Tornado Cash

Emanuel Cornaggia, Alessandro Genova

February 2025

Contents

Abstract

The rise of blockchain technology has introduced unprecedented levels of financial transparency through public ledgers like Ethereum. However, this transparency also raises significant privacy challenges.Tornado Cash,a*smart contracts*decentralized and non-custodial on Ethereum, addresses these issues by allowing users to obfuscate transaction history through advanced cryptographic techniques, such as*zero knowledge proofs*.

Since its launch, Tornado Cash has fueled heated debate within the blockchain community and beyond. While it represents a legitimate tool to enhance privacy and financial sovereignty, it has also attracted the attention of regulators due to the risk of its use for illicit activities, such as money laundering and economic sanctions evasion. The controversy reached its peak in 2022, when the U.S. Department of the Treasury's Office of Foreign Assets Control (OFAC) imposed sanctions against the protocol[21], marking one of the first legal interventions against a system based on*smart contracts* decentralized. However, in November 2024, a US court overturned the sanctions, ruling that the*smart contracts*Tornado Cash's immutable assets cannot be considered property and, therefore, do not fall under the restrictions imposed by OFAC.

In this work we will explain how Tornado Cash works, including contracts and DAO. We will look at the underlying mathematical structures. We will implement the contracts on a Testnet and discuss the controversies related to the use of the protocol and the current state.

Introduction

One of the main differences between the Ethereum blockchain and the Bitcoin blockchain is the management of wallets, change addresses, and transaction models.

2.1 Bitcoin UTXO Model

Bitcoin adopts a transaction model based onUTXO (Unspent Transaction Output), where each transaction spends one or more unspent outputs (UTXOs) as inputs and generates a new UTXO for each output. When a user makes a transaction, the wallet selects a set of UTXOs to cover the requested amount. If the sum of the inputs exceeds the transaction amount, the remaining balance is sent to a change address (*change*), controlled by the sender's wallet[18].

2.2 Ethereum Account-Based Model

In the case of the Bitcoin blockchain, the central element is represented by the Bitcoin cryptocurrency itself, around which all exchange operations develop. On the contrary, in the Ethereum blockchain, the Ether cryptocurrency mainly plays the role of "fuel" necessary for the functioning of the network, while the most important tools are thesmart contracts hey token,both fungible and non-fungible. Furthermore, through specific*smart contracts*, it is possible to temporarily lock in a certain amount of tokens through a mechanism known asstaking,commonly used within decentralized finance (DeFi) contracts.

Ethereum uses a modelaccount-based (instead of UTXO) because it is more suitable for managing*smart contracts*complex. This model allows for direct updates of global state, including account balances, contract variables, and other persistent information. It also simplifies programmability by making it easier for contracts and accounts to interact, and by avoiding the transaction fragmentation typical of the UTXO model, which does not natively handle persistent state. Each Ethereum address has a persistent balance that

is directly modified by transactions. In this system there are no UTXOs; transactions simply deduct the amount from the sender's balance and credit it to the recipient[10].

2.3 Privacy and Anonymity: Introduction to Tornado Cash

Having a fixed personal account means that all transactions are completely visible to anyone, almost completely eliminating privacy, which remains only in linking the wallet address to a physical person.

Tornado Cash was created with the idea of improving privacy by ensuring anonymity on transactions. It can be particularly useful for activists, such as those operating in countries with repressive regimes, journalists who cover sensitive topics, or even companies that want to protect their financial transactions. It can also support donors who contribute to controversial or politically sensitive causes, without risking identification or prosecution.

It's a "*non-custodial protocol*", meaning users maintain full control of their funds and private keys, without having to entrust them to a third party. Thanks to the*smart contracts*, this protocol is visible to everyone and cannot be changed, not even by the developer. The main idea is the following:

1. The user sends a certain amount of cryptocurrency that he intends to use to make the payment to the Tornado Cash contract.
2. The user generates proof of having made a deposit.
3. Using this proof, any other user can withdraw the funds to a different wallet, without there being a direct connection to the initial deposit.

Anyone who wants to monitor the transactions of our address will be able to see that we have sent cryptocurrency to the Tornado Cash contract, but will not be able to trace the recipient of the withdrawal. Likewise, if we receive funds from Tornado Cash, the observer will be able to see the credit but will not be able to identify the sender's address.

2.4 Supported Tokens and Chains

The original implementation, in addition to Ether (ETH), provides for the use of the following tokens (always on the Ethereum chain):

- COME ON
- cDAI
- USDC

- USDT
- WBTC

With the latest version updated to 2021, Tornado Cash has been made available, in addition to Ethereum, also for the following chains:

- Binance Smart Chain
- Polygon Network
- Gnosis Chain (former xDAI Chain)
- Avalanche Mainnet
- Optimism
- Arbitrum One

2.5 Tornado Cash Versions

There are two versions of Tornado Cash:

- Tornado Cash Classic
- Tornado Cash Nova

In Tornado Cash Classic, when a user deposits cryptocurrency into a pool, a secret is generated. This secret works like a private key: a user who has it can withdraw the funds to another address at any time.

"In Tornado Cash Nova, the funds are directly tied to the address used for deposit, without resorting to a secret. The user can withdraw them by connecting to the pool through the same address. The robustness of the protocol relies on the number of users and the size of the pool (the more transactions occur, the more difficult it is to perform an analysis that reconnects the transactions)

Primitive

3.1 zk-SNARKs

In this paragraph we will talk briefly about zk-SNARKs, as we have dedicated a separate chapter to talk about them in much more depth.

3.1.1 ZKP

A Zero-Knowledge Proof (ZKP) is a cryptographic protocol that allows one party (called the prover) to convince another party (called the verifier) that a statement is true, without revealing any information beyond the simple fact that the statement is true.

This concept was introduced in 1985 by Goldwasser, Micali and Rackoff in their paper ”The Knowledge Complexity of Interactive Proof-Systems”[13].

A Zero-Knowledge Proof protocol must satisfy three essential properties:

- Completeness:If the statement is true and the prover follows the protocol correctly, the verifier will accept the proof with high probability.
- Correctness:A dishonest prover cannot fool the verifier into accepting a false statement, except with negligible probability.
- Zero Knowledge:The proof does not reveal any useful information about the underlying secret, other than the truth of the claim.

These properties ensure that the verifier can be certain that the statement is correct without obtaining details that compromise the confidentiality of the secret.

ZKPs can be classified into two main categories

- Interactive tests:require an exchange of messages between prover and verifier to reach the belief in the validity of the statement. An example is the Schnorr protocol for authentication.

- Non-interactive tests (NIZK):eliminate the need for interaction, allowing verification via a single data transmission. The Fiat–Shamir heuristic can be used to transform some interactive ZKPs into NIZKs.

3.1.2 zk-SNARKs

The Zero-Knowledge Succinct Non-Interactive Argument of Knowledge is a subclass of NIZK.

