ZKSECURITY

Audit of Reclaim Protocol's ChaCha20 Circuit

September 23rd, 2023

Introduction

On July 31st, 2023, zkSecurity was tasked to audit Questbook's ChaCha20 circuits for use in the **Reclaim Protocol**.

One consultant spent a week looking for security issues. A number of findings were reported in this document.

Two days of review were added to the audit to help Questbook fix the issues. No issues were found on the end result and we detail the fixes in the findings section of this document (under each original findings).

Scope

The implementation used Circom, targeting the Groth16 proof system, and offering a flexible API for the user to use. In addition, it purposefully does not provide authentication over the ciphertext as the Reclaim Protocol does not need it. Oracles can see the ciphertexts and authenticate their integrity by passing them as public inputs to the circuit when verifying the proofs.

The audit followed **RFC 7539: ChaCha20 and Poly1305 for IETF Protocols** in order to ensure that the implementation matched the specification.

Review phase

To address original findings and subsequent reviews, Questbook ended up changing its approach to encode all bitwise logic in the circuits with bit-level constraints. This last subsection summarizes the changes and the reasoning behind them.

Questbook uses the Groth16 proof system, which works with circuits that are encoded using the R1CS arithmetization. Such an arithmetization allows someone to encode a circuit as a list of quadratic constraints which are often referred to as the rows or constraints of the circuits. Specifically, each constraint has full access to a list of "witness values" (that usually starts with the public input values), and can encode the multiplication of two linear combinations of that witness list and enforce that it is equal to another linear combination of that same witness list.

In other words, for a witness vector $\vec{w} = (w_0, w_1, \cdots)$ we can add new constraints by hardcoding the scalar values a_i, b_i, c_i which will enforce the following:

$$(a_0 w_0 + a_1 w_1 + \cdots)(b_0 w_0 + b_1 w_1 + \cdots) = c_0 w_0 + c_1 w_1 + \cdots$$

Circuit size is usually paramount to a zero-knowledge application, as it directly impacts the proof size and prover time. As such, reducing a circuit is often a priority.

Non-snark friendly cryptography primitives, like ChaCha20 implemented here, are often full of bitwise operations (e.g. XOR, ROT, etc.) which are not efficient to encode in such arithmetization systems. This often leads to circuits blowing up.

Some systems accelerate unfriendly low-level operations by making use of lookup tables or higher-degree constraints (that fits in a single row). Groth16 does not support any of these. For this reason, bitwise operations

in the ChaCha20 circuit have to be encoded by first unpacking values down to their bit representations, and then acting on the bits directly. Later, bits can be packed back into field elements if needed.

Originally, the ChaCha20 circuit attempted to save constraints by avoiding the packing and unpacking of values. Dealing with bits directly is undesirable as each value being unpacked leads to a growth of the witness size by the number of bits. Then each bit must be constrained to ensure that they can only be 0 or 1 and nothing else. These "bit constraints" are quadratic constraints (i.e. $x(x-1)=0$) and thus cost one constraint/row in R1CS for each of them. That is unlike the packing and unpacking operations ($x=\sum_i x_i 2^i$) which are linear and thus can be encoded in a single constraint/row.

Unfortunately, as it was found in the original review contained which forms the base of this report, that approach was found not to be sound.

The circuits were subsequently refactored to perform all bitwise operations on the bits of the values directly. In addition, packing and unpacking operations are removed as bits can be passed from one operation to the other without paying the cost of constantly re-encoding.

Findings

Below are listed the findings found during the engagement.

ID	COMPONENT	NAME
00	generics.circom	Unsound Addition Gadget
01	generics.circom	Unsound Left Rotation Gadget
02	generics.circom	Unsound XOR gadget
03	chacha20.circom	Potentially Easy-to-Misuse Interface

00 - Unsound Addition Gadget

generics.circom

Description. The `Add32Bits` gadget constrains the addition of two (assumed to be 32-bit) values to wrap around. In other words, if the addition of two 32-bit values overflows, the gadget removes the carry bit (the 33rd most-significant bit).

To do that, the logic witnesses a carry bit, which is used to remove the carry from the result if set to 1:

```
tmp <-- (a + b) >= (0xFFFFFFFF + 1) ? 1 : 0;
tmp * (tmp - 1) === 0;
out <== (a + b) - (tmp * (0xFFFFFFFF + 1));
```

The problem is that the carry bit `tmp` is not constrained to be correctly computed. That is, there are two scenarios in which this function can be maliciously used:

1. If $a + b$ are overflowing (the result is 33 bits), then you can set `tmp=0` and the output will be 33 bits
2. If $a + b$ is not overflowing (the result is 32 bits), then you can set `tmp=1` and the output will underflow (it should be around the bit size of the circuit field)

Both are problems if the output is not constrained to be 32 bits on the caller side, which seems to be the case.

