the **reduced bases** $G^\star, H^\star$ are known transcript-dependent combinations of the original bases (think $G^\star = \langle \mathbf{s}, \mathbf{G} \rangle$, $H^\star = \langle \mathbf{s}^{-1}, \mathbf{H}' \rangle$ for challenge-derived weights $\mathbf{s}$).

4. Conclude with the reduced opening (the prover reveals a, b):

$$\underbrace{P}_{\text{commitment input}} + \underbrace{t(x)\,Q}_{\text{commitment output}} + \underbrace{\sum_i \left(u_i^2 L_i + u_i^{-2} R_i\right)}_{\text{term to produce the reduction}} = \underbrace{a\,G^\star + b\,H^\star + a\,b\,Q}_{\text{reduced form}}$$

Now we substitute the **concrete form** of P from the aggregated range proof:

$$P = -e_{\text{blind}} H + A + x\,S + \langle z\,\mathbf{y}^N + \mathbf{Z}',\, \mathbf{H}' \rangle - z\,\langle \mathbf{1}, \mathbf{G} \rangle.$$

After substitution, the verifier actually checks:

$$\left(-e_{\text{blind}} H + A + xS + \langle z\,\mathbf{y}^N + \mathbf{Z}', \mathbf{H}' \rangle - z\langle \mathbf{1}, \mathbf{G} \rangle\right) + t(x)Q + \sum_i \left(u_i^2 L_i + u_i^{-2} R_i\right) = aG^\star + bH^\star + abQ.$$

On the **left**, we've replaced P with its concrete commitment built from $(A, S, e_{\text{blind}}, z, y, \mathbf{Z}')$, and added $t(x)Q$ plus the IPA folding terms. On the **right**, the verifier expects the reduced opening: the prover's revealed scalars a, b multiplied with the reduced bases $G^\star$, $H^\star$, plus abQ.

Final Check of the Implemented Protocol

The verifier checks that the following multiscalar multiplication equals the identity:

```
let mega_check = RistrettoPoint::optional_multiscalar_mul(
    // A
    iter::once(Scalar::ONE)
        // + x * S
        .chain(iter::once(x))
        // + d ( x * T_1 )
        .chain(iter::once(d * x))
        // + d ( x^2 * T_2 )
        .chain(iter::once(d * x * x))
        // - e_blinding * H
        // - d ( t_x_blinding * H )
        .chain(iter::once(-self.e_blinding - d * self.t_x_blinding))
        // + w * (t_x - a * b) * G
        // + d (delta(bitlengths, y, z) - t_x) * G
        .chain(iter::once(basepoint_scalar))
        // + <x_sq, L>
        .chain(x_sq.iter().cloned())
        // + <x_inv_sq, R>
        .chain(x_inv_sq.iter().cloned())
        // - z * <1, G_IPA> - a <s, G_IPA>
        .chain(gs)
        // + <z y^n + Z', H_IPA> - b H_IPA
        .chain(hs)
        // + d (\sum_i z^{2+i} * V_i)
        .chain(value_commitment_scalars),
    iter::once(self.A.decompress())
        .chain(iter::once(self.S.decompress()))
        .chain(iter::once(self.T_1.decompress()))
        .chain(iter::once(self.T_2.decompress()))
        .chain(iter::once(Some(*H)))
        .chain(iter::once(Some(G)))
        .chain(self.ipp_proof.L_vec.iter().map(|L| L.decompress()))
        .chain(self.ipp_proof.R_vec.iter().map(|R| R.decompress()))
        .chain(bp_gens.G(nm).map(|&x| Some(x)))
        .chain(bp_gens.H(nm).map(|&x| Some(x)))
        .chain(comms.iter().map(|V| Some(*V.get_point()))),
)
.ok_or(RangeProofVerificationError::MultiscalarMul)?;
```

The verifier checks that the following multiscalar multiplication equals the identity:

$$
\begin{aligned}
MSM = \; & 1 \cdot A \\
& + x \cdot S \\
& + d\,x \cdot T_1 \\
& + d\,x^2 \cdot T_2 \\
& + \left(-e_{\text{blind}} - d\,t_x^{(\text{blind})}\right) \cdot H \\
& + \left(w\,(t_x - a\,b) + d\,(\delta - t_x)\right) \cdot G \\
& + \textstyle\sum_i u_i^2 \cdot L_i \\
& + \textstyle\sum_i u_i^{-2} \cdot R_i \\
& + \textstyle\sum_j \left(-z - a\,s_j\right) \cdot G_{\texttt{IPA},j} \\
& + \textstyle\sum_j \left(z + y^{-j-1}\left(Z'[j] - b\,s_j^{-1}\right)\right) \cdot H_{\texttt{IPA},j} \\
& + \textstyle\sum_k d\,z^{k+2} \cdot V_k \\
& = \mathcal{O}.
\end{aligned}
$$

We can see more clearly what this “mega check” verifies by splitting the checks that are not shifted by the challenge d as:

$$
\begin{aligned}
\text{MSM}_{\text{free}} = \; & 1 \cdot A \\
& + x \cdot S \\
& - e_{\text{blind}} \cdot H \\
& + w\,(t_x - a\,b) \cdot G \\
& + \textstyle\sum_i u_i^2 \cdot L_i \\
& + \textstyle\sum_i u_i^{-2} \cdot R_i \\
& + \textstyle\sum_j \left(-z - a\,s_j\right) \cdot G_{\texttt{IPA},j} \\
& + \textstyle\sum_j \left(z + y^{-j-1}\left(Z'[j] - b\,s_j^{-1}\right)\right) \cdot H_{\texttt{IPA},j}.
\end{aligned}
$$

And the checks that are shifted by the challenge d as:

$$
\begin{aligned}
\text{MSM}_d = \; & x \cdot T_1 \\
& + x^2 \cdot T_2 \\
& - t_x^{(\text{blind})} \cdot H \\
& + (\delta - t_x) \cdot G \\
& + \textstyle\sum_k z^{k+2} \cdot V_k.
\end{aligned}
$$

So the check is aggregated as

$$
\text{MSM}_{\text{free}} + d \cdot \text{MSM}_d = \mathcal{O}.
$$

When max is not a Power of Two

To apply the range proof for $v \in [0, \mathsf{max})$ in the case where max is not a power of two, we first compute a complement value v' and then show that v and v' are in the range $[0, 2^k)$, where k is chosen such that $\mathsf{max} < 2^k$.

Specifically, the complement value v' is computed as:

$$
v' := (\mathsf{max} - 1) - v.
$$

We can see why constraining v and v' prove the desired statement by considering the following series of implications:

$$
\begin{aligned}
& 0 \leq v' < 2^k \\
\iff & 0 \leq (\mathsf{max} - 1) - v \leq 2^k - 1 \\
\iff & 2^k - 1 \leq v - (\mathsf{max} - 1) \leq 0 \\
\iff & 2^k - 1 + \mathsf{max} - 1 \leq v \leq \mathsf{max} - 1
\end{aligned}
$$

Combining this and the other range for v, we conclude that $v \in [0, \mathsf{max})$.

Overview of the ZK Elgamal Proof Program

The verifiers for the different zero-knowledge proofs implemented in the zk-sdk are exposed in Solana as the built-in ZK ElGamal Proof program with program id `zkE1Gama1Proof11111111111111111111111111111`.