The main features are:

- Zero Knowledge:The proof does not reveal any information about the underlying secret, other than the truth of the claim.
- Succinctness:The proof is small in size and can be verified in polynomial time with respect to the size of the input.
- Non-interactive:The protocol does not require interaction between prover and verifier.
- Argument of Knowledge:The prover can generate a valid proof only if it actually possesses knowledge of the witness (it contains values and secrets that satisfy a certain mathematical statement or relation).

zk-SNARKs are designed to be efficient and can be used to test a large number of statements at once.

3.2 Pedersen Hash

[19] Let it be G a cyclic group of order q first and characteristic p, generated by an element g, with a second generator h chosen so that the discrete logarithm problem (DLP) between g And h be computationally intractable (in other words, it must be very difficult to compute the exponent x such that $h = g^x$).

The Pedersen Hash of a Message m is defined as:

$$H(m) = g^m h^r \pmod{p}$$

Where:

- m Andthe message to be hashed, considered as a numeric element of the underlying finite field
- r Anda random value chosen uniformly in Z_q, used to introduce entropy and obtain security properties
- g And h they are generators of the group G

Property:

- Collision Resistance:if the group GAndlarge enough and the discrete logarithm problem is difficult, find two distinct messages m_1, m_2 such that $H(m_1) = H(m_2)$ is computationally difficult.
- Hiding:The hash value does not reveal direct information about the message m, since it is masked by the random value r.
- Binding:if $r$it is not known, it is computationally difficult to find another message m', and a random value r'such that $H(m) = H(m')$.

Pedersen hash is designed to optimize computational efficiency within Zero-Knowledge circuits.

Hashing a message using the Pedersen hash function compresses the bits of the message to a point on an elliptic curve called the Baby Jubjub. This curve is defined within the BN128 elliptic curve, which is supported by pre-built operations on the Ethereum network introduced with EIP-196. As a result, operations that use the Baby Jubjub curve, such as Pedersen hashing, are highly gas efficient.

When computing the Pedersen hash of a message, the resulting point on its elliptic curve is extremely efficient to verify, but the inversion process to recover the original message is computationally intractable.

3.3 MiMC Hash Function

[1] Minimized Multiplicative Complexity is a cryptographic algorithm specifically designed to optimize computational efficiency in zero-knowledge proof (ZKP) contexts, such as SNARKs and STARKs. Its structure was developed to reduce multiplicative complexity, a critical factor in zero-knowledge proof protocols, where the computational costs of modular multiplications directly affect the efficiency of the system.

Specifically, MiMC is an iterative hash function built on a Feistel network approach.

Is $x$the message input and ka secret key (optional). The round function is iterated over a number T of rounds with the following transformation:

$$x_{the+1} = (x_{the} + C_{the} + k)_{And} \pmod{p}$$

Where:

- C_{the}Anda predefined and public round constant
- AndAnda small odd exponent
- pAnda large first that defines the underlying field F_p

- kAndan optional key value

The final hash value is the output after T iterations of the transformation described above.

MiMC provides cryptographic security by exploiting the nonlinearity introduced by the exponent$And$and the difficulty of the discrete logarithm problem in the finite fieldF_p. Key properties include:

- Preimage resistance:given the modular operations with an exponent, the function is difficult to invert.
- Collision Resistant:Using different round constants ensures that no two distinct inputs will produce the same hash with significant probability. Using an odd exponent helps avoid symmetries (which we would have with an even exponent:$x_{And} = (-x)_{And}$).
- Low computational complexity:the cost of modular operations is reduced by using a single small fixed exponent, minimizing the number of multiplications needed compared to other cryptographic constructions based on more complex permutations.

3.4 Merkle Tree

[17] Also known as a binary hash tree, it is a hierarchical data structure used in cryptography and distributed computing to ensure the integrity and verifiability of information in an efficient and secure manner. It is named after its inventor, Ralph Merkle, who introduced it in the 1980s as part of his research into secure methods of data verification in distributed settings.

The Merkle tree is essentially a binary tree where leaf nodes represent hashes of individual data, while internal nodes contain hashes of child nodes. The root of the tree, known as the Merkle root, represents an aggregate hash of all the underlying data, which can be used to verify the integrity of the entire data set in a compact way.

A Merkle tree is constructed iteratively in a way that reduces the number of hashes needed to verify the integrity of the data. Its structure is as follows:

- Leaf nodes:Each leaf node of the tree represents a hash of the original data, denoted as$H(d_{the})$, Whered_{the}Andthe original data item.
- Internal nodesEach internal node is obtained by calculating the hash of the concatenation of its children's hashes. For example, if a node has two children TOAnd$B$, the parent node will contain the hash$H(A \,//\, B)$ (Where$//$is the concatenation operator).

- Mertkle Tree Root isthe final hash of the tree. It represents a single string of data that can be used to verify the integrity of the entire tree. It contains information about all leaf nodes.

The Merkle tree is particularly useful for large data structures, as it greatly reduces the amount of data needed to verify the integrity of a single element.

Property:

- Data integrity:The Merkle root provides a compact summary of the entire data set, and any change to a single piece of data (e.g., a leaf of the tree) will cause a significant change in the root hash. Therefore, if the root remains unchanged, the integrity of the underlying data is guaranteed.
- Efficient Verifiability:One of the main strengths of the Merkle tree is the ability to verify the integrity of a single piece of data without having to download or verify the entire tree. To verify a single piece of data, it is sufficient to know the hash of the root and a “Merkle proof,” which consists of the hashes of the internal nodes along the path from the leaf to the root. This reduction in the amount of data to verify makes Merkle trees ideal for applications with large amounts of distributed data.
- Collision Resistance:If the underlying hash function is collision-resistant, then the Merkle tree will also be collision-resistant. In other words, it will be computationally difficult for an attacker to find two distinct data sets that generate the same Merkle root.

How Smart Contracts Work

In this section, we analyze the technical functioning of Tornado Cash, examining the cryptographic mechanisms and data structures that ensure the privacy of transactions. We will delve into the following fundamental components:

- Deposit:the mechanism by which users lock funds into the contract.
- Merkle Tree:the structure used to organize the deposits and verify the validity of the evidence.
- Generating the test:the cryptographic process that allows one to demonstrate ownership of a bond without revealing its value.
- Withdrawal:the withdrawal stage, where a user can redeem funds anonymously to a different address.

