ZKSECURITY

Audit of Silent Protocol Circuits

October 13th, 2023

Introduction

In the two weeks from September 22nd to October 6th 2023, zkSecurity performed a security audit of Silent Protocol's Circom circuits.

A number of observations and findings have been reported to the Silent Protocol team. The findings are detailed in the latter section of this report.

Silent Protocol's circuits were found to be of high quality, accompanied with thorough documentation, specifications and tests. **As of writing, all high and medium-severity issues we found were patched by the Silent Protocol team, and zkSecurity confirms that these patches properly address our findings.**

Note that security audits are a valuable tool for identifying and mitigating security risks, but they are not a guarantee of perfect security. Security is a continuous process, and organizations should always be working to improve their security posture.

Scope

A consultant from zkSecurity spent two weeks auditing the Circom circuits for the Silent multi-asset shielded pool (SMASP) application. These circuits represent the privacy-preserving portion of the overall SMASP application. They are used in a set of Solidity smart contracts that verify zero-knowledge proofs created from these circuits.

Smart contracts were reviewed by zkSecurity at an earlier date, and were only in scope insofar as they perform preparation and validation of public circuit inputs. The main focus of this audit were the application's circuits written in Circom. These include circuits for the main SMASP protocol, circuits used by the compliance logic, and circuits representing future functionality not used in the contracts at the time of review. An overview of audited circuits can be found below.

The audit also covers all auxiliary circuit templates and utilities maintained by Silent Protocol itself, including (but not limited to) templates for foreign-field arithmetic and ElGamal encryption. Not covered by the audit are templates imported from circomlib, Circom's standard library; and functionality contained in snarkjs, such as the Groth16 prover.

Circuits overview

An overview of the overall SMASP protocol can be found in zkSecurity's first report for Silent Protocol. Here, we focus on introducing recurring concepts that provide context for understanding our findings below.

Encrypted anonymized asset transfers

To understand the flow of assets through Silent's shielded pool, we focus on one example first: the deposit circuit.

Deposit. A public Ethereum account deposits assets into a shielded pool account.

- The deposited amount is added to a shielded balance, without revealing that balance.
- The sender's shielded account is anonymized by making indistinguishable dummy updates to 7 other accounts.

The privacy properties of this method are achieved by combining a zero-knowledge proof with ElGamal encryption of balances.

The ElGamal ciphertext representing a balance is defined as

$$(\mathrm{C_1, C_2}) = (\mathrm{rG, bG + rX})$$

Here, $(\mathrm{C_1, C_2})$ is the ciphertext, r is encryption randomness, G is a public elliptic curve base point, b is the balance, and X is the public key of the account owner.

The deposit circuit performs two operations on ciphertexts:

- The sender account is updated by adding the deposited amount a to the balance, exploiting additive homomorphism:

$$(\mathrm{C_1, C_2}) \leftarrow (\mathrm{C_1, C_2 + aG})$$

- All 8 balance ciphertexts (on the sender's and 7 decoy accounts) are updated with new randomness $\mathrm{r'}$:

$$(\mathrm{C_{1i}, C_{2i}}) \leftarrow (\mathrm{C_{1i} + r'G, C_{2i} + r'X_i})$$

The re-randomization step is what makes the sender's balance update indistinguishable from updates to the 7 decoy accounts, which leave their balance in place. Note that we can perform both the re-randomization and the balance update without knowing the accounts owner's private keys, and we also preserve their ability to decrypt their balances. Therefore, decoys can be real, active accounts by other users, which is necessary for this scheme to provide actual anonymity for the sender. We have to ensure that the sender knows their own private key, by proving that we can rederive their public key $\mathrm{xG = X}$.

The ciphertexts before and after applying the deposit update are public inputs to the circuit, while the new randomness is a private input. The deposit amount is public as well, because the contract needs to equate it to the amount received from the sender's Ethereum account. The circuit asserts correct execution of the ciphertext updates given above.

A second part of the circuit computes a `senderHash`, defined as

$$\mathrm{senderHash} = \mathrm{H(r''G + xX_{aml})}.$$

Here, $\mathrm{H()}$ is the Poseidon hash function, $\mathrm{r''}$ is randomness, x is the sender's private key and $\mathrm{X_{aml}}$ is the public key shared by the compliance committee. The `senderHash` and $\mathrm{r''G}$ are made available as public inputs. Note that the key $\mathrm{xX_{aml}}$ is shared between the sender and compliance committee. Both the sender and the compliance committee can recompute the `senderHash` to check whether this transaction belongs to the sender.

SMASP circuits

Other circuits to interact with the shielded pool make use of the same concepts outlined above for the deposit circuit, with slight variations:

`DepositAndWithdraw`. The withdraw logic shares its circuit with deposits, but uses a negative amount. The only difference, which is handled in-circuit, is that for withdrawals we also check that the amount is smaller than the balance. In order to do this, the circuit takes the current balance as private input and verifies that it is encrypted correctly.