The built-in is declared in the agave node code in `builtins/src/lib.rs`:

```
pub static BUILTINS: &[BuiltinPrototype] = &[
    // TRUNCATED...
    testable_prototype!(BuiltinPrototype {
        core_bpf_migration_config: None,
        name: zk_elgamal_proof_program,
        enable_feature_id: Some(feature_set::zk_elgamal_proof_program_enabled::id()),
        program_id: solana_sdk_ids::zk_elgamal_proof_program::id(),
        entrypoint: solana_zk_elgamal_proof_program::Entrypoint::vm,
    }),
];
```

where the entrypoint referenced is generated via a macro and is implemented to dispatch different instructions:

```
declare_process_instruction!(Entrypoint, 0, |invoke_context| {
    if invoke_context
        .get_feature_set()
        .disable_zk_elgamal_proof_program
        && !invoke_context
            .get_feature_set()
            .reenable_zk_elgamal_proof_program
    {
        ic_msg!(
            invoke_context,
            "zk-elgamal-proof program is temporarily disabled"
        );
        return Err(InstructionError::InvalidInstructionData);
    }

    let transaction_context = &invoke_context.transaction_context;
    let instruction_context = transaction_context.get_current_instruction_context()?;
    let instruction_data = instruction_context.get_instruction_data();
    let instruction = ProofInstruction::instruction_type(instruction_data)
        .ok_or(InstructionError::InvalidInstructionData)?;

    match instruction {
        ProofInstruction::CloseContextState => {
            invoke_context
                .consume_checked(CLOSE_CONTEXT_STATE_COMPUTE_UNITS)
                .map_err(|_| InstructionError::ComputationalBudgetExceeded)?;
            ic_msg!(invoke_context, "CloseContextState");
            process_close_proof_context(invoke_context)
        }
        ProofInstruction::VerifyZeroCiphertext => {
            invoke_context
                .consume_checked(VERIFY_ZERO_CIPHERTEXT_COMPUTE_UNITS)
                .map_err(|_| InstructionError::ComputationalBudgetExceeded)?;
            ic_msg!(invoke_context, "VerifyZeroCiphertext");
            process_verify_proof::<ZeroCiphertextProofData, ZeroCiphertextProofContext>(
                invoke_context,
            )
        }
        // TRUNCATED...
```

As the instructions are implemented natively, the compute units (on-chain cost of running these instructions) are hardcoded as follows:

```
pub const CLOSE_CONTEXT_STATE_COMPUTE_UNITS: u64 = 3_300;
pub const VERIFY_ZERO_CIPHERTEXT_COMPUTE_UNITS: u64 = 6_000;
pub const VERIFY_CIPHERTEXT_CIPHERTEXT_EQUALITY_COMPUTE_UNITS: u64 = 8_000;
pub const VERIFY_CIPHERTEXT_COMMITMENT_EQUALITY_COMPUTE_UNITS: u64 = 6_400;
pub const VERIFY_PUBKEY_VALIDITY_COMPUTE_UNITS: u64 = 2_600;
pub const VERIFY_PERCENTAGE_WITH_CAP_COMPUTE_UNITS: u64 = 6_500;
pub const VERIFY_BATCHED_RANGE_PROOF_U64_COMPUTE_UNITS: u64 = 111_000;
pub const VERIFY_BATCHED_RANGE_PROOF_U128_COMPUTE_UNITS: u64 = 200_000;
pub const VERIFY_BATCHED_RANGE_PROOF_U256_COMPUTE_UNITS: u64 = 368_000;
pub const VERIFY_GROUPED_CIPHERTEXT_2_HANDLES_VALIDITY_COMPUTE_UNITS: u64 = 6_400;
pub const VERIFY_BATCHED_GROUPED_CIPHERTEXT_2_HANDLES_VALIDITY_COMPUTE_UNITS: u64 = 13_000;
pub const VERIFY_GROUPED_CIPHERTEXT_3_HANDLES_VALIDITY_COMPUTE_UNITS: u64 = 8_100;
pub const VERIFY_BATCHED_GROUPED_CIPHERTEXT_3_HANDLES_VALIDITY_COMPUTE_UNITS: u64 = 16_400;
```

There are essentially two types of instructions:

1. The verify functions, which allow you to verify different types of zero-knowledge proofs
2. A close context state function, which allows you to close a storage account meant to be a temporary place to hold claims, where a claim is an on-chain evidence that a specific proof was verified (think the type of the proof and the instance)

All proofs follow the same format, which is a structure that includes both the statement/instance and the proof. All proof types implement the following trait, where `Pod` is the atomic type being handled by the SVM, and `context_data` and the type `T` refer to the statement/instance associated with a ZKP:

```
pub trait ZkProofData<T: Pod> {
    const PROOF_TYPE: ProofType;

    fn context_data(&self) -> &T;

    #[cfg(not(target_os = "solana"))]
    fn verify_proof(&self) -> Result<(), ProofVerificationError>;
}
```

Verification of a proof comes with two knobs that one can tweak:

1. A proof can be provided directly as part of the instruction data, or it can be provided as part of some arbitrary account's data (a "proof_data_account"), given an additional offset in that data (allowing multiple proofs to be stored in temporary storage accounts).
2. An externally-allocated "context state" account can be provided in order to store a claim (the result of the verification) to be used by another instruction. If this account is not provided, the result is short-lived as it will have to be accessed by another instruction from the same transaction (using instruction introspection).

Overview of the Token 2022 Confidential Transfer Extension

Confidential transfers in Solana are added on top of the token-2022 standard as extensions, which we go into more detail in this section. Essentially, base mint accounts are augmented with confidential transfer extensions to advertise that confidential transfers are enabled; from there on, token accounts created for the mint are also augmented with their own confidential transfer extensions which will store the account's encrypted balances (in the style of zether).

Verifying Proofs And Extracting Instances

As seen in the previous section, proofs relevant to the Token-2022 confidential transfer extensions are verified via the ZK ElGamal Proof program. When the confidential extensions want to enforce some zero-knowledge proofs and extract their instances/statements, they have to either:

1. find if a ZK ElGamal Proof instruction for a specific proof type is present, and if so extract the statement associated with that proof within its associated instruction data
2. find a "proof context account" owned by the ZK ElGamal Proof program that contain the correct proof type, and extract the statement (called context data) stored next to the proof type under that account data

Both ways ensure that the proof (described uniquely by its type and its context data) has been correctly verified, because if this wasn't the case the whole transaction would have been reverted (in the first case) or no proof context account owned by the ZK ElGamal Proof program would exist.

Accounts and Extensions in Token-2022

Accounts owned by the Token-2022 program have an **implicit** type (which is known due to the length of the data field itself, for example, a multisig account is always 355 bytes), which is made **explicit** via the encoding of an `AccountType` if their data field also contains extensions.

There are two main `AccountType`s: `Mint` and `Account`. A single `Mint` is used to represent a specific mint (i.e. a token), whereas many `Account`s are used to represent individual user accounts associated to a mint. The function `unpack` is used to load these from the `data` field of Solana accounts while ensuring that the `AccountType` is correctly set in case of the presence of extensions:

```
// checks that the token_account_data account stores an Account account type (if it has any extension data)
let mut token_account = PodStateWithExtensionsMut::<PodAccount>::unpack(token_account_data)?;

// checks that the mint_data account stores a Mint account type (if it has any extension data)
let mut mint = PodStateWithExtensionsMut::<PodMint>::unpack(mint_data)?;
```

Note also that an `unpack_uninitialized` can be used to ensure that the base type is filled with zeros (uninitialized), as extensions are meant to be initialized before a base type is (effectively locking any extension initialization thereafter).

Visually, extensions are encoded (using a Tag-Length-Value encoding) in the data field of an account containing a base type `Mint` or `Account` in the following way:

Instructions

The Token-2022 program processes an instruction by deserializing the first byte of the input and routing the instruction to the correct implementation. As implemented, it recognizes three main confidential instructions:

```rust
pub fn process(program_id: &Pubkey, accounts: &[AccountInfo], input: &[u8]) -> ProgramResult {
    if let Ok(instruction_type) = decode_instruction_type(input) {
        match instruction_type {
            // TRUNCATED...
            PodTokenInstruction::ConfidentialTransferExtension => {
                confidential_transfer::processor::process_instruction(
                    program_id,
                    accounts,
                    &input[1..],
                )
            }
            // TRUNCATED...
            PodTokenInstruction::ConfidentialTransferFeeExtension => {
                confidential_transfer_fee::processor::process_instruction(
                    program_id,
                    accounts,
                    &input[1..],
                )
            }
            // TRUNCATED...
            PodTokenInstruction::ConfidentialMintBurnExtension => {
                msg!("Instruction: ConfidentialMintBurnExtension");
                confidential_mint_burn::processor::process_instruction(
                    program_id,
                    accounts,
                    &input[1..],
                )
            }
```

The second byte then tells us about subinstructions. Let's first take a look at the main three confidential instructions:

ConfidentialTransfer. This instruction manages the two main confidential transfer extensions: one to manage a confidential transfers configuration on the mint, and one to manage the confidential transfers on individual token accounts. It can be used without the two other extensions, and contain all the required subinstructions from managing the mint to handling transfers between token accounts of the mint.