Each smart contract on Ethereum is identified by a unique address, known as aContract Address (CA).There are multiple contracts that provide different functionalities. The contract you need to interact with to deposit and withdraw cryptocurrency is called TornadoRouter. This contract acts as a single entry point: when you interact with it, you pass as input the contract related to the cryptocurrency pool you want to deposit or withdraw. The advantage of this choice is that the pool contracts can remain fixed, and if there is a need to apply a change to the protocol, you can deploy another TornadoRouter. It should also be noted that in the Classic version of Tornado Cash, the pools represent the cryptocurrency and a fixed denomination (for ETH there are pools of 0.1, 1, 10 and 100 ETH). This choice was made to make it more difficult any type of analysis in which you want to reconnect the same amount of cryptocurrency of a deposit to that of a withdrawal. The address of TornadoRouter is the following:

0xd90e2f925DA726b50C4Ed8D0Fb90Ad053324F31b

The contract was distributed on the blockchain through a specific transaction, identified by aTransaction Hash.This hash represents the operation ofsmart contract creationon the Ethereum network. The corresponding transaction is:

0x51baa3dbb7e8756d52ba1b863f41e7af38373d0bb9943b9585d7f0db2d419e6e

Let's analyze the code of the first pool created, in fact it is called Tornado-Cash eth and does not specify the ETH denomination, which is 10. To directly analyze the source code of the contract, you can consult it on Etherscan at the following link:
https://etherscan.io/address/0x910cbd523d972eb0a6f4cae4618ad62622b39dbf#code [11]

4.1 Deposit

```
1        /**
2        @dev Deposit funds into the contract. The caller must send (for
        ETH) or approve (for ERC20) value equal to or `denomination ` of this
        instance. @param _commitment the note commitment , which is
3        PedersenHash(nullifier + secret)

4     * /
5     function deposit(bytes32 _commitment) external
        payable nonReentrant {
6        require (! commitments[_commitment], "The commitment has been
         submitted ");
7
8        uint32 insertedIndex = _insert(_commitment);
9        commitments[_commitment] = true;
10       _processDeposit();
11
12       emit Deposit(_commitment , insertedIndex , block. timestamp);

13    }
```

A user generates two prime numbers with length 31 bits. We call the first numbernullifierwhile the secondsecret (as it will represent the secret to redeem the sum). Nullifier and secret are concatenated and the function is applied to their result.Pedersen hash.This hash represents ourcommitment (the parameter _commitmentof the function deposit).

This commitment generated by us is passed as a parameter to the function _insert,which will insert it into theMerkleTreeand will give us back theindexwhere the commitment was inserted.
commitments [_commitment] = true ;adds the commit to the commitment dictionary, setting its value to true. This step is done to avoid inserting the same commitment twice in the tree. The check is done just above:require(!commitments[_commitment])

_processDeposit();This is used to check that the funds have been sent to the correct pool.

Finally, through the call

emit Deposit(_commitment, insertedIndex, block.timestamp);,a Deposit event is generated, which is recorded in the transaction log. Since events are not directly accessible from the contract once issued, this recording allows to keep a permanent trace of the operation on the blockchain. Through this event is obtained as the index of the leaf of the Merkle Tree in which the commitment has been inserted.

4.2 Merkle Tree

The Merkle Tree construction in Tornado Cash is done only once when the contract is deployed on the blockchain. This happens in the constructorconstructor(uint32 treeLevels)of the classMerkleTreeWithHistory, where the tree is initialized with a default depth level (in the case of Tornado Cash, 20 levels). This allows the pool to be used for 2_{20}
deposits, then a new one will have to be created.

After this phase, the tree is ready to receive new deposits, which will be inserted in real time via the functioninsert(),updated whenever a new one *commitment*is recorded.

4.2.1 Construction

```
contract MerkleTreeWithHistory {
    uint256 public constant FIELD_SIZE = 218882428718392
      752222464057452572750885483644004160343436982041865
      75808495617;
    uint256 public constant ZERO_VALUE = 216638390044169
      329453823559087905992252665018229079114575049785155
      78255421292; // = keccak256(" tornado ") % FIELD_SIZE

    uint32 public levels;

    // the following variables are made public for
      easier testing and debugging and
    // are not supposed to be accessed in regular code bytes32[]
                        public      filledSubtrees;
    bytes32[]       public      zeros;
    uint32 public currentRootIndex = 0; uint32 public
    nextIndex = 0;
    uint32 public constant ROOT_HISTORY_SIZE = 100;
    bytes32[ROOT_HISTORY_SIZE] public roots;

    constructor(uint32 _treeLevels) public {
```

```
        require(_treeLevels > 0, "_treeLevels should be greater than
        zero");
        require(_treeLevels < 32, "_treeLevels should be less than
                        32");
        levels =        _treeLevels;

        bytes32 currentZero = bytes32(ZERO_VALUE);
        zeros.push(currentZero);
        filledSubtrees.push(currentZero);

        for (uint32 i = 1; i < levels; i++) {
            currentZero = hashLeftRight(currentZero , currentZero);

            zeros.push(currentZero);
            filledSubtrees.push(currentZero);
        }

        roots[0]       =   hashLeftRight(currentZero ,                currentZero)
      ;
    }

    // Other functions follow in the original code
```

The Merkle Tree used by Tornado Cash takes as input a depth level, which is between 0 and 32 endpoints, exclusive.

To obtain this information we have two options:

- Interacting with the contract and requesting the field valueuint32 public levels.
- Go to the contract creation transaction and look at the inputs provided.

Using these two methods, it turns out that the height that has been decided to be attributed to the Merkle Tree is 20.

The valueZERO_VALUE iscalculated as follows:keccak256("tornado") % FIELD_SIZE. This can be easily verified with the following python script:

```
import sha3                # install safe -pysha3

FIELD_SIZE = 21888242871839275222246405745257275088548
     364400416034343698204186575808495617

keccak = sha3.keccak_256()
keccak.update(b"tornado ")

print(int(keccak.hexdigest (), 16) % FIELD_SIZE)
```

The code

```
bytes32 currentZero = bytes32(ZERO_VALUE);
zeros.push(currentZero);
filledSubtrees.push(currentZero);
```

creates an initial value and uses it to fill the tree by adding it to the arrays zerosAndfilledSubtrees.These arrays are used to represent theempty knots heyfilled undertrees,respectively.

The code iteratively proceeds to build the MerkleTrees by calculating the hash of the leaves (which currently have valuecurrentZero)to get the parent hash of the two leaves, then calculate the hash of the rootMerkleTree.

Hashes are computed by the MiMC algorithm using the library implementationHasher.MiMCSponge:

```
        /**
      @dev Hash 2 tree leaves , returns MiMC(_left , _right)

  * /
  function hashLeftRight(bytes32 _left ,                    bytes32     _right
    ) public pure returns (bytes32)
      { require(uint256(_left) <              FIELD_SIZE ,       "_left       should
      be inside the field ");
      require(uint256(_right) < should be   FIELD_SIZE ,       "_right
    inside the field"); uint256 R =
      uint256(_left); uint256 C = 0;

      (R, C) = Hasher.MiMCSponge(R, C);
      R = addmod(R, uint256(_right), FIELD_SIZE); (R, C) =
      Hasher.MiMCSponge(R, C);
      return bytes32(R);
  }
```

4.2.2 Insertion

Previously we saw that when we make a deposit and pass the commitment to the functiondeposit,the smart contract calls the method _insert,to go and insert it into the Merkle Tree:

```
      function _insert(bytes32 _leaf) internal returns( uint32 index) {

      uint32 currentIndex = nextIndex; require(currentIndex !=
      uint32(2)**levels , "Merkle tree is full. No more leafs can be added");
      nextIndex += 1;

      bytes32 currentLevelHash = _leaf; bytes32
                  left;
      bytes32     right;
```

```
8
9           for (uint32 i = 0; i < levels; i++) {
10              if (currentIndex % 2 == 0) {
11                  left =        currentLevelHash;
12                  right =        zeros[i];
13
14                  filledSubtrees[i]               =   currentLevelHash;
15              }   else    {
16                  left      =   filledSubtrees[i];
17                  right      =   currentLevelHash;
18              }
19
20              currentLevelHash = hashLeftRight(left , right);
21
22              currentIndex        /=    2;
23          }
24
25          currentRootIndex = (currentRootIndex + 1) %
            ROOT_HISTORY_SIZE;
26          roots[currentRootIndex] = currentLevelHash; return
27          nextIndex - 1;
28      }
```

The function insertaccepts as input thecommitmentand returns theLeaf Index where it was inserted. Updating the tree consists of inserting commitment into the first available leaf, subsequently recalculating the intermediate MiMC hashes along the path from the leaf to the root.