`Transfer` differs from deposits in that the sent amount is private, and that two accounts are anonymously updated: the sender and recipient. The circuit likewise encrypts transaction details in two versions: One shared between sender and recipient, and one shared with the compliance committee. Both versions derive a shared secret using ECDH and use it as the key in MiMC-based encryption.

`Register` is the circuit that creates new shielded accounts. It generates new ElGamal ciphertexts of zero balances for four different assets.

`FeeRegistration` lets a user subscribe for zero-fee withdrawals. It uses the same technique as deposits to encrypt the end of the subscription period.

`WithdrawFeeReduction` is the method unlocked for a user after calling `FeeRegistration`. It is similar to withdraw, except that it also verifies the validity of the encrypted subscription period against the current block number, which is a public input.

`TransferToNonSilent`, `ClaimAndRegister` and `ClaimBatchPoints` are not used by smart contracts in the audited version of the protocol. They use the same techniques as `Transfer` and `Register` to verify encrypted balances and compute encrypted transaction details.

All of these circuits -- with the exception of `Register`, `ClaimAndRegister` and `ClaimBatchPoints` -- hide the sender in a size-8 anonymity set, using the same re-encryption technique as deposits.

Generally speaking, we found the implementation of encrypted and anonymized asset transfers to be solid, with consistent usage of the same patterns and well-documented core templates, like `BalanceVerify()` and `BalanceUpdate()`. Only one major issue was found in this part of the code, which stems from a non-standard application of ElGamal encryption in the `FeeRegistration` logic, breaking the sender's anonymity; see finding #00.

Secret sharing for compliance

After joining the compliance committee, every member will create a secret that they share with all other members. Likewise, the other members send a secret share to them. The scheme uses a variant of Shamir secret sharing that is suitable for threshold decryption, by avoiding the need for a single dealer; it also ensures that secret shares are verifiable against public commitments to the secret generated by each member. See AHS20 for an overview of the scheme.

The `SecretSharing` template represents the scheme in circuit form. The template is used by a committee member when they share secrets with other members, by posting them in encrypted form to a compliance smart contract. Along with encrypted shares, commitments to the underlying secret are also posted publicly; this enables members to verify their shares. The circuit's purpose is to prove that encrypted shares and commitments are computed correctly and from the same polynomial.

In mathematical terms, the secret is an element of a finite field, $s \in \mathbb{F}_p$. The sharing entity constructs a polynomial $p(X)$ of degree $T - 1$ which evaluates to the secret at 0:

$$p(X) = \sum_{j=0}^{T-1} p_j X^j, \quad \text{where } s = p_0 = p(0).$$

Polynomial coefficients are passed as private inputs, along with S public evaluation points $x_i, i = 0, \ldots, S-1$. The `` `SecretSharing` `` template evaluates the polynomial at each x_i to obtain the ith secret share, $ss_i = p(x_i)$. Note that $S \geq T$ (and uniqueness of the x_i) ensures that T or more shares can be combined to reconstruct the secret. In practice, members will only combine shares "in the exponent" so as to not reveal them, to collectively compute a curve point sC for ElGamal decryption.

Besides evaluating the polynomial, the template also needs to compute commitments to the polynomial coefficients, which are defined as

$$A_j = p_j G.$$

The A_j are broadcast by storing them on the smart contract. To validate the share they received, each member can check that

$$ss_i G = p(x_i) G = \sum_{j=0}^{T-1} x_i^j A_j.$$

To do scalar multiplications $p_j G$ efficiently in the circuit, the chosen curve is BabyJubJub, whose base field is the native circuit field. Note, however, that the polynomial lives in the *scalar field* of that curve. This means we have to perform polynomial operations in non-native arithmetic modulo the curve order p; with coefficients, evaluation points and secret shares all represented as bigints.

Non-native arithmetic is a major source of complexity in the `` `SecretSharing` `` template. Indeed, out of the 6 high-to-medium findings reported, 5 are related to `` `SecretSharing` `` and non-native arithmetic (see findings #01 through #05).

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	feeRegistration.circom	Broken anonymity in zero-fee registration	High
01	secretSharing.circom	Unconstrained secret share encryption	High
02	secretSharing.circom	Missing range checks on all input bigint limbs	High
03	secretSharing.circom	Underconstrained native arithmetic on scalars, instance 1	Medium
04	secretSharing.circom	Underconstrained native arithmetic on scalars, instance 2	Medium
05	polynomial.circom	Secret shares computed ambiguously by not reducing modulo p	Medium
06	depositAndWithdraw.circom	Unconstrained booleans leave amount range check unenforced	Low
07	crypto/src/*.ts	Insecure randomness used for anonymity set sender position	Low
08	bigint_func.circom	log_ceil() produces off-by-one error	Informational

# 00 - Broken anonymity in zero-fee registration

feeRegistration.circom

High

Description. The `feeRegistration` circuit is used when a user registers for zero-fee withdrawals. They pay a subscription fee and, in return, are able to use the complementary `WithdrawFeeReduction` method for a period of time. The block number at which the subscription ends is a public input of the `feeRegistration` circuit, called `validUntil`.