ConfidentialTransferFee. This instruction contains the logic to initialize the `ConfidentialTransferFeeConfig` extension in a mint, which is required when fees are enabled on the base mint (via another base fee extension). The subinstructions defined there allow "harvesting" fees, and sending these to a destination account (if authorized by the base fee extension authority). In addition, a confidential fee authority is set to allow rotation of its Elgamal public key (which is needed as fees are encrypted).

ConfidentialMintBurn. Finally, a mint can be configured via a confidential mint-burn extension to only allow for direct minting of confidential balances, disallowing the existence of a base token. Without this extension, minting new tokens has to happen on the base mint, and then transfered to the "shielded world" via the confidential transfer deposit subinstruction. (Note that even without this extension, the confidential transfer can choose to disallow non-confidential transfers.)

In the next sections we cover each of the main confidential instructions in more detail.

Confidential Transfer Instruction

The confidential transfer instruction is the first instruction for confidential transfer, and it can be used by itself without the other two. It consists of 13 subinstructions that are either about initializing and managing a mint, or about initializing and managing token accounts. The latter also includes transferring between accounts.

Activating this extension creates a "two-worlds system" with a base balance (in the base type) and an encrypted balance (in the extension). Allowing base → encrypted movements via the Deposit subinstruction and encrypted → base movements via the Withdraw subinstruction.

Here are its subinstructions:

1. **InitializeMint** - Set up confidential transfer settings (authority, auditor, auto-approve) on a new mint
2. **UpdateMint** - Change auto-approve setting and auditor ElGamal key on existing mint
3. **ApproveAccount** - Mint authority approves a configured account to use confidential transfers
4. **ConfigureAccount** - Set up an account's ElGamal public key and enable confidential transfers
5. **EmptyAccount** - Withdraw all available balance to prepare account for closing
6. **Deposit** - Convert non-confidential tokens into confidential pending balance
7. **Withdraw** - Convert confidential available balance into non-confidential tokens
8. **Transfer** - Send confidential tokens from one account to another
9. **ApplyPendingBalance** - Move pending balance into available balance (activate received transfers)

10. **EnableConfidentialCredits** - Allow account to receive incoming confidential transfers
11. **DisableConfidentialCredits** - Reject incoming confidential transfers
12. **EnableNonConfidentialCredits** - Allow account to receive incoming non-confidential transfers
13. **DisableNonConfidentialCredits** - Reject incoming non-confidential transfers

A base mint, still uninitialized, can be configured with the following `ConfidentialTransferMint` extension in order to activate confidential transfers:

```rust
// --- token-2022/interface/src/extension/confidential_transfer/mod.rs ---

pub struct ConfidentialTransferMint {
    /// Authority to modify the `ConfidentialTransferMint` configuration and to
    /// approve new accounts (if `auto_approve_new_accounts` is true)
    ///
    /// The legacy Token Multisig account is not supported as the authority
    pub authority: OptionalNonZeroPubkey,

    /// Indicate if newly configured accounts must be approved by the
    /// `authority` before they may be used by the user.
    ///
    /// * If `true`, no approval is required and new accounts may be used
    ///   immediately
    /// * If `false`, the authority must approve newly configured accounts (see
    ///   `ConfidentialTransferInstruction::ConfigureAccount`)
    pub auto_approve_new_accounts: PodBool,

    /// Authority to decode any transfer amount in a confidential transfer.
    pub auditor_elgamal_pubkey: OptionalNonZeroElGamalPubkey,
}
```

Once the base mint is initialized, extensions cannot be changed. From there on, any token account initialized for this mint can also ask to receive a `ConfidentialTransferAccount` extension (through the `ConfigureAccount` extension) in order to enable the “encrypted world”:

```rust
pub struct ConfidentialTransferAccount {
    /// `true` if this account has been approved for use. All confidential
    /// transfer operations for the account will fail until approval is
    /// granted.
    pub approved: PodBool,

    /// The public key associated with ElGamal encryption
    pub elgamal_pubkey: PodElGamalPubkey,

    /// The low 16 bits of the pending balance (encrypted by `elgamal_pubkey`)
    pub pending_balance_lo: EncryptedBalance,

    /// The high 32 bits of the pending balance (encrypted by `elgamal_pubkey`)
    pub pending_balance_hi: EncryptedBalance,

    /// The available balance (encrypted by `encryption_pubkey`)
    pub available_balance: EncryptedBalance,

    /// The decryptable available balance
    pub decryptable_available_balance: DecryptableBalance,

    /// If `false`, the extended account rejects any incoming confidential
    /// transfers
    pub allow_confidential_credits: PodBool,

    /// If `false`, the base account rejects any incoming transfers
    pub allow_non_confidential_credits: PodBool,

    /// The total number of `Deposit` and `Transfer` instructions that have
    /// credited `pending_balance`
    pub pending_balance_credit_counter: PodU64,

    /// The maximum number of `Deposit` and `Transfer` instructions that can
    /// credit `pending_balance` before the `ApplyPendingBalance`
    /// instruction is executed
    pub maximum_pending_balance_credit_counter: PodU64,

    /// The `expected_pending_balance_credit_counter` value that was included in
    /// the last `ApplyPendingBalance` instruction
    pub expected_pending_balance_credit_counter: PodU64,

    /// The actual `pending_balance_credit_counter` when the last
    /// `ApplyPendingBalance` instruction was executed
    pub actual_pending_balance_credit_counter: PodU64,
}
```

Note that at this point, the account might still not be approved, depending on the mint configuration (which might require manual approving from the mint authority). A token account can still trade in the “base world” (via its non-encrypted balance available in the base type).

We summarize how the approval logic impacts the different functionalities of the program in the table below (ignoring `process_initialize_mint` and `process_update_mint` which are pure mint subinstructions):

SUBINSTRUCTION	APPROVAL
`ConfigureAccount`	Sets approved
`ConfigureAccountWithRegistry`	Sets approved
`ApproveAccount`	Sets approved=true
`EmptyAccount`	Ignores approval status
`Deposit`	Enforces approved account
`Withdraw`	Enforces approved account
`Transfer`	Enforces approved account

SUBINSTRUCTION	APPROVAL
`TransferWithFee`	Enforces approved account
`ApplyPendingBalance`	Ignores approval status
`EnableConfidentialCredits`	Ignores approval status
`DisableConfidentialCredits`	Ignores approval status
`EnableNonConfidentialCredits`	Ignores approval status
`DisableNonConfidentialCredits`	Ignores approval status

Note that confidential transfers still have to play nice with other (confidential and non-confidential) extensions, we summarize some of that in the following table:

SUB-INSTRUCTION	PAUSABLECONFIG	TRANSFERFEECONFIG	NONTRANSFERABLEACCOUNT
InitializeMint	N/A	N/A	N/A
UpdateMint	N/A	N/A	N/A
ConfigureAccount	N/A	Check if present for fee	N/A
ApproveAccount	N/A	N/A	N/A
EmptyAccount	N/A	N/A	N/A
Deposit	Check if paused	N/A	Block if present
Withdraw	Check if paused	N/A	Block if present
Transfer	Check if paused	Check presence for fee logic	Block source if present
TransferWithFee	Check if paused	Get fee parameters	Block source if present
ApplyPendingBalance	N/A	N/A	N/A
DisableConfidentialCredits	N/A	N/A	N/A
EnableConfidentialCredits	N/A	N/A	N/A
DisableNonConfidentialCredits	N/A	N/A	N/A
EnableNonConfidentialCredits	N/A	N/A	N/A

Finally, note that another “ElGamal Registry” program is also implemented in order to facilitate creating a registry of *pubkey validity proofs* (which are proofs that someone knows the private key associated with the public key). In this registry, anyone can post a link between any (valid) public key to an actual Solana system account. This permits someone that does not know your ElGamal private key to still create a confidential token account (presumably for a new confidential mint) for you, as the token account creation subinstruction requires the sigma proof.