The process uses a mechanism based oncurrentIndexAndfilledSubtrees. At every level, ifcurrentIndex iseven, the new hash is assigned to the left node and the right one is set tozeros[i],updatingfilledSubtrees[i] with the new value. If currentIndex isodd, the left node is recovered from filledSubtrees[i]and the new hash becomes the right one. This system allows you to keep track of the complete subtrees and efficiently update the tree.

The Merkle Tree has a depth of 20, allowing for a maximum of $2_{20} \approx 1.048.$ 576 leaves. The number of currently occupied leaves is indicated by the public variablenextIndex,which represents both the counter of deposits made and the index of the next available leaf. WhennextIndexreaches 2_{20}, the tree is full and further deposits are no longer possible. Currently, the average value ofnextIndex isaround 50,000, well below the maximum limit.

4.2.3 Execution example

Consider a 4-level Merkle Tree (depth 3, with 2_3=8 leaves). Let's briefly analyze 6 insertions, showing the state offilledSubtreesand the tree. Let's define:

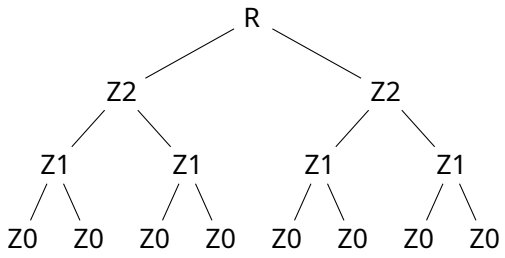
- Levels: 0 (leaves) to 3 (root).
- filledSubtrees:array of size 3 for levels 0, 1, 2. Stores the hash of the complete left subtree at each level, used to compute the higher nodes.

- Z_{the}=zeros[*the*]: default hashes for empty nodes.
- N_{the}: intermediate hashes (e.g.$N1$ = hashLeftRight($C0$, $C1$)).

Initial Construction

The Merkle Tree is initialized withlevels = 3.

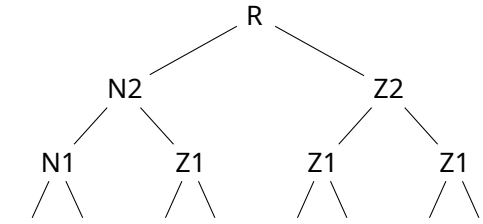
- nextIndex =0
- filledSubtrees = [Z_0, Z_1, Z_2], WhereZ_0=ZERO VALUE,Z_1=h(Z_0, Z_0), Z_2=h(Z_1, Z_1)
- Tree:



- Initial root:R=h(Z_2, Z_2)

Insert 1: Leaf$C0$ (index 0)

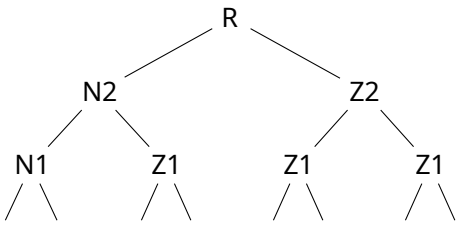
- nextIndex =1
- filledSubtrees = [$C0$, $N1$, $N2$]



- Tree: C0 Z0 Z0 Z0 Z0 Z0 Z0 Z0

Insert 2: Leaf $C1$ (index 1)

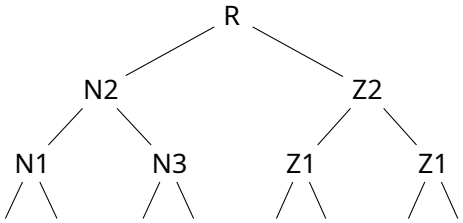
- nextIndex =2
- filledSubtrees = [$C0$, $N1$, $N2$]



- Tree: C0 C1 Z0 Z0 Z0 Z0 Z0 Z0

Insert 3: Leaf $C2$ (index 2)

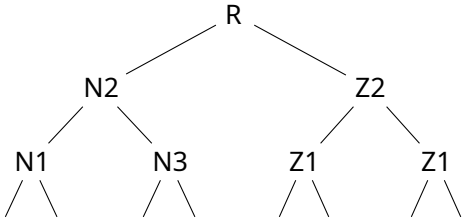
- nextIndex =3
- filledSubtrees = [$C2$, $N1$, $N2$]



- Tree: C0 C1 C2 Z0 Z0 Z0 Z0 Z0

Insert 4: Leaf $C3$ (index 3)

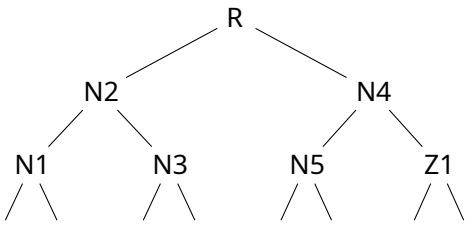
- nextIndex =4
- filledSubtrees = [$C2$, $N1$, $N2$]



- Shaft: C0 C1 C2 C3 Z0 Z0 Z0 Z0

Insert 5: Leaf $C4$ (index 4)

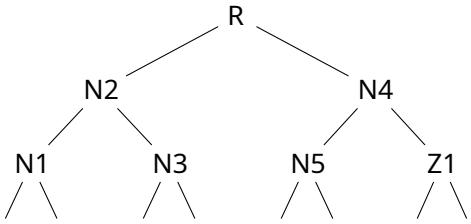
- nextIndex =5
- filledSubtrees = [$C4$, $N5$, $N2$]



- Tree: C0 C1 C2 C3 C4 Z0 Z0 Z0

Insert 6: Leaf $C5$ (index 5)

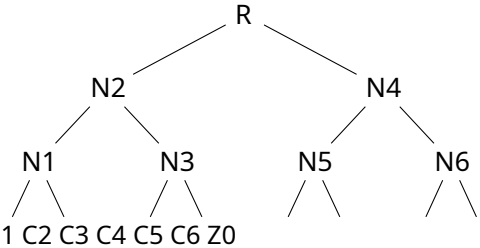
- nextIndex =6
- filledSubtrees = [$C4$, $N5$, $N2$]



- Tree: C0 C1 C2 C3 C4 C5 Z0 Z0

Insert 7: Leaf $C6$ (index 6)

- nextIndex =7
- filledSubtrees = [$C6$, $N5$, $N2$]



- Tree: C0 C1 C2 C3 C4 C5 C6 Z0

Where:

- $N1$ = hashLeftRight($C0$, $C1$),

- $N3$ = hashLeftRight($C2$, $C3$),
- $N5$ = hashLeftRight($C4$, $C5$),
- $N6$ = hashLeftRight($C6$, $Z0$),
- $N2$ = hashLeftRight($N1$, $N3$),
- $N4$ = hashLeftRight($N5$, $N6$),
- R=hashLeftRight($N2$, $N4$) (last state).