Like other Silent Protocol methods, the interacting user (sender) is kept hidden in an anonymity set of 8 accounts. The sender's index within the array of size 8 is a private input. No-op account updates are performed on all 8 accounts, such that the actual update to the sender's account -- setting a new `validUntil` value -- can't be distinguished from the outside.

However, contrary to the other Silent Protocol methods reviewed by zkSecurity, `feeRegistration` falls short of protecting the sender's anonymity.

For context, each user account stores ElGamal ciphertexts (C_1, C_2) which represent $v :=$`validUntil` via

$$(C_1, C_2) = (rG, vG + rX)$$

where r is encryption randomness and X is the users public key. The circuit takes two versions of each of the 8 ciphertexts as public inputs: before and after applying the update.

The circuit takes new encryption randomness r' as private input. The following operations are then performed to update ciphertexts:

- For the sender account, the new v is entirely re-encrypted using r': $(C_1, C_2) \leftarrow (r'G, vG + r'X)$
- All 8 accounts are rerandomized, using the same r': $(C_1, C_2) \leftarrow (C_1 + r'G, C_2 + r'X)$

By contrast, in balance updates we don't encrypt the sender value from scratch, but just add an amount aG to C_2 exploiting the additive homomorphism of ElGamal encryption.

The problem with the outlined method of resetting v is that it's easy to distinguish the sender update from the others, just based on how the ciphertexts change. Note that for all other accounts, C_1 changes by adding $r'G$. This is not the case for the sender account, which is *reset* to the new value $2r'G$. So, to find out the secret index in $0, \ldots, 7$ which belongs to the sender, just look at the difference ΔC_1 for each account and note that one is different from all the others. Then, look up which public key belongs to that index, and you have successfully deanonymized the sender.

Impact. Deanonymizing the sender breaks the core promise of the protocol, which is to hide users within an anonymity set.

Recommendation. We can mimic how balance updates work. The recommendation is to introduce the previous value v_{old} as a private circuit input, so that the sender update can just add to C_2:

- For the sender account, update: $C_2 \leftarrow C_2 + (v - v_{\text{old}})G$
- Rerandomize all 8 accounts: $(C_1, C_2) \leftarrow (C_1 + r'G, C_2 + r'X)$

For the v update, the `BalanceUpdate` template can be reused:

```
component updateSender = BalanceUpdate();
updateSender.amount <== validUntil + SUBORDER - oldValidUntil;
```

Since `validUntil` is checked to be a smallish value by the smart contract logic that validates public inputs, we recommend to pass $\mathrm{v} + \mathrm{p} - \mathrm{v}_{\mathrm{old}}$ as input to `BalanceUpdate`, where p is the scalar field size. This expression will never overflow the native field since p is much smaller than the native modulus. By adding p, we ensure that passing $\mathrm{v} < \mathrm{v}_{\mathrm{old}}$ doesn't cause an underflow, which would lead to a very long subscription period.

Client response. Silent Protocol fixed the issue, following our recommendation.

# 01 - Unconstrained secret share encryption

secretSharing.circom

High

Description. To distribute secret shares publicly, they are encrypted using hashed ElGamal encryption:

$$(\mathrm{C}_1, \mathrm{c}_2) = (\mathrm{rG}, \mathrm{ss} + \mathrm{H}(\mathrm{rX}))$$

where ss is the secret share, X is the public key of its recipient and r is encryption randomness.

In the protocol, c_2 and ss are BabyJubJub scalars, not native field elements. The **`SecretSharing`** template represents them as bigints and computes them in non-native arithmetic.

To compute $\mathrm{H}(\mathrm{rX})$, the protocol uses Poseidon over the *native* field. Since the native field is bigger than the scalar field, the hash output is reduced modulo the subgroup order to become a valid scalar.

The template uses the following code:

```
secretHasher[i] = Poseidon(2);
secretHasher[i].inputs[0] <== rPubKeys[i].out[0];
secretHasher[i].inputs[1] <== rPubKeys[i].out[1];

var SUBORDER = 2736030358979909402780800718157159386076813972158567259200215660948447373041;
hashed[i] <-- secretHasher[i].out % SUBORDER;
sum === hashed[i];
```

Here, **`sum`** contains $\mathrm{c}_2 - \mathrm{ss}$, stored in a single field element. **`hashed[i]`** is supposed to represent $\mathrm{H}(\mathrm{rX})$ reduced modulo **`SUBORDER`**. In the last line, the code asserts the equality $\mathrm{c}_2 - \mathrm{ss} = \mathrm{H}(\mathrm{rX})$ to show that c_2 is the correct encrypted secret share.