Confidential Transfer Fee

The second confidential instruction is used only when the base mint has transfer fees enabled (via a base transfer fee extension). This extension allows fees to be calculated (via the OR sigma proof explained previously) and withheld from the transaction amount. To do that, the previous instruction is configured to detect the presence of the transfer fee extension in the mint, and add a `ConfidentialTransferFeeAmount` extension to any token account being configured for confidential transfer. The role of that new extension is to hold fees collected in all transfers sent to that account, until they get harvested by a mint authority, which is what this instruction is about:

it helps create and manage that authority, as well as harvest the fees from many accounts and deposit them into arbitrary accounts. Here are its subinstructions:

1. **InitializeConfidentialTransferFeeConfig** - Set up confidential fee configuration with withdrawal authority ElGamal key
2. **EnableHarvestToMint** and **DisableHarvestToMint** - Pause or unpause the harvesting of fees
3. **HarvestWithheldTokensToMint** - Move withheld fees from accounts to mint (permissionless batching operation)
4. **WithdrawWithheldTokensFromMint** - Send accumulated fees from mint to destination account (after harvesting them with the previous instruction)
5. **WithdrawWithheldTokensFromAccounts** - The two previous instructions can be compressed into a single one, avoiding the need to move the fees to the mint, by collecting and sending fees directly from multiple accounts to a destination account

A `ConfidentialTransferFeeConfig` extension must be added to the mint, if the mint is configured with the base transfer fee (and in fact, confidential transfer functions won't work if the mint wasn't initialized correctly with that extension):

```
// -- token-2022/interface/src/extension/confidential_transfer_fee/mod.rs --

pub struct ConfidentialTransferFeeConfig {
    /// Optional authority to set the withdraw withheld authority ElGamal key
    pub authority: OptionalNonZeroPubkey,

    /// Withheld fees from accounts must be encrypted with this ElGamal key.
    ///
    /// Note that whoever holds the ElGamal private key for this ElGamal public
    /// key has the ability to decode any withheld fee amount that are
    /// associated with accounts. When combined with the fee parameters, the
    /// withheld fee amounts can reveal information about transfer amounts.
    pub withdraw_withheld_authority_elgamal_pubkey: PodElGamalPubkey,

    /// If `false`, the harvest of withheld tokens to mint is rejected.
    pub harvest_to_mint_enabled: PodBool,

    /// Withheld confidential transfer fee tokens that have been moved to the
    /// mint for withdrawal.
    pub withheld_amount: EncryptedWithheldAmount,
}

pub struct ConfidentialTransferFeeAmount {
    /// Amount withheld during confidential transfers, to be harvested to the mint
    pub withheld_amount: EncryptedWithheldAmount,
}
```

Note that the design has fees deposited in recipient accounts, as opposed to always depositing them to the same "mint collecting fee" account, in order to avoid a situation where every transaction of that confidential mint locks the same account (which is unfavorable in the highly-parallelizable transaction execution model of Solana). This mirrors the design of the base transfer fee extension.

As with the previous confidential transfer instruction, this one also has to play nice with non-confidential extensions:

SUB-INSTRUCTION	TRANSFERFEECONFIG
InitializeConfidentialTransferFeeConfig	N/A
WithdrawWithheldTokensFromMint	Required for authority
WithdrawWithheldTokensFromAccounts	Required for authority
HarvestWithheldTokensToMint	Required for existence
EnableHarvestToMint	N/A
DisableHarvestToMint	N/A

Confidential Mint Burn

Finally, the last confidential instruction allows the minting of new tokens **confidentially**. Previously, tokens had to be minted through the base token, and deposited onto the confidential-side of the fence (and withdrawn back to the base-side of the fence). A mint configured with the confidential mint burn extension does not allow the existence of base tokens point blank. Here are its subinstructions:

1. **InitializeMint** - Set up confidential mint/burn extension with supply encryption key
2. **RotateSupplyElGamalPubkey** - Change the ElGamal key used to encrypt supply
3. **UpdateDecryptableSupply** - Update the AES-encrypted supply for authority's tracking
4. **Mint** - Create new tokens directly into an account's confidential pending balance
5. **Burn** - Destroy tokens from an account's confidential available balance
6. **ApplyPendingBurn** - Subtract accumulated burns from the confidential supply

The configuration extension struct looks like the following, keeping track of the total supply in an encrypted balance:

```
// --- token-2022/interface/src/extension/confidential_mint_burn/mod.rs ---

pub struct ConfidentialMintBurn {
    /// The confidential supply of the mint (encrypted by `encryption_pubkey`)
    pub confidential_supply: PodElGamalCiphertext,
    /// The decryptable confidential supply of the mint
    pub decryptable_supply: PodAeCiphertext,
    /// The ElGamal pubkey used to encrypt the confidential supply
    pub supply_elgamal_pubkey: PodElGamalPubkey,
    /// The amount of burn amounts not yet aggregated into the confidential supply
    pub pending_burn: PodElGamalCiphertext,
}
```

Minting acts similarly to normal transfers, where two recipients are created: the account the mint goes to, and the `confidential_supply` balance above.

Since everyone can "burn", the burning transfers have to be processed periodically by the mint authority similarly to pending balances in the main confidential transfer account extension.

As usual, the added instruction/extension has to play nice with non-confidential ones:

SUB-INSTRUCTION	PAUSABLECONFIG	TRANSFERFEECONFIG	NONTRANSFERABLEACCOUNT
InitializeMint	N/A	N/A	N/A
RotateSupplyElGamalPubkey	Check if paused	N/A	N/A
UpdateDecryptableSupply	N/A	N/A	N/A
Mint	Check if paused	N/A	N/A
Burn	Check if paused	N/A	N/A
ApplyPendingBurn	N/A	N/A	N/A

Initialization and Interaction Between Confidential Extensions

In total, there are five confidential extensions (implementing the `Extension` trait in the code) that can be encoded in mint or token accounts. Mint extensions have to be initialized before the base mint is initialized, whereas the confidential token extensions can be added on top of an initialized base token. This allows mints to honestly expose and maintain their configurations to users. We recapitulate where these extensions are initialized for the first time in the table below:

INSTRUCTION	SUBINSTRUCTION	TARGET	EXTENSION INITIALIZED
ConfidentialTransfer	InitializeConfidentialTransferMint	Mint	ConfidentialTransferMint
ConfidentialTransfer	ConfigureAccount / ConfigureAccountWithRegistry	Token Account	ConfidentialTransferAccount
ConfidentialTransfer	ConfigureAccount / ConfigureAccountWithRegistry	Token Account (conditional)	ConfidentialTransferFeeAmount (if mint has TransferFeeConfig)
ConfidentialTransferFee	InitializeConfidentialTransferFeeConfig	Mint	ConfidentialTransferFeeConfig

INSTRUCTION	SUBINSTRUCTION	TARGET	EXTENSION INITIALIZED
ConfidentialMintBurn	InitializeMint	Mint	ConfidentialMintBurn

While many interactions exist, we note a final one implemented in the following function:

```
    pub fn check_for_invalid_mint_extension_combinations(
        mint_extension_types: &[Self],
    ) -> Result<(), TokenError> {
        let mut transfer_fee_config = false;
        let mut confidential_transfer_mint = false;
        let mut confidential_transfer_fee_config = false;
        let mut confidential_mint_burn = false;
        // TRUNCATED...

        for extension_type in mint_extension_types {
            match extension_type {
                ExtensionType::TransferFeeConfig => transfer_fee_config = true,
                ExtensionType::ConfidentialTransferMint => confidential_transfer_mint = true,
                ExtensionType::ConfidentialTransferFeeConfig => {
                    confidential_transfer_fee_config = true
                }
                ExtensionType::ConfidentialMintBurn => confidential_mint_burn = true,
                // TRUNCATED...
            }
        }

        if confidential_transfer_fee_config && !(transfer_fee_config && confidential_transfer_mint)
        {
            return Err(TokenError::InvalidExtensionCombination);
        }

        if transfer_fee_config && confidential_transfer_mint && !confidential_transfer_fee_config {
            return Err(TokenError::InvalidExtensionCombination);
        }

        if confidential_mint_burn && !confidential_transfer_mint {
            return Err(TokenError::InvalidExtensionCombination);
        }

        // TRUNCATED...
```

called by `Processor::_process_initialize_mint()` (`InitializeMint` and `InitializeMint2`), which enforces the following invariants:

- if confidential transfer fees are activated, then confidential transfers must be activated first, as well as base transfer fees
- if transfer fees are enabled (via the base extension) and confidential transfers are enabled as well, then confidential transfer fees must be activated
- if confidential minting is activated, then confidential transfers must be activated first

Interaction With Base Operations

Furthermore, there are a number of other already-existing operations in the base account types that must continue to work as intended with the newly added confidential extensions. We recapitulate how the different extensions affect the already existing operations:

INSTRUCTION	BASE BEHAVIOR	CONFIDENTIAL EXTENSION EFFECT	BLOCKS?
InitializeMint	Sets mint authority, decimals, freeze authority	None - CT mint extension initialized separately via `ConfidentialTransferInstruction::InitializeMint`	No
InitializeMint2	Same as InitializeMint but doesn't require rent sysvar account	None - CT mint extension initialized separately	No

INSTRUCTION	BASE BEHAVIOR	CONFIDENTIAL EXTENSION EFFECT	BLOCKS?
InitializeAccount	Creates token account, sets owner, validates extensions from mint	None - CT account extension must be explicitly initialized via `ConfigureAccount`	No
InitializeAccount2	Same as InitializeAccount but owner is instruction data instead of account	None - CT account extension must be explicitly initialized via `ConfigureAccount`	No
InitializeAccount3	Same as InitializeAccount2 but doesn't require rent sysvar account	None - CT account extension must be explicitly initialized via `ConfigureAccount`	No
InitializeMultisig	Creates multisig account with M-of-N signers	None - multisig is orthogonal to CT	No
InitializeMultisig2	Same as InitializeMultisig but doesn't require rent sysvar account	None - multisig is orthogonal to CT	No
Transfer	Moves tokens from source to destination base balance	**Checks destination's `allow_non_confidential_credits` flag**	**Yes** - if destination has CT extension and flag is false
TransferChecked	Same as Transfer but validates decimals	**Checks destination's `allow_non_confidential_credits` flag**	**Yes** - if destination has CT extension and flag is false
Approve	Sets delegate and delegated_amount on base account	None - delegation only affects base balance, not CT balance	No
ApproveChecked	Same as Approve but validates decimals	None - delegation only affects base balance, not CT balance	No
Revoke	Clears delegate and delegated_amount	None - only affects base delegation	No
SetAuthority	Changes authority for various account/mint operations	Supports `ConfidentialTransferMint` and `ConfidentialTransferFeeConfig` authority types	No
MintTo	Increases destination base balance and mint supply	**If ConfidentialMintBurn extension present, blocks regular minting**	**Yes** - if mint has ConfidentialMintBurn extension
MintToChecked	Same as MintTo but validates	**If ConfidentialMintBurn extension present, blocks regular minting**	**Yes** - if mint has ConfidentialMintBurn

INSTRUCTION	BASE BEHAVIOR	CONFIDENTIAL EXTENSION EFFECT	BLOCKS?
	decimals		extension
Burn	Decreases source base balance and mint supply	None	No
BurnChecked	Same as Burn but validates decimals	None	No
CloseAccount	Transfers lamports to destination, deletes account	**Requires ConfidentialTransferAccount and ConfidentialTransferFeeAmount balances to be zero**	**Yes** - if any CT balances non-zero, must call `EmptyAccount` first
FreezeAccount	Sets account state to Frozen	**Blocks ALL operations** - both base and CT (deposit, withdraw, transfer)	**Yes** (indirectly)
ThawAccount	Sets account state back to Initialized (unfreezes)	Allows operations again - both base and CT	No (enables operations)
SyncNative	Updates wrapped SOL account amount to match lamports	None - works independently of CT	No
GetAccountDataSize	Returns the size needed for an account with specified extensions	None - just calculates size, doesn't check CT	No
InitializeMintCloseAuthority	Sets close authority for mint account	None - CT doesn't affect mint closing	No
InitializeImmutableOwner	Marks account owner as immutable	None - CT works with immutable owners	No
AmountToUiAmount	Converts raw amount to UI amount string	None - operates on base amounts only	No
UiAmountToAmount	Converts UI amount string to raw amount	None - operates on base amounts only	No
Reallocate	Reallocates account to fit new extensions	None - can reallocate to add CT extension	No
CreateNativeMint	Creates the native SOL mint	None - native mint doesn't support CT	No
InitializeNonTransferableMint	Initializes non-transferable mint extension	None - separate extension, but CT transfers would also be blocked	No
InitializePermanentDelegate	Sets permanent delegate for the	None - permanent delegate can burn from base balance only	No

INSTRUCTION	BASE BEHAVIOR	CONFIDENTIAL EXTENSION EFFECT	BLOCKS?
	mint		
WithdrawExcessLamports	Withdraws lamports above rent-exempt minimum	None - only affects lamports, not token balances	No

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	zk-sdk/sigma_proofs/percentage_with_cap	Verifier Should Compute Expected Instance of the Fee Proof By Themselves	Medium
#01	zk-sdk	No Proofs Handle Fiat-Shamir Correctly In Standalone Versions	Medium
#02	token-2022/confidential/elgamal-registry	ElGamal Registry Doesn't Authenticate Owner	Medium
#03	zk-sdk/rangeproof	Fiat-Shamir Issue Leads To Colliding Transcripts In Standalone Range Proof	Medium
#04	zk-sdk/rangeproof	Standalone Range Proof Verifier API Can Crash Under Malicious Inputs	Medium
#05	zk-sdk/sigma_proofs/percentage_with_cap	Unconstrained Fee in Standalone percentage_with_cap Proof	Medium
#06	zk-sdk	Unnecessary Deterministic Randomness in Batch Verification	Low
#07	zk-sdk/sigma_proofs/percentage_with_cap	Uncapped Fees For Generous Provers	Low
#08	token-2022/confidential_transfer_fee	Minor Code Inconsistencies	Informational
#09	zk-sdk	Fiat-Shamir of Public Parameters	Informational
#0b	token-2022	Typed Abstraction Could Go a Long Way	Informational

#00 - Verifier Should Compute Expected Instance of the Fee Proof By Themselves

Severity: Medium **Location:** zk-sdk/sigma_proofs/percentage_with_cap

Description. As part of the verification of the percentage with cap sigma proof, the verifier must compute C_{delta} as

$$10000 \cdot [\text{fee}] - \text{bp} \cdot [\text{tx-amt}]$$

(where $[\cdot]$ indicates a Pedersen commitment to an underlying value)

This is important as the verifier must ensure, at this point, that a "correcting" value has been applied faithfully to the round-up fee calculation.

Yet, the value is directly given to the verifier as `delta_commitment`:

```
    pub fn verify(
        self,
        percentage_commitment: &PedersenCommitment,
        delta_commitment: &PedersenCommitment,
        claimed_commitment: &PedersenCommitment,
        max_value: u64,
        transcript: &mut Transcript,
    ) -> Result<(), PercentageWithCapProofVerificationError> {
```

This interface is easy to misuse, as someone could call this function with an untrusted `delta_commitment`; essentially ignoring a crucial part of the verification logic and making the protocol insecure. This is especially strange as the `percentage_commitment`, which is given as an argument, is used in the computation of the `delta_commitment`.

Recommendation. We recommend moving the computation of `delta_commitment` within this function instead of at the call sites.

#01 - No Proofs Handle Fiat-Shamir Correctly In Standalone Versions

Severity: Medium **Location:** zk-sdk

Description. The Fiat-Shamir transformation of all the proofs (this includes sigma proofs, range proofs, and the inner-product argument (IPA) bulletproofs implementation) is insecure if used as a standalone. While this is not an issue in the overall protocol, if used directly in other contexts the current zk-sdk library could lead to soundness issues.

The problem systematically comes from the fact that some of the public inputs are not absorbed in the transcript in the verifier implementations of all proofs. Importantly, the protocol is currently not vulnerable because all wrapper code (bulletproofs for the IPA, and the Solana zk elgamal proof program for all other proofs) correctly initializes the transcript from the missing public inputs.

In this finding we review how the different proofs are expecting callers to initialize the transcript in the right way.