Recommendations. In addition to fixing the issue, document the function to warn callers that they must be ensure that the two inputs are well-constrained to be 32-bit values.

Client response. The client fixed the issue by encoding the operation on the bits of the operands. This ensured that the result of adding the two 32-bit values had a unique decomposition of 32 bits and a carry bit.

Specifically, they add the two packed values together (as field elements):

$$r = \sum_{i=0}^{31} a_i 2^i + \sum_{i=0}^{31} b_i 2^i$$

and then unpack them in a 32-bit result $(r_0, \cdots, r_{31})$ plus a carry c:

$$r = \sum_{i=0}^{31} r_i 2^i + 2^{32} c$$

where a_i, b_i are the inputs considered to be valid values, and the rest of the variables (r_i and c) are constrained to be 0 or 1.

01 - Unsound Left Rotation Gadget

generics.circom

Description. The `RotateLeft32Bits` gadget is used to perform left rotations of (assumed to be 32-bit) values. The rotation is parameterized by an argument given to the Circom template.

The following snippet shows where the issue lies in:

```
signal part1 <-- (in << L) & 0xFFFFFFFF;
signal part2 <-- in >> (32 - L);
(part1 / 2**L) + (part2 * 2**(32-L)) === in;
```

As one can see, both `part1` and `part2` are witness values that are constrained only by the last line.

The problem is that the constraint is not enough, and both `part1` and `part2` can be chosen arbitrarily. For example, by following these steps:

- fix $\mathrm{part2}$ arbitrarily
- set $\mathrm{part1} = (\mathrm{in} - \mathrm{part2} \cdot 2^{32-\mathrm{L}}) \cdot 2^{\mathrm{L}}$

Using **sage**, one can easily find such values:

```
# default circom circuit field
p = 21888242871839275222246405745257275088548364400416034343698204186575808495617
F = GF(p)

# settings
in_ = F(5) # input is 5
L = 3 # rotation of 3

# fix part2 to 2, for example
part2 = F(2)

# compute part1 as shown above
part1 = (in_ - part2 * 2^(32-L)) * 2^L
part1
21888242871839275222246405745257275088548364400416034343698204186567218561030

# the constraint passes
(part1 / 2^L) + (part2 * 2^(32-L)) == in_
True
```

Recommendations. Constrain `part1` (resp. `part2`) to be $32 - \mathrm{L}$ (resp. L) bit-sized values.

In addition, add documentation to ensure that the caller is aware that the inputs have to be well-constrained 32-bit values. Consider changing the name of the function to emphasis that it is not validating the inputs. (Usually the keywords `unchecked` or `unsafe` appended at the end of the template name are good indicators.)

Client response. The client fixed the issue by manually re-ordering bits based on a shift (`L`):

```
for (var i = 0; i < BITS; i++) {
    out[i] <== in[(i + L) % BITS];
}
```

As one can see using zkSecurity's **circomscribe tool** (introduced **here**), the circuit correctly routes the bits:

0 backward forward reset

```
template RotateLeftBits(BITS, L) {
    signal input in[BITS];
    signal output out[BITS];
    for (var i = 0; i < BITS; i++) {
        out[i] <== in[(i + L) % BITS];
    }
}

component main = RotateLeftBits(5,2);
```

```
RotateLeftBits.in[2] - 1 * RotateLeftBits.out[0] = 0
- 1 * RotateLeftBits.out[1] + RotateLeftBits.in[3] = 0
- 1 * RotateLeftBits.out[2] + RotateLeftBits.in[4] = 0
- 1 * RotateLeftBits.out[3] + RotateLeftBits.in[0] = 0
- 1 * RotateLeftBits.out[4] + RotateLeftBits.in[1] = 0
```

02 - Unsound XOR gadget

generics.circom

Description. The `XorWords` gadget constraints the XOR of two arrays of `M`-bit values (for some fixed array size and value `M`).

In order to implement the XOR in-circuit, the logic implements the XORs bit by bit. It was found that the bit constraints for the witness values were commented at the time of the audit:

```
abits[l] <-- ain >= j ? 1 : 0;
bbits[l] <-- bin >= j ? 1 : 0;
// ensure abits[l] and bbits[l] are either 0 or 1
// below should be uncommented in prod?
// abits[l] * (abits[l] - 1) === 0;
// bbits[l] * (bbits[l] - 1) === 0;
xors[l] <== abits[l] + bbits[l] - 2 * abits[l] * bbits[l];
```

In addition, the bit decomposition of both operands `a` and `b` must be constrained to be correct. While the gadget's logic attempt to do just that, it was found that the check was flawed.