The problem is that the modulo reduction **`secretHasher[i].out % SUBORDER`** is computed outside the circuit, and stored in **`hashed[i]`** with the witness assignment operator **`<--`**. This means that **`hashed[i]`** is not provably linked to the output of **`secretHasher`** in any way. From the circuit point of view, it is completely unconstrained and its value can be set arbitrarily by the prover.

At zkSecurity, we have called this common mistake a **boomerang value**: A value that leaves the circuit and gets reinserted later, without being linked to the original computation.

Impact. A malicious user can publish any values they want as encrypted secret shares and have them accepted by the smart contract. This severely breaks the compliance component of the protocol.

Client response. Silent Protocol patched the issue by redefining the protocol, such that hashed ElGamal encryption now happens in the native field. The secret share ss is still a bigint scalar, but it can naturally be embedded in the native field because the scalar field is smaller. The check $\mathrm{c}_2 - \mathrm{ss} = \mathrm{H}(\mathrm{rX})$ is now a simple constraint between two native field elements and we no longer need reduction modulo the subgroup order.

02 - Missing range checks on all input bigint limbs

secretSharing.circom

High

Description. Secret shares are computed by evaluating a polynomial $\mathrm{p}(\mathrm{X})$ on a set of points $(\mathrm{x_i})$:

$$\mathrm{ss_i} = \mathrm{p}(\mathrm{x_i}) = \sum_{\mathrm{j}=0}^{\mathrm{T}-1} \mathrm{p_j} \mathrm{x_i^j} \qquad \text{for } \mathrm{i} = 0, \ldots, \mathrm{S} - 1$$

This evaluation happens in non-native arithmetic, so the polynomial coefficients $(\mathrm{p_j})$ and evaluation points $(\mathrm{x_i})$ are passed in as bigints: fixed-size arrays of signals which represent the limbs of a bigint. Concretely, 4 limbs of 64 bits each are used, so a value like $\mathrm{x_i}$ is represented as $(\mathrm{x_{i0}}, \mathrm{x_{i1}}, \mathrm{x_{i2}}, \mathrm{x_{i3}})$ where $\mathrm{x_i} = \sum_{\mathrm{k}=0}^{3} \mathrm{x_{ik}} 2^{64\mathrm{k}}$.

The polynomial evaluation circuit is built on top of non-native arithmetic gadgets like `FpAdd` and `FpMultiply` originally taken from circom-pairing, which themselves rely on bigint implementations taken from circom-ecdsa.

These non-native gadgets constrain their outputs to be valid bigints with limbs in $[0, 2^{64})$. However, none of them constrain their *inputs*; so, input limbs to `FpAdd` and `FpMultiply` are assumed to already be range-checked to 64 bits.

The choice to range-check outputs but not inputs is very natural. If each gadget would range-check its inputs, then range checks would be applied multiple times when you used different gadgets on the same input values, wasting constraints. By following the convention of always constraining witnesses where they originate, we keep the code auditable and also ensure that the same constraints aren't applied multiple times.

The issue is that `secretShare.circom` doesn't follow this convention: None of the input bigint limbs are range-checked. In the abbreviated code below, the signals `c2`, `indexes`, `polynomial` and `secretShare` all represent bigints that are not range-checked.

```
template SecretSharing(THRESHOLD, SHARES, n, k, p) {

    // Public Signals
    // ...
    signal input c2[SHARES][k];
    signal input indexes[SHARES][k];

    // Private Signals
    signal input polynomial[THRESHOLD][k];
    signal input secretShares[SHARES][k];

    // ...

    /// 4- evaluate the polynomial at index_i and
    /// verify p(index_i) = secretShares_i
    component evalPoly[SHARES];
    // ...

    for (var i = 0; i < SHARES; i++) {
        evalPoly[i] = EvalPoly(THRESHOLD, n, k, p);
        for (var j = 0; j < THRESHOLD; j++) {
            for (var l= 0; l < k; l++) {
                evalPoly[i].polynomial[j][l] <== polynomial[j][l];
            }
        }

        for (var j=0; j < k; j++) {
            evalPoly[i].eval[j] <== secretShares[i][j];
            evalPoly[i].at[j] <== indexes[i][j];
        }

        // ...
```

Impact. To understand the impact at the hand of a simple example, let's look at the `ModSum` template which is used under the hood by `FpAdd` to add two limbs with a carry:

```
// addition mod 2**n with carry bit
template ModSum(n) {
    assert(n <= 252);
    signal input a;
    signal input b;
    signal output sum;
    signal output carry;

    component n2b = Num2Bits(n + 1);
    n2b.in <== a + b;
    carry <== n2b.out[n];
    sum <== a + b - carry * (1 << n);
}
```

If both `a` and `b` are constrained to n bits, this template indeed performs addition modulo 2^n. However, if `a` and `b` can be any field element, we can make `a + b` overflow the native field. For example, let $\mathrm{a} = \mathrm{q} - 1$ where q is the native modulus, and $\mathrm{b} = 1$. Then the `sum` is 0, which is not the correct result mod 2^n.