Sigma proofs. In all sigma proof implementations (`zk-sdk/src/sigma_proofs/`), none of the public inputs are absorbed in the transcript. This is done instead in each of the wrapper code within the zk elgamal proof program (`zk-sdk/src/zk_elgama_proof_program/`) meant to be deployed on the Solana blockchain. One can see that by looking for `new_transcript` functions that systematically initialize the transcript to be used by the verifier using the `context` (public inputs). For example, in the ciphertext-ciphertext equality proof (`zk-sdk/src/zk_elgamal_proof_program/proof_data/ciphertext_ciphertext_equality.rs`):

```
    fn new_transcript(&self) -> Transcript {
        let mut transcript = Transcript::new(b"ciphertext-ciphertext-equality-instruction");

        transcript.append_message(b"first-pubkey", bytes_of(&self.first_pubkey));
        transcript.append_message(b"second-pubkey", bytes_of(&self.second_pubkey));
        transcript.append_message(b"first-ciphertext", bytes_of(&self.first_ciphertext));
        transcript.append_message(b"second-ciphertext", bytes_of(&self.second_ciphertext));

        transcript
    }
```

RangeProofs via Bulletproofs. The API of the rangeproof takes a vector of commitments and bitlengths, producing a proof that each value being committed is of the claimed associated bitlength:

```
impl RangeProof {
    // TRUNCATED...
    pub fn verify(
        &self,
        comms: Vec<&PedersenCommitment>,
        bit_lengths: Vec<usize>,
        transcript: &mut Transcript,
    ) -> Result<(), RangeProofVerificationError> {
```

Yet, the commitments to the values are not absorbed in the transcript, allowing a prover to easily produce a “corrected” commitment (potentially among other fixed invalid commitments as well) that would make an IPA proof verify. One can reproduce the issue by running the following test:

```rust
#[test]
fn zksecurity_fs_omit_v_attack_demonstration() {
    use crate::range_proof::errors::RangeProofVerificationError;

    // false statement: "65535 fits in 2 bits"
    let bogus_amount: u64 = 65535;
    let bits: usize = 2;
    assert!(bogus_amount >= (1u64 << bits)); // see!

    // create proof that doesn't work with the original commitment
    let (original_com, any_opening) = Pedersen::new(bogus_amount);
    let mut t_create = Transcript::new(b"FS_OMIT_V_ATTACK");
    let proof = RangeProof::new(
        vec![bogus_amount],
        vec![bits],
        vec![&any_opening],
        &mut t_create,
    )
    .expect("proof creation for bogus statement should succeed");

    // verify with original commitment (should fail and give us the forged commitment)
    let mut t_verify = Transcript::new(b"FS_OMIT_V_ATTACK");
    let verification_result = proof.verify(vec![&original_com], vec![bits], &mut t_verify);

    let corrected_com = match verification_result {
        Err(RangeProofVerificationError::ZkSecurityForgedV(forged_commitment)) => {
            println!("original_com={:?}", original_com);
            println!("corrected_com={:?}", forged_commitment);
            forged_commitment
        }
        other => panic!("Expected ZkSecurityForgedV error, got {:?}", other),
    };

    // the proof passes with the forged commitment
    let mut t_verify = Transcript::new(b"FS_OMIT_V_ATTACK");
    proof
        .verify(vec![&corrected_com], vec![bits], &mut t_verify)
        .expect("attack should make verification pass");
}
```

after applying the following diff:

```
diff --git a/zk-sdk/src/range_proof/errors.rs b/zk-sdk/src/range_proof/errors.rs
index f3c304f..5734336 100644
--- a/zk-sdk/src/range_proof/errors.rs
+++ b/zk-sdk/src/range_proof/errors.rs
@@ -1,6 +1,9 @@
 //! Errors related to proving and verifying range proofs.
 use {crate::errors::TranscriptError, thiserror::Error};

+use crate::encryption::pedersen::PedersenCommitment;
+
 #[cfg(not(target_os = "solana"))]
 #[derive(Error, Clone, Debug, Eq, PartialEq)]
 pub enum RangeProofGenerationError {
@@ -36,6 +39,9 @@ pub enum RangeProofVerificationError {
     MaximumGeneratorLengthExceeded,
     #[error("commitments and bit lengths vectors have different lengths")]
     VectorLengthMismatch,
+    #[error("...")]
+    ZkSecurityForgedV(PedersenCommitment),
 }

 #[cfg(not(target_os = "solana"))]
diff --git a/zk-sdk/src/range_proof/mod.rs b/zk-sdk/src/range_proof/mod.rs
index d25f0b2..d811b1a 100644
--- a/zk-sdk/src/range_proof/mod.rs
+++ b/zk-sdk/src/range_proof/mod.rs
@@ -392,11 +392,12 @@ impl RangeProof {

         let basepoint_scalar =
             w * (self.t_x - a * b) + d * (delta(&bit_lengths, &y, &z) - self.t_x);
-        let value_commitment_scalars = util::exp_iter(z).take(m).map(|z_exp| d * zz * z_exp);

         // 4. Perform the final "mega-check"
         // This single multiscalar multiplication verifies all relations simultaneously.
-        let mega_check = RistrettoPoint::optional_multiscalar_mul(
+
+        // Compute the part of mega-check excluding the value commitments
+        let mega_check_without_comms = RistrettoPoint::optional_multiscalar_mul(
             iter::once(Scalar::ONE)
                 .chain(iter::once(x))
                 .chain(iter::once(d * x))
@@ -406,8 +407,7 @@ impl RangeProof {
                 .chain(x_sq.iter().cloned())
                 .chain(x_inv_sq.iter().cloned())
                 .chain(gs)
-                .chain(hs)
-                .chain(value_commitment_scalars),
+                .chain(hs),
             iter::once(self.A.decompress())
                 .chain(iter::once(self.S.decompress()))
                 .chain(iter::once(self.T_1.decompress()))
@@ -417,14 +417,37 @@ impl RangeProof {
                 .chain(self.ipp_proof.L_vec.iter().map(|L| L.decompress()))
                 .chain(self.ipp_proof.R_vec.iter().map(|R| R.decompress()))
                 .chain(bp_gens.G(nm).map(|&x| Some(x)))
-                .chain(bp_gens.H(nm).map(|&x| Some(x)))
-                .chain(comms.iter().map(|V| Some(*V.get_point()))),
+                .chain(bp_gens.H(nm).map(|&x| Some(x))),
         )
         .ok_or(RangeProofVerificationError::MultiscalarMul)?;

+        // Add the commitment part
+        let value_commitment_scalars: Vec<Scalar> = util::exp_iter(z)
+            .take(m)
+            .map(|z_exp| d * zz * z_exp)
+            .collect();
+        let commitment_part = RistrettoPoint::optional_multiscalar_mul(
+            value_commitment_scalars.iter().cloned(),
+            comms.iter().map(|V| Some(*V.get_point())),
         )
         .ok_or(RangeProofVerificationError::MultiscalarMul)?;
```

```
+        let mega_check = mega_check_without_comms + commitment_part;
+
         if mega_check.is_identity() {
             Ok(())
         } else {
+            // For single commitment case, compute the forged commitment that would make verification pass
+            if comms.len() == 1 {
+                let coeff = value_commitment_scalars[0];
+                if coeff != Scalar::ZERO {
+                    let corrected_point = -(coeff.invert()) * mega_check_without_comms;
+                    let forged_commitment = PedersenCommitment::new(corrected_point);
+                    return Err(RangeProofVerificationError::ZkSecurityForgedV(
+                        forged_commitment,
+                    ));
+                }
+            }
             Err(RangeProofVerificationError::AlgebraicRelation)
         }
     }
```

Inner-Product Argument (IPA) via Bulletproofs. In the range proof context in which the IPA is used, the commitment P to $l(x)$ and $r(x)$ (or a, b in the IPA notation) is **implicit** as it is derived from transcript values A, S, and challenges $\tilde{e}, z, \mathbf{y}$ (see Dalek docs on Proving that $l(x), r(x)$ are correct). Indeed, the prover doesn't send additional messages to the verifier in that protocol for the verifier to be able to go through the IPA.