Iteration of each bitstring goes through the following logic, cleaned up to showcase only the checks applied to the variable `a[i]`:

```
ain = a[i];

// TRUNCATED...

for(j = 2 ** (M-1);j >= 1;j /= 2) {
    abits[l] <-- ain >= j ? 1 : 0;

    // TRUNCATED...

    ain -= abits[l] * j; // ain -= bit[j] * 2^j

    // TRUNCATED...
}

ain * a[i] === 0;
```

As you can see, the last check constraints that `ain * a[i] == 0`. This is true if either `ain = 0` or `a[i] == 0`.

What we really want to check is that `ain = 0`, but the check doesn't apply if the value of `a[i]` is 0.

In the case where the value of `a[i]` is 0, then a malicious prover can use an arbitrary bit decomposition for `a[i]`.

Recommendations. Use circomlib's Num2Bits function to convert both operands to bitstrings.

Client response. The client fixed the issue by enforcing an XOR constraint ($\mathrm{res} = \mathrm{a} + \mathrm{b} - 2\mathrm{ab}$) on each bits individually. As one can see using zkSecurity's **circomscribe tool** (introduced **here**), the constraint is correctly enforced by the circuit (which includes correctly constraining the quadratic term):

0 backward forward reset

```
template XorBits(BITS) {
    signal input a[BITS];
    signal input b[BITS];
    signal output out[BITS];
    var mid[BITS];

    for (var k=0; k<BITS; k++) {
        mid[k] = a[k]*b[k];
        out[k] <== a[k] + b[k] - 2*mid[k];
    }
}

component main = XorBits(32);
```

```
(2 * XorBits.a[0] )*(XorBits.b[0] ) = XorBits.b[0] + XorBits.a[0] - 1 * XorBits.out[0]
(2 * XorBits.a[1] )*(XorBits.b[1] ) = XorBits.b[1] - 1 * XorBits.out[1] + XorBits.a[1]
(2 * XorBits.a[2] )*(XorBits.b[2] ) = - 1 * XorBits.out[2] + XorBits.b[2] + XorBits.a[2]
(2 * XorBits.a[3] )*(XorBits.b[3] ) = XorBits.b[3] + XorBits.a[3] - 1 * XorBits.out[3]
(2 * XorBits.a[4] )*(XorBits.b[4] ) = XorBits.a[4] - 1 * XorBits.out[4] + XorBits.b[4]
(2 * XorBits.a[5] )*(XorBits.b[5] ) = XorBits.a[5] + XorBits.b[5] - 1 * XorBits.out[5]
(2 * XorBits.a[6] )*(XorBits.b[6] ) = - 1 * XorBits.out[6] + XorBits.a[6] + XorBits.b[6]
(2 * XorBits.a[7] )*(XorBits.b[7] ) = - 1 * XorBits.out[7] + XorBits.b[7] + XorBits.a[7]
(2 * XorBits.a[8] )*(XorBits.b[8] ) = XorBits.b[8] + XorBits.a[8] - 1 * XorBits.out[8]
(2 * XorBits.a[9] )*(XorBits.b[9] ) = - 1 * XorBits.out[9] + XorBits.a[9] + XorBits.b[9]
(2 * XorBits.a[10] )*(XorBits.b[10] ) = - 1 * XorBits.out[10] + XorBits.a[10] + XorBits.b[10]
(2 * XorBits.a[11] )*(XorBits.b[11] ) = - 1 * XorBits.out[11] + XorBits.a[11] + XorBits.b[11]
(2 * XorBits.a[12] )*(XorBits.b[12] ) = - 1 * XorBits.out[12] + XorBits.a[12] + XorBits.b[12]
```

03 - Potentially Easy-to-Misuse Interface

chacha20.circom

Description. The ChaCha20 interface exposed by the library takes a key, nonce, a starting counter, and a plaintext (or ciphertext as the encryption and decryption algorithm is the same).

As there's usually a common way to initialize the nonce part of the algorithm, it might be safer to not have the user provide it. As per RFC 7539:

> ***"A 32-bit initial counter. This can be set to any number, but will usually be zero or one. It makes sense to use one if we use the zero block for something else, such as generating a one-time authenticator key as part of an AEAD algorithm."***

In addition, the interface seems to be easy to misuse as it does not constrain any of the inputs provided (which are assumed to be 32-bit inputs).

At the very least, the documentation should warn the user that they must take care of constraining the inputs to be 32-bit values. Changing the template name to `ChaCha20Unsafe` or `ChaCha20Unchecked` is a good way to warn users that they must be careful when using this interface.

In addition, it might make sense to provide an alternative interface that enforces these checks on all the inputs.