This demonstrates a general principle: Missing range checks often let us introduce overflow of q in *non-native* arithmetic, where it's not supposed to happen. This can make room for the prover to modify the output or input of non-native computations, by adding or subtracting multiples of q.

We explain a concrete attack which breaks the compliance protocol when exploiting this in combination with finding #03; see that finding for details. But the missing range checks alone can likely be exploited to modify either commitments to polynomial coefficients or secret shares.

Recommendation. All limbs of the input signals `c2`, `indexes` and `polynomial` should be range-checked to 64 bits. An appropriate way to do this is to pass the checked limb as input to `Num2Bits(64)` from `circomlib/circuits/bitify.circom`. Notably, `secretShares` does *not* need to be range-checked because it is forced to equal the output of the `EvalPoly` template, which is already range-checked.

Client response. Silent Protocol fixed the issue by adding range checks to `indexes` and `polynomial` as recommended. The signal `c2` was refactored to be a native field element in response to finding #01, so the need for a range check no longer applies to it.

# 03 - Underconstrained native arithmetic on scalars, instance 1

secretSharing.circom

Medium

Description. One of the objectives of the `SecretSharing` template is to prove the correctness of commitments to the polynomial which is used for secret sharing. Commitments are of the form $\mathrm{p_i G}$ where $\mathrm{p_i}$ is a polynomial coefficient and G a constant base point on the BabyJubJub curve.

Polynomial coefficients $\mathrm{p_i}$ are passed in as bigints (arrays of limbs). To perform scalar multiplication, the code collapses the limbs to obtain a single field element which is passed to the `MulBase` template as the scalar.

```
/// 1 - compute polynomial coefficients p_i
/// p = p_0 + p_1 x + p_2 x^2 + ... + p_n-1 x^n-1
signal poly[THRESHOLD];
for (var i=0; i < THRESHOLD; i ++){
    var sum = 0;
    for (var j=0; j<k; j++){
        sum += polynomial[i][j] * ((2 ** n) ** j);
    }
    poly[i] <== sum;
}

/// 2- check broadcast_i = p_i * G
component mulScalars[THRESHOLD];
for(var i=0; i<THRESHOLD; i++) {
    mulScalars[i] = MulBase();
    mulScalars[i].e <== poly[i];\
    // ...
```

There is an interplay between arithmetic in two fields here: the native field and the scalar field. Let's define

- $\mathrm{q} :=$ the native modulus
- $\mathrm{p} :=$ the scalar modulus (i.e. p is the order of the subgroup generated by G).

Note how `poly[i]` is computed: The limbs `polynomial[i][j]` are multiplied with their corresponding power of two and summed, in native field arithmetic. So we end up constraining `poly[i]` to be

$$\mathrm{p_i'} = \sum_{\mathrm{j}=0}^{\mathrm{k}} \mathrm{p_{ij}} 2^{\mathrm{nj}} \mod \mathrm{q}$$

But the value we *actually* would like to put in `poly[i]` is

$$\mathrm{p_i} = \sum_{\mathrm{j}=0}^{\mathrm{k}} \mathrm{p_{ij}} 2^{\mathrm{nj}} \mod \mathrm{p}.$$

This is the scalar that $\mathrm{p_{ij}}$ are supposed to represent, and that is implicitly used elsewhere in the code where the polynomial is evaluated in scalar arithmetic. We also don't care about multiples of p added to $\mathrm{p_i}$, since they are eliminated by scalar multiplication.

The problem is that the prover can easily make p_i differ from p_i', if the limbs p_{ij} can represent values larger than q. Just write

$$p_i = p_i' + c_i q$$

and choose limbs p_{ij} that represent p_i. c_i is an arbitrary value with the requirement that $p_i' + c_i q < (c_i + 1)q$ is within the max value that the limb sum can represent. Note that p_{ij} are entirely controlled by the secret sharing prover and can be chosen to facilitate an attack.

In the audited version of the code, p_{ij} are not range-checked and take values in $[0, q)$, see **finding #02**. Therefore, a limb sum can represent values up to $q2^{(k-1)n}$ (plus a bit more), which means that c_i can be any number in $[0, 2^{(k-1)n})$. Plugging in $k = 4$ and $n = 64$, we see that c_i can take one out of 2^{192} values.

Impact. When picking $c_i \neq 0$, the scalar multiplication creates a modified commitment $A_i' = (p_i - c_i q)G = A_i - c_i qG$. The recipient of the share is supposed to use this commitment to validate their share $p(x_i)$, by checking that

$$p(x_j)G = \sum_{i=0}^{T-1} x_j^i A_i.$$

This fails when one of the commitments is modified, breaking the protocol. Since there are 2^{192} ways for the prover to modify each commitment, there is no way for the share recipient to recover the true commitments $A_i = p_i G$.