But when used as a standalone, the IPA protocol accepts an **explicit** commitment to P which then needs to be absorbed in the `transcript` (but is not):

```rust
impl InnerProductProof {
    // TRUNCATED...
    pub fn verify<IG, IH>(
        &self,
        n: usize,
        G_factors: IG,
        H_factors: IH,
        P: &RistrettoPoint, // <-- right here
        Q: &RistrettoPoint,
        G: &[RistrettoPoint],
        H: &[RistrettoPoint],
        transcript: &mut Transcript,
    ) -> Result<(), RangeProofVerificationError>
```

Furthermore, the API allows passing the parameter points G and H (which we reference as G_{IPA} and H_{IPA} in the overview) dynamically, which could mean that these parameters would have to be absorbed in the transcript as well if they could be changed as part of the protocol.

Note that the `range_proof` module is not public in the crate, which makes this less severe, although adding documentation warning developers who would want to extract the code to be used as a standalone in different contexts would be welcome.

#02 - ElGamal Registry Doesn't Authenticate Owner

Severity: Medium **Location:** token-2022/confidential/elgamal-registry

Description. An ElGamal Registry program is implemented in order to facilitate the creation of confidential token accounts for other peers. As the creation of token accounts require pubkey validity proofs (see the section on Public key validity), and each new (confidential) mint requires a new (confidential) token account, the registry can be used to publish these to everyone.

The registry also lets you post an associated owner, which is the public key of a Solana system account, also required to administrate a (confidential) token account.

The issue is that there is no authentication mechanism that enforces that the link between the (validated) ElGamal public key and the owner is legitimate.

Recommendation. Adding the (initial) owner of the to-be-created token account to the transcript would naturally authenticate the intent of the proof, narrowing the portability of this proof.

#03 - Fiat-Shamir Issue Leads To Colliding Transcripts In Standalone Range Proof

Severity: Medium **Location:** zk-sdk/rangeproof

Description. The interfaces of the rangeproof implementation, on both the prover and the verifier sides, take a vector of (committed) values and their claimed bitlengths. You can see that in `zk-sdk/src/range_proof/mod.rs`:

```
impl RangeProof {
    pub fn new(
        amounts: Vec<u64>,
        bit_lengths: Vec<usize>,
        openings: Vec<&PedersenOpening>,
        transcript: &mut Transcript,
    ) -> Result<Self, RangeProofGenerationError> {
        // TRUNCATED...
        let nm: usize = bit_lengths.iter().sum();
        // TRUNCATED...
        transcript.range_proof_domain_separator(nm as u64);
        // TRUNCATED...
    }

    pub fn verify(
        &self,
        comms: Vec<&PedersenCommitment>,
        bit_lengths: Vec<usize>,
        transcript: &mut Transcript,
    ) -> Result<(), RangeProofVerificationError> {
        // TRUNCATED...
        let nm: usize = bit_lengths.iter().sum();
        // TRUNCATED...
        transcript.range_proof_domain_separator(nm as u64);
        // TRUNCATED...
    }
}
```

As one can see in the previous code, the Fiat-Shamir implementation absorbs the sum of all bitlengths `nm` instead of each bitlength individually. This ambiguity means that different transcripts might collide and produce the same challenges for different instances. For example, range checks on bit ranges `[32, 32]` and `[31, 33]` will produce the same challenge at some point in the transcript, in spite of validating two different instances.

While this is hard to exploit in practice due to the lack of freedom for the attacker (they can only alter bitlengths without being able to significantly correct any error term that would result in the proof), we would strongly recommend absorbing a canonical schema that would unambiguously describe the instance.

Note that as pointed out in No Proofs Handle Fiat-Shamir Correctly In Standalone Versions, this is not an issue when the implementation of the rangeproof is not used as a standalone. This is because the wrapper code in the Solana program making use of the rangeproof code is correctly absorbing the individual bitlengths as can be seen in `zk-sdk/src/zk_elgamal_proof_program/proof_data/batched_range_proof/mod.rs`:

```
pub struct BatchedRangeProofContext {
    pub commitments: [PodPedersenCommitment; MAX_COMMITMENTS],
    pub bit_lengths: [u8; MAX_COMMITMENTS],
}

#[allow(non_snake_case)]
#[cfg(not(target_os = "solana"))]
impl BatchedRangeProofContext {
    fn new_transcript(&self) -> Transcript {
        let mut transcript = Transcript::new(b"batched-range-proof-instruction");
        transcript.append_message(b"commitments", bytes_of(&self.commitments));
        transcript.append_message(b"bit-lengths", bytes_of(&self.bit_lengths));
        transcript
    }
```

#04 - Standalone Range Proof Verifier API Can Crash Under Malicious Inputs

Severity: Medium **Location:** zk-sdk/rangeproof

Description. In the range proof API, if the verifier is given a proof with an `R_vec` field of an unexpected length, the verifier will panic. This is in contrast with the highly defensive style of the codebase, which attempts to avoid panics at all costs. This only seems to be an issue if used as a standalone, as the size of the proof (and its elements) is fixed when used as part of a Solana program.

The issue arises from an invalid array access on `challenges` after a short-circuited `zip` loop (due to `zip` stopping as soon as one of its operands is exhausted):

```
    pub(crate) fn verification_scalars(
        &self,
        n: usize,
        transcript: &mut Transcript,
    ) -> Result<(Vec<Scalar>, Vec<Scalar>, Vec<Scalar>), RangeProofVerificationError> {
        let lg_n = self.L_vec.len();
        if lg_n == 0 || lg_n >= 32 {
            // 4 billion multiplications should be enough for anyone
            // and this check prevents overflow in 1<<lg_n below.
            return Err(RangeProofVerificationError::InvalidBitSize);
        }
        if n != (1_usize.checked_shl(lg_n as u32).unwrap()) {
            return Err(RangeProofVerificationError::InvalidBitSize);
        }

        transcript.inner_product_proof_domain_separator(n as u64);

        // 1. Recompute challenges `u_i` from the proof transcript (`x_i` in the paper).
        let mut challenges = Vec::with_capacity(lg_n);
        for (L, R) in self.L_vec.iter().zip(self.R_vec.iter()) {
            transcript.validate_and_append_point(b"L", L)?;
            transcript.validate_and_append_point(b"R", R)?;
            challenges.push(transcript.challenge_scalar(b"u"));
        }

        // 2. Compute `u_i^-1` for all `i`.
        let mut challenges_inv = challenges.clone();
        // This computes `(u_k * ... * u_1)^-1` and stores `u_i^-1` in `challenges_inv`.
        let allinv = Scalar::batch_invert(&mut challenges_inv);

        // 3. Compute `u_i^2` and `u_i^-2` for all `i`.
        for i in 0..lg_n {
            challenges[i] = challenges[i] * challenges[i];
            challenges_inv[i] = challenges_inv[i] * challenges_inv[i];
        }
```

Recommendation. Enforce that both `L_vec` and `R_vec` are of the expected lengths, and return an error otherwise.

#05 - Unconstrained Fee in Standalone percentage_with_cap Proof

Severity: Medium **Location:** zk-sdk/sigma_proofs/percentage_with_cap

Description. The standalone percentage-with-cap proof system allows transfer fees to be arbitrarily large when the fee calculation `(amount * basis_points + delta) / 10000` does not result in an integer. By setting `delta` to a non-zero value that makes the division a non-integer in the field, the computed fee can wrap around the field modulus, resulting in fees that exceed the transaction amount.

The percentage-with-cap proof verifies the relationship:

```
(amount * basis_points + delta) = fee * 10000
```

Delta is a value added to make the left-hand side a multiple of 10,000, yet it can be arbitrarily set by the prover and thus does not necessarily have to be a small rounding error.

The fee equation is checked over the prime order group of Ristretto (of size $2^{252} - 27742317777372353535851937790883648493$). While the proof includes range proofs on `delta` and `claimed_fee`, there is no range proof on the actual `fee` commitment used in the transfer.

We use SageMath to demonstrate the issue over the Ristretto scalar field:

```
p = 7237005577332262213973186563042994240857116359379907606001950938285454250989
F = GF(p)

tx_amt = 10^9 # < 64-bit
bp = 200
fee = F(bp * tx_amt + 1) / F(10000) # using x=1

print(fee)
# 6585675075372358614715599772369124759179975887035715921461775353841763368 4
print(len(bin(fee)) - 2)
# 246
```

Note that it is unlikely that this issue leads to further problems within the confidential transfer protocol that makes use of it, due to range proofs that should later arise when debiting or crediting accounts with these amounts.