4.3 Generating the test

In this step, the key step of the entire protocol occurs: you have to demonstrate that you are aware of the pre-image of a commitment present in the Merkle Tree, without being able to reveal it, as this would lead to being able to calculate the commitment and therefore connect the deposit action with that of whitrdraw, thus completely eliminating all the work done up to now, in order to guarantee anonymity.

This step is done using the zk-SNARK algorithm.

Thewitness isthe set of data that is used to generate evidence that we are aware of private data, without revealing it.

This step cannot be done directly in the Tornado Cash smart contract, as the parameters passed as input must remain private. It is therefore performed by the user off-chain on their device (the Tornado Cash github repository provides tools to do this step from the command line or from a graphical interface, to make the user's work easier).

Let's analyze the CLI source code: https://github.com/tornadocash/tornado-cli/blob/master/cli.js

```
1    async function generateProof ({ deposit , currency , amount ,
     recipient , relayerAddress = 0, fee = 0, refund = 0 }) {

2   // Compute merkle proof of our commitment const { root ,
3   pathElements , pathIndices } = await
      generateMerkleProof(deposit , currency , amount);
4
5   // Prepare circuit input const
6   input = {
7       // Public snark inputs root:
8       root ,
9       nullifierHash: deposit.nullifierHash , recipient:
10      bigInt(recipient),
11      relayer: bigInt(relayerAddress), fee:
12      bigInt(fee),
13      refund: bigInt(refund),
```

```
14
15          // Private snark inputs nullifier:
16          deposit.nullifier , secret: deposit.secret ,
17
18          pathElements:         pathElements ,
19          pathIndices:          pathIndices
20      }
21
22      console.log('Generating SNARK                    proof ');
23      console.time('Proof time ');
24      const proofData = await websnarkUtils.
          genWitnessAndProve(groth16, input , circuit , proving_key);

25      const { proof } = websnarkUtils.toSolidityInput(
          proofData);
26      console.timeEnd('Proof time ');
27
28      const args = [
29          toHex(input.root),
30          toHex(input.nullifierHash),
31          toHex(input.recipient , 20),
32          toHex(input.relayer , 20),
33          toHex(input.fee),
34          toHex(input.refund)
35      ];
36
37      return     {    proof ,     args    };
38  }
```

The witness of the deposit is given by the following private data:

- secret
- nullifier
- pathElements:nodes belonging to the path from the leaf from the repository, to the Merkle Tree Root
- pathIndices:pathElements node indices

and from the following public data:

- root:any merkleRoot contained in the variableroots
- nullifierHash:the Pedersen Hash of the nullifier
- relayer:optional, the relayer address
- fee:optional, the fees to be given to the relayer

- refund:optional, the refund to be given to the relayer

The SNARK proof is generated by the librarywebsnarkthrough the function genWitnessAndProof.

4.4 Whitdrawal

```
        * *
        @dev Withdraw a deposit from the contract. `proof ` is a zkSNARK
        proof data, and input is an array of circuit public inputs

        `input` array consists of:
            - merkle root of all deposits in the contract
            - hash of unique deposit nullifier to prevent
      double spend
            - 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 already
      been spent");
        require(isKnownRoot(_root), "Cannot find your merkle root"); // Make
      sure to use a recent one require(verifier.verifyProof(_proof ,
        [uint256( _root), uint256(_nullifierHash), uint256(_recipient ) ,
      uint256(_relayer), _fee , _refund ]), "Invalid withdraw proof ");



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

        emit Withdrawal(_recipient , _relayer ,   _nullifierHash ,
        _fee);
    }
```

Once you have made a deposit, in order to withdraw it, you need to call the whitdraw function.

The data that a user must have in order to withdraw the funds are the public parameters used to generate the proof, and the proof itself.

The contract will carry out some control operations:

1. Relayer fees must not exceed the deposit amount (to avoid underflow).

2. The nullifierHash must not be present in the nullifierHashes dictionary, if it were present it would indicate that the amount has already been withdrawn (double spending).

3. The Merkle Tree Root is present in the root variable of the Merkle Tree.

4. Check that the proof is actually valid.

Once the conditions are valid, the hash of the nullifier is added to the nullifierHashes dictionary and the Withdrawal event is emitted.

Practical part

In this section we are going to analyze a transaction using Tornado Cash. We cannot do this, for legal reasons, with the original contract. We have deployed a copy of the contract on the Sepolia Base TestNet.

5.1 Sepolia Base Testnet

The Sepolia Base Testnet is a public testnet used to develop and test applications on the Base blockchain, a Layer 2 of Ethereum created by Coinbase[5]. This testnet provides a secure environment for testing smart contracts and dApps.

This network replicates the behavior of Ethereum, allowing developers to test their applications with confidence that the results will be consistent once they are deployed on the Ethereum mainnet.

The main difference is the consensus protocol. Since coins on Sepolia Base have no real value, there is no economic incentive to ensure the proper functioning of the network through the Proof of Stake mechanism. For this reason, validators are selected directly by Coinbase, making the network centralized. However, this is not a problem, since Sepolia Base is a test environment designed exclusively for development and experimentation.

To get Ether on Sepolia Base, you need to use faucets, services that distribute a small amount of ETH for free to requesting wallets. These test ETH are used to pay transaction fees, allowing developers to interact with the blockchain without real costs.

5.2 Poseidon Hash

The version of Tornado Cash we created has an improvement over the basic version: the use of Poseidon Hash instead of Pedersen Hash. This improvement was proposed by ABDK Consulting, when they asked to audit the Tornado Cash contract.

An audit is a verification of the security of a contract. A developer can request an audit from a company, which after studying the code guarantees its security, providing a certificate. This is a crucial step in blockchains, because in a decentralized environment if you interact with the wrong contract you risk losing money and there is no central entity that can cancel the transaction, so projects rely on these audits to demonstrate the security of their contracts.

5.2.1 Technical part about Poseidon Hash

Poseidon Hash is a cryptographic hashing function designed to provide high efficiency in zero-knowledge (ZK) circuits and zero-knowledge proofs (ZKPs), such as SNARKs and STARKs. It was introduced in 2019 in the paper *"Poseidon: A New Hash Function for Zero-Knowledge Proof Systems"*[14], as part of a family of hashes optimized for arithmetic circuits. In 2023, the same authors proposed an updated version, Poseidon2 [15], which further improves performance.

Poseidon Hash is built as a sponge function (*sponge function*) which uses the POSEIDON permutation π, a variant of the HadesMiMC strategy. Map strings to a finite field F_p (Where p And a prime number approximately $2_n > 2_{31}$) in fixed-length strings on the same field. The POSEIDON permutation π It is based on an SPN structure (*Substitution-Permutation Network*), combining layers of nonlinear S-boxes (e.g. x_α, with α = 5 or 3) and linear layers (MDS matrix multiplications). This structure is optimized for R1CS systems (*Rank-1 Constraint Systems*), commonly used in SNARKs, significantly reducing the number of constraints compared to other functions such as Pedersen Hash and MiMC.