Assuming that **finding #02** is fixed and each limb is constrained to the range $[0, 2^n)$, a limb sum can only represent values up to $2^{kn} = 2^{256}$. This means c_i in our analysis can at most be $\lfloor 2^{256}/q \rfloor = 5$. Since commitments can be modified independently and the polynomial degree is currently set to $5 - 1$, there are $6^5 = 7776$ ways for the prover to change the broadcasted commitments. This should be recoverable by a custom trial and error algorithm -- assuming that the secret share $p(x_j)$ is received in untampered form; see **finding #04** for why this is not the case.

Recommendation. There is no need to collapse bigint limbs to a single field element before using them for scalar multiplication. The fact that this is done seems to be just an artifact of the `` `MulBase` `` API which takes a field element. Under the hood, `` `MulBase` `` uses circomlib's `` `BabyPk` `` which unpacks the field element to an array of bits. The bits are passed to `` `EscalarMulFix` `` which contains the core scalar multiplication circuit.

We recommend to unpack the bigint limbs `` `polynomial[i][j]` `` into bits directly. This can serve the double purpose of constraining those limbs to their correct bit size. The resulting bits can be passed to `` `EscalarMulFix` `` to perform scalar multiplication.

Client response. The issue was fixed by Silent Protocol, following our recommendation of using `` `EscalarMulFix` `` directly.

04 - Underconstrained native arithmetic on scalars, instance 2

secretSharing.circom

Medium

Description. After the `` `SecretSharing` `` template evaluated a polynomial on several points to compute secret shares ss_i, the goal is to prove correct encryption of those shares by showing that

$$c_{2i} - ss_i = H(rX).$$

Here, c_{2i} is the encrypted value which is a public input. The code represents c_{2i} and ss_i as bigints and computes the subtraction $c_{2i} - ss_i$ in foreign-field arithmetic, using the `` `BigSubModP` `` template. Then, the bigint version of $c_{2i} - ss_i$ is collapsed into a single field element using native arithmetic, which finally is compared with the hash value on the right hand side.

```
/// 5- verify secret shares are encrypted correctly
subModP[i] = BigSubModP(n,k);
for (var j=0; j <k; j++){
    subModP[i].a[j] <== c2[i][j];
    subModP[i].b[j] <== secretShares[i][j];
    subModP[i].p[j] <== p[j];
}

var sum = 0;
for (var j=0; j<k; j++){
    sum += subModP[i].out[j] * ((2 ** n) ** j);
}

// ...
sum === hashed[i];
```

The way the `` `sum` `` is only constrained modulo the native modulus q creates a problem analogous to finding #03. The bigint stored in `` `subModP` `` can hold values larger than q, which allows the prover to provide multiple modified versions of the public output `` `c2[i][]` ``. Namely, we can use

$$c'_{2i} = c_{2i} + d_i q$$

for a few different values d_i, which in turn will cause the share recipient to obtain the modified decrypted secret share

$$ss'_i = ss_i - d_i q \bmod p.$$

The value d_i has to be small enough such that `` `subModP[i]` ``$\in [0, 2^{256})$ can still hold $c_{2i} - ss_i + d_i q$. Typically, d_i can take 6 different values $0, \ldots, 5$. A crucial point here is that while `` `BigSubModP` `` forces its outputs to be valid bigints with 64-bit limbs, it doesn't require the inputs or output to be smaller than p.

Impact. Allowing 6 different values for the main circuit output c_{2i}, exactly one of which lies in $[0, q)$, might seem non-ideal, but not catastrophic. Can't the recipient of the secret share simply reduce their encrypted share modulo

q to obtain the correct value? The reason we count this as a medium-severity issue is due to its interplay with two other findings:

- **Finding #05** shows that there is ambiguity in the value of the secret share ss_i: It can be modified by a multiple of p. The multiple of p can also be added to c_{2i} without affecting the difference $c_{2i} - ss_i$. With this extra degree of freedom in c_{2i}, there is no longer a unique reduction that a share recipient can perform to obtain a decrypted share in the correct range. A trial-and-error procedure using the broadcast polynomial commitments to validate secret shares might be possible.
- However, as **finding #03** demonstrates, polynomial commitments can themselves be modified by the prover, which further complicates any recovery of the correct secret shares.

In combination, findings #03, #04 and #05 can likely be used to put the compliance protocol in a temporarily broken or hard-to-recover-from state.

Recommendation. There are several general recommendations to make. First, instead of collapsing the bigint $c_{2i} - ss_i$ to a native field element before comparing with another value, consider if it is possible to just compare the bigint limbs directly. This requires both sides to be represented in bigint form.

Second, in the case of a public input, ambiguity can be mitigated by making the verifier (here: the smart contract) validate the input; this would mean checking that each c_{2i} is in $[0, p)$.

Finally, a bigint comparison in the circuit can be used to ensure a value is in its canonical range. Constraining `` `subModP[i]` `` to $[0, p)$ would have prevented the issue.