Recommendation. Add a range proof on the actual transfer fee commitment to ensure it stays within reasonable bounds (e.g., proving fee $< 2^{64}$). This should enforce that `(amount * basis_points + delta)` must be divisible by 10000 in the integers, not just in the field.

In addition, write a proof that the following formula holds in the integers, when both sides are upper-bounded by a value that is smaller than the size of the field:

```
(amount * basis_points + delta) = fee * 10000
```

Client Response. The client acknowledged this issue exists in the standalone percentage_with_cap proof module. However, when used in the transfer-with-fee instruction context (its primary use case), additional range proofs are always included that certify that, amongst others, the fee amount is a positive number within some range. The client agrees that documentation should be improved to clarify this requirement for standalone usage of the module.

#06 - Unnecessary Deterministic Randomness in Batch Verification

Severity: Low **Location:** zk-sdk

Description. Batch verification of checks can often be done more efficiently by aggregating all checks into a single check via a random linear combination. We noticed this being done in two places in ways that can be (and have been) damaging to the protocol.

The verification of the percentage-with-cap sigma proof (in `zk-sdk/src/sigma_proofs/percentage_with_cap.rs`) uses a Fiat-Shamir method to deterministically generate the batch verification randomness w and w^2 for aggregating multiple MSM checks into a single verification equation (each check being assigned a different power of w).

Specifically, the verifier derives w deterministically via `transcript.challenge_scalar(b"w")` after manually absorbing all proof components. Using the Schwartz-Zippel lemma, it is relatively straightforward to show that the batch verification is secure as long as w is truly unpredictable to malicious provers. While the current implementation appears to absorb all necessary instance and proof elements, this introduces unnecessary complexity and potential for future bugs.

The same is being done in the range proof where a `d` challenge is sampled (deterministically, in a Fiat-Shamir way) in order to batch the two bit constraints with the main IPA constraint.

Deterministic verification enables reproducible results that, when well-studied, can avoid bugs. That being said, this technique is most often used on the other side, with algorithms like deterministic ECDSA, EdDSA, or SIV that have the signer or encrypter deterministically generate some of the algorithm values (most often nonces). This is rarely done as an optimization but rather to defend against nonce-reuse attacks.

Note that this has been an issue in the past, as previously reported by zkSecurity in the percentage-with-cap proof and also fixed in the range proof.

Note also that the logic we're looking at here is meant to be ran in the context of the Solana VM (SVM), where hosts might not have access to cryptographically secure sources of random numbers. As far as we know, no native function in the SVM currently needs to read random numbers from their host system.

This is clearly a trade-off, and we would like to point out that for minimal overhead one could choose the trade-off of security over speed by just checking the three checks independently, without randomness (deterministic or not).

#07 - Uncapped Fees For Generous Provers

Severity: Low **Location:** zk-sdk/sigma_proofs/percentage_with_cap

Description. In the Percentage With Cap Argument paper supporting the implementation with proofs of the implemented OR sigma protocol, the second *OR branch* is meant to be taken only if the max fee has been reached (emphasis ours):

> *Case 2: Fee greater than maxfee. Consider the case when the fee is equal to maxfee. In this case, the fee commitment Commfee* ***must be*** *a commitment to maxfee [...]*

Yet, a prover that is generous with their money can still choose to use the first branch in order to pay fees larger than the hardcoded cap.

While this does not seem like an important issue, because these edge cases tend to only impact malicious users (as they will simply choose to pay more fees if they do not follow the protocol to the letter), we decided to point it out as it is still dictated in the specifications that this should not happen.

Recommendation. Either publicly document this edge case, or extend the protocol in order to prevent it.

#08 - Minor Code Inconsistencies

Severity: Informational **Location:** token-2022/confidential_transfer_fee

Description. In this finding we point out two minor inconsistencies.

Missing ownership checks. A program's logic will often check that a storage account passed to one of its instructions is rightly owned by the program. This is necessary in a number of cases as otherwise the program would read from untrusted storage. This "ownership check" is present across all relevant instructions of the confidential transfer extensions, except for two of them: `InitializeConfidentialTransferFeeConfig` and `HarvestWithheldTokensToMint`.

This check seems unnecessary in a number of cases, including the ones we pointed out, as the instruction ends up mutating the storage account anyway, which will return an error if the program does not own the account in the first place. Nevertheless, this pattern is used throughout the extension's logic, and it might be a good idea to ensure that it is consistent in these places too. For example, in the `UpdateMint` function:

```
fn process_update_mint(
    accounts: &[AccountInfo],
    auto_approve_new_account: PodBool,
    auditor_encryption_pubkey: &OptionalNonZeroElGamalPubkey,
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let mint_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;

    check_program_account(mint_info.owner)?; // <-- the check
```

Useless Pending Counter Check. Another minor inconsistency was found in the `valid_as_destination()` method which, amongst its many checks, enforces that a token account's pending counter has not been maxed out. While necessary in many cases, the method is called in both the `WithdrawWithheldTokensFromMint` and the `WithdrawWithheldTokensFromAccounts` subinstructions, which update the main balance directly, and thus do not need to abort if the maximum counter value has been reached.

```
pub fn valid_as_destination(&self) -> ProgramResult {
    // TRUNCATED...
    let new_destination_pending_balance_credit_counter =
        u64::from(self.pending_balance_credit_counter)
            .checked_add(1)
            .ok_or(TokenError::Overflow)?;
    if new_destination_pending_balance_credit_counter
        > u64::from(self.maximum_pending_balance_credit_counter)
    {
        return Err(TokenError::MaximumPendingBalanceCreditCounterExceeded.into());
    }

    Ok(())
}
```

#09 - Fiat-Shamir of Public Parameters

Severity: Informational **Location:** zk-sdk

Description. In addition to the discussion on absorbing public inputs, we note that the programs do not absorb public parameters (*e.g.*, curve parameters, generators, etc...) as part of the implementation of the Fiat-Shamir transform. Consequently, all proof implementations are unsound in the presence of an adversary that is allowed to choose and manipulate the public parameters (they do not have *adaptive* soundness).

In the current state of the system, these attacks are not a threat. Indeed, the canonical verifier is an on-chain program in which the public parameters are hard-coded; this verifier will never verify or accept proofs with respect to a set of maliciously-chosen parameters. However, any extension to the system or third-party applications will also need to hard-code these parameters.

Recommendation. Following the principle of defense-in-depth, we recommend to absorb public parameters (or a public digest thereof) in the Fiat-Shamir transcripts. Alternatively, documentation should be extremely clear that hard-coding parameters is a critical security feature.

#0b - Typed Abstraction Could Go a Long Way

Severity: Informational **Location:** token-2022

Description. The Token-2022 confidential transfer extensions include a lot of code duplication, which makes auditing and maintaining the code harder and could also lead to developer mistakes. For example, (sub)instructions often have to:

- perform the same checks (e.g., ownership checks, as explained in Minor Code Inconsistencies) when given storage accounts (like a mint account or a token account)
- enforce that specific mint operations are currently allowed depending on base mint values, base extensions (e.g., `PausableConfig`), or confidential extensions (e.g., if `allow_non_confidential_credits` is set to `true`)
- enforce similar checks on operated token accounts (e.g., check if the account can transfer according to its `NonTransferableAccount` extension)
- enforce that used storage accounts are uninitialized or initialized
- enforce a general expected schema on the different accounts passed by the instructions (e.g., some accounts are owned, some are signers, some contain some expected data, etc.)

Having a typed abstraction that would allow succinctly encoding what accounts are expected by the subinstruction, how the subinstruction relates to other extensions, and how it relates to the overall state machine (initialization vs post-initialization) would go a long way.

That being said, the code is currently thorough, well-commented, and solid (as the report can attest with its lack of serious findings), which creates a good argument for avoiding big refactors. However, if more extensions are planned to be added to the Token-2022 program, unforeseen interactions between new extensions and the current confidential extensions could still generate issues due to the current verbosity of the implementation.