The original paper highlights a key advantage:

> "Our POSEIDON hash function uses up to 8x fewer constraints per message bit than Pedersen Hash."

This efficiency is crucial for ZK applications, where the number of constraints determines the computational complexity of proof generation and verification.

Poseidon Hash is recommended for all applications that require ZK-compatible hashing functions, in particular:

1. Commitments: For protocols where knowledge of a committed value is demonstrated without revealing it. We recommend using Poseidon-128 with single-call permutations and widths (*width*) from 2 to 5 field elements. Compared to Pedersen Hash, Poseidon is faster and can also be used in signature schemes, reducing the code footprint.

2. Multiple Object Hashing: To encode multiple fields as field elements and demonstrate their properties in ZK. It is suggested to use a variable length sponge with Poseidon-128 or Poseidon-80 and width 5 (with *installments* 4).

3. Merkle Trees: To allow ZK proofs of knowledge of a leaf in a tree, with possible statements about the leaf's contents.
   An arity of 4 (width 5) is recommended with Poseidon-128 for best performance, although more conventional arities (e.g. 2 or 8) are supported.

In SNARKs, the prime field is often the scalar field of an elliptic curve *pairing friendly*. The POSEIDON permutation π can be represented as a circuit with a relatively low number of *gate*, but its parameters must be adapted to the value of p. In particular, after having fixed p, it occurs if x_α Andreversible in F_p, condition satisfied if $p \bmod \alpha \neq 1$.

Poseidon Hash stands out for its efficiency compared to Pedersen Hash and MiMC, two hashing functions used in ZK contexts. The following table shows the number of R1CS constraints needed to prove knowledge of a leaf in a Merkle tree with 2_{30} elements, comparing Poseidon-128 with Pedersen Hash and MiMC-2p/p (Feistel):

Poseidon-128				
Arity	Length	R_F	R_P	Total Constraints
2:1	3	8	57	7290
4:1	5	8	60	4500
8:1	9	8	63	4050
Pedersen Hash				
-	171	-	-	41400
MiMC-2p/p (Feistel)				
1:1	2	324	-	19440

Table 5.1: Comparison of R1CS constraints for a Merkle tree with 2_{30} elements.

- Poseidon-128: With arity 4:1 (width 5), it requires only 4500 total constraints, thanks to a small number of complete rounds (R_F=8) and partial (R_P=60).
- Pedersen Hash: It requires 41,400 constraints, a significantly higher value, due to its structure based on discrete logarithms, less suitable for arithmetic circuits.
- MiMC-2p/p: With 19,440 constraints, it is more efficient than Pedersen but less than Poseidon, due to the higher number of rounds (324) needed to ensure safety.

5.2.2 Implementation

To implement the contracts on Sepolia Base, the code provided by Chih Cheng Liang, a developer at the Ethereum Foundation, who made his version public (https://github.com/ChihChengLiang/poseidon-tornado)

Posiedon Hash is used instead of MiMC to hash leaves and instead of Pedersen Hash to make commitments.

	Classic	PoseidonHash
Deposit (Gas)	1088354	874131
Whitdraw (Gas)	301233	322005
Constraints	28271	5386

The commitment no longer uses the secret value but only the nullifier:
commitment = PoseidonHash(nullifier, 0)
To generate proof and witness we use the nullifierHash, calculated in the following way:
nullifierHash = PoseidonHash(nullifier, 1, leafIndex) The data provided by Chih Cheng Liang are as follows:
To compile the project you need to have the following programs installed:

- Node.js (not a recent version, for this project we used v16.20.2)
- Circom version 2.0.2

Once the repository has been downloaded, the commands to execute are the following:

```
npm-install
npm run build
npmtest
```

Once the build is successful, an artifact folder will be created containing the .json file of the compiled contracts.
One problem that was encountered when deploying contracts on testnet was that the Hasher.json contract had empty bytecode. So the contract was compiled usinggenContract.jsof circolibjs with the following code:

```
const path = require('path '); const fs =
require('fs ');
const centomlibjs = require('circomlibjs ');

const outputPath = path.resolve(__dirname , '../
    artifacts/contracts/Hasher.json ');

async function main() {
  console.log('Generating Poseidon hasher with
    circomlibjs ...');

  const poseidonContract = await centomlibjs.
    poseidonContract;
  const numInputs = 2;

  const abi = poseidonContract.generateABI(numInputs);
```

```
    const bytecode = poseidonContract.createCode(
      numInputs);

    console.log('ABI length:', abi.length); console.log('Bytecode length:',
    bytecode.length); console.log('Bytecode preview:', bytecode.slice(0, 1

      00) + '...');

    const contract = {
        contractName:       'Hasher',
        abi: abi ,
        bytecode:       bytecode ,
    };

    fs.writeFileSync(outputPath ,                   JSON.stringify(contract
      , null , 2));
    console.log('Hasher.json generated at:', outputPath)
      ;

    // Test with JS Poseidon
    const poseidon = await centomlibjs.buildPoseidon (); const testInputs =
    [1, 2];
    const jsHash = poseidon(testInputs); console.log('JS Poseidon hash:',
    poseidon.F.toString
      (jsHash));
}

main().catch(( error) => {
    console.error('Error:',                   error);
    process.exit(1);
});
```

Once you have Verifier.json, Hasher.json and ETHTornado.json, you need to deploy the contracts.

The contract that will be used to interact with for deposits and withdrawals will be the one generated by the deployment of ETHTornado.json. The way in which the three contracts are connected is done with the deployment of the latter, whose constructor takes as input the address of the two contracts (Hasher and Verifier).