Client response. In light of both this finding and **finding #01**, Silent Protocol redefined secret share encryption to happen in the native field. c_{2i} is now a native field element and this finding, as well as the recommendations, no longer apply.

# 05 - Secret shares computed ambiguously by not reducing modulo p

polynomial.circom

Medium

Description. The `EvalPoly` template computes the evaluation of a polynomial in non-native arithmetic:

$$s = p(x) = \sum_{j=0}^{T-1} p_j x^j$$

Multiplications use the `FpMultiply` gadget taken from circom-pairing. This gadget allows the reduction $b = a \bmod p$ by witnessing bigints X and b, and proving that $a = pX + b$. There is no check that $b < p$ which would fix the result b to a unique value; let's call a b *canonical* if $b \in [0, p)$. Since X and b are provided as a witness, the prover can easily add multiples of p to b by subtracting from X.

There are good reasons to not do a canonical check after every multiplication. If you need many multiplications, nothing is gained by ensuring every intermediate result is canonical if you only care that the final result is.

The `EvalPoly` template does not enforce that its final output is canonical. Therefore, the prover can modify the result to $s' = s + cp$, where c can be as large as allowed by $s + cp < 2^{256}$. At most, there are $\lfloor 2^{256}/p \rfloor + 1 = 43$ possible values for c.

The output of `EvalPoly` is used as the secret share by `SecretSharing`, so the prover can pick between 43 possible values per secret share.

Impact. There are two ways we found this ambiguity to impact the protocol at different iterations of the code base. In the audited version of the code, the secret share was encrypted in non-native arithmetic. As explained in finding #04, the freedom to add multiples of p helps the prover to modify the final encrypted secret share c_2 in ways that can't trivially be recovered.

A similar impact was found in a later version of the code after Silent Protocol fixed finding #01 and #04 by performing encryption of secret shares in the native field. To collapse the `EvalPoly` output to a native field value, a sum in native arithmetic is performed:

```
// `result[N][i]` holds the polynomial evaluation

var sum = 0;
for (var j=0; j<k; j++){
    sum += result[N][j] * ((2 ** n) ** j);
}
eval <== sum;
```

In this case, after summing in native arithmetic, the 43 possible values for `eval`$= s + cp \bmod q$ are no longer distinguishable by size or by reducing modulo p; some of them will end up being smaller than the unmodified value s. So, yet again, the prover is able to publish incorrect secret shares.

Recommendation. We recommend to add a dedicated canonical check at the end of `EvalPoly` which constrains the `eval` output to a unique value by enforcing that `result[N][j]` is smaller than p.

Client response. Silent Protocol fixed the issue by following our recommendation.

# 06 - Unconstrained booleans leave amount range check unenforced

depositAndWithdraw.circom

Low

Description. The `depositAndWithdraw` circuit handles either deposits or withdraws. In the deposit case, it takes a positive `amount` which is added to the balance. In the withdraw case, `amount` contains the negative amount, i.e. a value of the form $\mathrm{p} - |\mathrm{a}|$ where p is the scalar field size, and the absolute value $|\mathrm{a}|$ is stored in another variable called `isWithdraw`.

To check that the amount is within the range $[0, 2^{30})$, the circuit attempts to encode the following logic:

- In the deposit case, assert that `amount` is smaller than 2^{30}.
- In the withdraw case, assert that `isWithdraw` is smaller than 2^{30}.

The code looks like this:

```
/// 9- Checks that amount/withdraw amount is nonzero and less than 2^30
component isAmountZero = IsZero();
isAmountZero.in <== amount;
component isAmountLessThan = LessThan(252);
isAmountLessThan.in[0] <== amount;
isAmountLessThan.in[1] <== 1 << 30;

component isWithdrawAmountZero = IsZero();
isWithdrawAmountZero.in <== isWithdraw;
component isWithdrawAmountLessThan = LessThan(30);
isWithdrawAmountLessThan.in[0] <== isWithdraw;
isWithdrawAmountLessThan.in[1] <== 1 << 30;

signal depositCheck <-- !isWithdrawCase.out && !isAmountZero.out && isAmountLessThan.out;
signal withdrawCheck <-- isWithdrawCase.out && !isWithdrawAmountZero.out &&
isWithdrawAmountLessThan.out;

depositCheck + withdrawCheck === 1;
```

There are several issues, the most obvious of which is that `depositCheck` and `withdrawCheck` are completeley unconstrained: They only receive their values with the witness assignment `<--` and the only constraints put on them is that their sum is 1. So, nothing in the circuit relates those values to the deposit or withdraw cases, and the prover is therefore not forced to pass `amount` and `isWithdraw` that are within the 2^{30} range.

A second issue is the circular logic used in the `LessThan` checks. The `LessThan(30)` gadget used for `isWithdraw` is only valid with inputs that are at most 30 bits. So, if the `isWithdraw` input is larger than 30 bits, then the output is not necessarily correct, so the entire check is superfluous. The same is true for an `amount` that is larger than 252 bits.