The contracts were deployed using the web3 library, which allows us to connect to an rpc and send transactions to the network, using the following script:

```
const Web3 = require('web3'); const fs =
require('fs ');

// Connect to Base Sepolia
const web3 = new Web3('wss://base -sepolia -rpc.
```

```
    publicnode.com ');
6
7   // Account setup
8   const privateKey = 'REDACTED ';
9   const hasherAddress = '0x0ffc6149238E8c153e5ad30E71f37
        e50B9C87caC ';
10  const verifierAddress = '0x9A25Fb8207086912ee3af781277
        e14767f81BD71 ';
11  const account = web3.eth.accounts.privateKeyToAccount(
        privateKey);
12  web3.eth.accounts.wallet.add(account);
13
14  // Load contract data
15  const contractData = JSON.parse(fs.readFileSync ('../
        artifacts/contracts/ETHTornado.sol/ETHTornado.json ', 'utf8'));
16  const contractABI = contractData.abi;
17  const contractBytecode = contractData.bytecode;
18
19  const contract = new web3.eth.Contract(contractABI);
20
21  async function deployContract() {
22      try {
23          // Network info
24          const chainId = await web3.eth.getChainId ();
25          console.log('Chain ID:', chainId); //Should 84532
        well
26
27          // Account info
28          const balance = await
            web3.eth.getBalance( account.address);
29          console.log('Account:', account.address);
30          console.log('Balance:', balance , web3.utils.fromWei(
        'ether '), 'ETH ');
31          if (balance === '0 ') throw new Error('Account has no funds ');
32
33          // Check hasher address
34          const hasherCode = await
            web3.eth.getCode( hasherAddress);
35          console.log('Hasher code length:', hasherCode. length);
36          if (hasherCode === '0x') throw new Error(' Hasher address
        is not a deployed contract ');
37
38          // Estimate gas
```

```
39              const gasEstimate = await contract.deploy ({
40                  data: contractBytecode ,
41                  arguments: [verifierAddress , 1000000000000
    000,  20, hasherAddress]
42              }).estimateGas ({ from: account.address });
43              console.log('Estimated gas:', gasEstimate);
44
45              const gasPrice = await web3.eth.getGasPrice (); console.log('Gas
46              price:', web3.utils.fromWei( gasPrice , 'gwei '), 'gwei ');
47
48              // Deploy
49              const deployedContract = await contract
50                  . deploy({
51                      date:   contractBytecode ,
52                      arguments:   [verifierAddress ,   100000000
    0000000, 20, hasherAddress]
53                  })
54                  . send({
55                      from:   account.address ,
56                      gas:   Math.floor(gasEstimate),
57                      gasPrice:   gasPrice
58                  })
59                  . on('transactionHash ', (hash) =>
60                      { console.log('Tx hash:', hash);
61                  })
62                  . on('receipt ', (receipt) => { console.log('Tx
63                  receipt:',                          receipt);
64                  })
65                  . on('error ', (error) => { console.error('Tx
66                  error:',                          error);
67                  });
68
69              console.log('Deployed          at:',     deployedContract.
    options.address);
70              return deployedContract; } catch
71       (error) {
72              console.error('Deployment          failed:',     error.
                message);
73              if (error.data) console.error('Revert data:', error.data);
74              throw error;
75       }
76  }
77
78  deployContract();
```

The constructor of ETHTornado.json takes 4 parameters as input, in order:

- _verifyThe address to which the Verifier contract was deployed
- _denominationIt represents the fixed value of deposits and withdrawals, in our case it is 10000000000000000 wei (0.001ETH)
- _merkleTreeHeightThe height of the Merkle Tree, in our case 20
- _hasherThe address to which the Hasher contract was deployed

For the deployment of Hasher and Verifier the same script was used but with arguments the empty list ”[ ]” and with input Verifier.json and the Hasher.json generated with the previous script.

To check transactions and addresses on Sepolia Base you can use the following site:https://sepolia.basescan.org[2]

The contracts have been deployed to the following addresses:

- Hasher:0x0ffc6149238E8c153e5ad30E71f37e50B9C87caC
- Verifier:0x9A25Fb8207086912ee3af781277e14767f81BD71
- ETHTornado0x3a6d3B0F0bA61d2Ca2e56129E522982fFf0a545e

Let's now look at an example of a transaction:

5.2.3 Deposit

Script:

```
const { Web3 } = require('web3'); const centomlib =
require("circomlibjs"); const ethers = require(" ethers ");
const { BigNumber } = ethers;


// Configuration
const rpc = "wss://base -sepolia -rpc.publicnode.com"; const PRIVATE_KEY
= "REDACTED ";
const tornadoAddress = "0x3a6d3B0F0bA61d2Ca2e56129E522
    982fFf0a545e";

// Web3 initialization const web3 =
new Web3(rpc);
const contractJson = require(__dirname + '/../
    artifacts/contracts/ETHTornado.sol/ETHTornado.json ');

const tornado = new web3.eth.Contract(contractJson.abi
    , tornadoAddress);
```

```
15
16  // Account setup
17  const account = web3.eth.accounts.privateKeyToAccount
        ('0x' + PRIVATE_KEY);
18  web3.eth.accounts.wallet.add(account);
19  web3.eth.defaultAccount = account.address; const
20  senderAccount = account.address;
21
22  function toHex(number , length = 32) {
23          const str = number instanceof Buffer ? number. toString('hex ') :
        BigInt(number).toString(16); return '0x' + str.padStart(length * 2,
24          '0 ');
25  }
26
27  // Poseidon Hash function that returns a bytes32 function
28  poseidonHash(poseidon , inputs) {
29          const hash = poseidon(inputs.map((x) => BigNumber. from(x).toBigInt
            ()));
30          const hashStr = poseidon.F.toString(hash); // Convert to finite
            field
31          const hashHex = BigNumber.from(hashStr). toHexString(); //
        Convert to hexadecimal return ethers.utils.hexZeroPad(hashHex ,
32          32); // 32-byte padding

33  }
34
35  async function createDeposit(amount) {
36          // Initialize Poseidon
37          const poseidon = await centomlib.buildPoseidon ();
38
39          // Generate a random nullifier (15 bytes)
40          const nullifierBytes = ethers.utils.randomBytes(15
        ) ;
41          const nullifierHex = ethers.utils.hexlify( nullifierBytes);


42
43          // Calculate commitment with [nullifier , 0] const
44          commitment = poseidonHash(poseidon , [ nullifierHex , 0]);


45
46          // Return in JSON format return {
47
48                  nullifier: ethers.utils.hexZeroPad( nullifierHex , 32), //
        Nullifier as bytes32
49                  commitment: commitment
50          };
```

```
}

/**
  * Perform        a     deposit
  * @param {Object} params - Repository parameters
  * @param {string} params.currency - Currency (e.g. "eth ")
  * @param {number} params.amount - Deposit amount
  * /
async function deposit ({ currency , amount }) {
        try    {
                // Check the balance
                const balance = await
                web3.eth.getBalance( senderAccount);
                console.log('Account balance:', web3.utils.
      fromWei(balance , 'ether '), 'ETH ');

                // Check that the contract exists const code = await
                web3.eth.getCode( tornadoAddress);

                if (code === '0x') {
                        throw new Error('No contract
      deployed to the specified address !');
                }

                const deposit = await createDeposit ({ amount
      });

                // Calculate the value to send (amount in wei
      )
                const value = web3.utils.toWei(amount.toString 'ether ');
      (),
                console.log('Value to send:', web3.utils. fromWei(value , 'ether
      '), 'ETH ');

                const commitment = deposit.commitment

                // Estimate the gas
                const gasEstimate = await tornado.methods.
      deposit(commitment).estimateGas ({
                        from:     senderAccount ,
                        value:      value
                });
                console.log('Gas            estimated:', gasEstimate);