Impact. The reason why this is finding was given a low severity is that range checks on `amount` and `isWithdraw` are present in the contract logic which validates the public inputs to this circuit; therefore, the problem is not actually exploitable.

Recommendation. We don't recommend to leave broken circuit logic in place even if it is without effect. Deployed code is often used as example for other projects, copied from one place to another, etc. Therefore, the recommendation is to either fix the range-checking logic or remove it.

Here is how the conditional check could be written instead:

```
// the number to be checked is either `amount` or `isWithdraw`, depending on the case
signal checkedAmount <== isWithdrawCase.out * (isWithdraw - amount) + amount;

// non-zero check
signal checkedAmountInv <-- checkedAmount != 0 ? 1/checkedAmount : 0;
checkedAmount * checkedAmountInv === 1;

// 30-bit range check
component rangeCheck = Num2Bits(30);
rangeCheck.in <== checkedAmount;
```

Client response. Silent Protocol has addressed the finding by removing the in-circuit range-check.

# 07 - Insecure randomness used for anonymity set sender position

crypto/src/*.ts

Low

Description. The insecure randomness source of `Math.random()` is used to select the index of the anonymity set we are using for all entrypoints to the protocol. Under certain circumstances, an adversary might learn a bias for the random value and thus the anonymity set does not provide perfect indistiguishability between the choices in the set.

Example of one such infraction:

```
const senderIndex = Math.floor(Math.random() * RING_SIZE);
```

Recommendation. We recommend using cryptographically secure randomness sources for the sender index selection. In NodeJS 19 or higher as well as in modern browsers, you can use `crypto.getRandomValues(new Uint8Array(1))[0]`, which returns a single random byte. Since we always want to pick between indices 0 through 7, we can look at the first 3 bits to get our securely random index value:

```
const senderIndex =
  crypto.getRandomValues(new Uint8Array(1))[0] & (RING_SIZE - 1);
```

As long as `RING_SIZE` remains a power of 2, this code will still work when the anonymity set size is changed.

Client response. Silent Protocol pointed out that the finding is out of scope of the audit; it was going to be addressed anyway.

08 - log_ceil() produces off-by-one error

bigint_func.circom

Informational

Description. The `` `log_ceil()` `` helper function is used by `` `FpMultiply` `` to determine the maximum number of bits that certain expressions can have. Expressions arise in the bigint multiplication code and are of the form $\mathrm{S} = \sum_{\mathrm{i}=1}^{\mathrm{k}} \mathrm{a_i}$ where $\mathrm{a_i}$ are known to be n-bit integers. To determine the max number of bits of S, we want to estimate

$$\mathrm{S} = \sum_{\mathrm{i}=1}^{\mathrm{k}} \mathrm{a_i} \leq \mathrm{k} \cdot (2^{\mathrm{n}} - 1) < 2^{\mathrm{m}}$$

Finding the smallest m on the right hand side will give us a precise upper bound on the number of bits that S needs. Assuming we knew that $\mathrm{k} \leq 2^{\mathrm{K}}$, we could infer

$$\mathrm{k} \cdot (2^{\mathrm{n}} - 1) < 2^{\mathrm{n+K}}$$

so we find $\mathrm{m} = \mathrm{n} + \mathrm{K}$. Choosing the minimal K will make m minimal as well. Note here that the inequality on k is not strict -- it's fine if $\mathrm{k} = 2^{\mathrm{K}}$, thanks to the -1 in the other factor.

Taking logs, we get the condition

$$\log_2(\mathrm{k}) \leq \mathrm{K}$$

and so the minimal K can be written as $\mathrm{K} = \lceil \log_2(\mathrm{k}) \rceil$. The maximum number of bits of S is precisely $\mathrm{n} + \lceil \log_2(\mathrm{k}) \rceil$. This motivates the use of a `` `log_ceil()` `` function for this problem.

The `` `log_ceil()` `` function at the time of the audit was implementing something slightly different, namely, it was computing the *number of bits* of k.

Here are some examples:

Input k	Input (Binary)	Expected output `` `log_ceil(k)` ``	Actual output = number of bits
1	1	0	1
2	10	1	2
3	11	2	2
4	100	2	3

The difference is for k a power of two, when the actual implementation returns something too large by 1.

Impact. In the current usages of the function, its incorrect result is not a security issue. Because of the specific use as an upper bound, returning something too large is just more conservative than necessary. If used in other contexts in the future, it could lead to more serious issues.

Recommendations. We recommend fixing `` `log_ceil()` ``, first because off-by-one issues could become a problem when used in other circumstances, and second because using more bits than necessary implies using more constraints than necessary.

Furthermore, at the time of audit `log_ceil()` defaults to 254 for any input larger than 254 bits. Again this is fine because inputs that large are not used in practice. We recommend adding an assertion preventing this case from being exercised.