```

```
86              // Get the gas price
87              const gasPrice = await web3.eth.getGasPrice (); console.log('Gas
88              Price:', web3.utils. fromWei(gasPrice , 'gwei '), 'gwei ');
89
90              // Verify sufficient funds
91              const totalCost = BigInt(Number(gasEstimate))
        * BigInt(gasPrice) + BigInt(value); if (BigInt(balance) <
92              totalCost) {
93                      throw new Error('Insufficient funds for
        the deposit and the gas!');
94              }
95
96              // Execute the transaction
97              console.log('Submitting deposit transaction ');
98              const receipt = await
                tornado.methods.deposit( commitment).send({
99                      from:    senderAccount ,
100                     value:     value ,
101                     gas: gasEstimate ,
102                     gasPrice: gasPrice
103             });
104
105             console.log('Transaction completed! Hash:',
                receipt.transactionHash);
106             const data = receipt.logs[0].data; const firstHalf = '0x' +
107             data.slice(2, 66); const leafIndex = parseInt(firstHalf);
108             console.log('leafIndex:', leafIndex); const poseidon = await
109             centomlib.buildPoseidon
110
        ();
111             const nullifierHash = poseidonHash(poseidon ,
        [ deposit.nullifier , 1, leafIndex ]);
112             const nullifierHex = deposit.nullifier; return {
113                     nullifierHex , commitment , };
        nullifierHash
114       } catch (error) {
115             console.error('Error while depositing:', error);
116             if (error.receipt) {
117                     console.error('Receipt Details:', error.
        receipt);
118             }
119             throw error;
120       }
121 }
```

```
122
123 // Perform the deposit
124 deposit ({ currency: "eth", amount: 0.001 })
125         . then(result => { console.log('nullifier:',
126                                                     result.nullifierHex)
        ;
127                 console.log('commitment:',          result.commitment);
128                 console.log('nullifierHash:',             result.
                    nullifierHash);
129          })
130         . catch(err => console.error('Error:', err));
```

Output:

```
Account balance: 0.083258091675057736 ETH

Value to send: 0.001 ETH

Estimated gas: 828077n

Gas price: 0.001000319 gwei

Submitting deposit transaction

Transaction completed!

Hash:
↪ 0xf486e0c92708ef5794d851bc6698cafb7f40ec338b72907f053da7d1d8fe82d1

leafIndex: 11

nullifier:
↪ 0x00000000000000000000000000000001a34f8546195f074f9e7380446ff3a

commitment:
↪ 0x0b6014c902cd1e73dde4d951cc276fe7ee574fd39784a75a440911ae61c36f22

nullifierHash:
↪ 0x26f3b5d6175b61acd648a7c095b0c480eebe60297aab12b08980a5265da467b3
```

5.2.4 Test Generation

To write this script it was necessary to compile the file /src/MerkleTree.ts
in a javascript file using the commandnpx tsc.Getting the file /dist/src/merkleTree.js

```
1 const { Web3 } = require('web3'); const assert =
2 require('assert ');
```

```
const centomlib = require('circomlibjs '); const
{ BigNumber } = require('ethers '); const path =
require('path ');
const { groth16 } = require('snarkjs ');
const { MerkleTree } = require ('../ dist/src/merkleTree
    ');

const rpc = "wss://base -sepolia -rpc.publicnode.com"; const contractJson
= require(__dirname + '/../
    artifacts/contracts/ETHTornado.sol/ETHTornado.json ');

const tornadoAddress = "0x3a6d3B0F0bA61d2Ca2e56129E522
    982fFf0a545e";

const web3 = new Web3(rpc);
const tornado = new web3.eth.Contract(contractJson.abi
    , tornadoAddress);

const MERKLE_TREE_HEIGHT = 20; const
ETH_AMOUNT = 1e18;
const TOKEN_AMOUNT = 1e19;

// Utility functions
function toHex(number , length = 32) {
    const str = number instanceof Buffer ? number.
      toString('hex ') : BigInt(number).toString(16); return '0x' +
    str.padStart(length * 2, '0 ');
}

function poseidonHash(poseidon , inputs) {
    const hash = poseidon(inputs.map((x) => BigNumber.
      from(x).toBigInt());
    const hashStr = poseidon.F.toString(hash); const hashHex =
    BigNumber.from(hashStr).toHexString
      ();
    return web3.utils.padLeft(hashHex , 64);
}

// Poseidon Hasher
class PoseidonHasher            {
    constructor(poseidon)           {
        this.poseidon = poseidon;
    }

    hash(left, right) {
        return poseidonHash(this.poseidon ,                 [left ,       right]);
```

```
    }
}

// Fetch events in batches
async function getEventsInBatches(fromBlock ,                          toBlock ,
      batchSize = 49999n) { const
    events = [];
    const latestBlock = toBlock === 'latest ' ? await web
      3.eth.getBlockNumber () : BigInt(toBlock); for (let start =
    BigInt(fromBlock); start <=
      latestBlock; start += batchSize) {
        const end = latestBlock < start + batchSize - 1n ? latestBlock : start +
         batchSize - 1n;
        console.log(`Fetching events from block ${start} to ${end}`);

        const batch = await tornado.getPastEvents('Deposit ',
            {
            fromBlock: start.toString (), toBlock:
            end.toString (), });

        events.push(... batch);
    }
    return events;
}

// Generate the Merkle proof
async function generateMerkleProof(deposit) {
    console.log('Getting current state from tornado
      contract ');
    const events = await getEventsInBatches(22237544, '
      latest ');

    const leaves = events
        . sort((a, b) => {
            const indexA = BigInt(a.returnValues.leafIndex); const indexB =
            BigInt(b.returnValues.leafIndex); return indexA < indexB ? -1 :
            indexA > indexB ? 1 : 0;

        })
        . map(e => e.returnValues.commitment);

    if    (! leaves.includes(deposit.commitment)) { throw new
        Error(`Deposit commitment ${deposit. commitment} not found
      in the fetched leaves `);
    }

```

```
    const poseidon = await centomlib.buildPoseidon (); const tree = new
    MerkleTree(MERKLE_TREE_HEIGHT , '
      tornado ', new PoseidonHasher(poseidon));

    console.log('Inserting leaves into Merkle Tree ...'); for (const leaf of
    leaves) {
        await tree.insert(leaf);
    }
    console.log('All leaves inserted ');

    const leafIndex = leaves.indexOf(deposit.commitment)
      ;
    console.log('Adjusted leafIndex:', leafIndex); assert(leafIndex >= 0, 'The
    deposit is not found in
      the tree ');

    const root = await tree.root(); const rootHex
    = toHex(root);
    const isValidRoot = await tornado.methods.
      isKnownRoot(rootHex).call();
    const isSpent = await tornado.methods.isSpent(
      deposit.nullifierHash).call();

    console.log('Root:', rootHex); console.log('Is valid root:',
    isValidRoot); console.log('Is spent:', isSpent);


    assert(isValidRoot === true , 'Merkle tree is
      corrupted ');
    assert(isSpent === false , 'The note is already spent
      ');

    const { path_elements: pathElements, path_index:
      pathIndices } = await tree.path(leafIndex);

    return    {   root ,    pathElements ,       pathIndices       };
}

// Generate the SNARK test async function
proof(witness) {
    const wasmPath = path.join(__dirname , '../ build/
      withdraw_js/withdraw.wasm ');
    const zkeyPath = path.join(__dirname , '../ build/
      circuit_final.zkey ');

    const { proof } = await groth16.fullProve(witness ,
```

```
      wasmPath , zkeyPath);
    const solProof = {
      to:  [proof.pi_a[0],            proof.pi_a[1]],
      b:   [
        [proof.pi_b[0][1],            proof.pi_b[0][0]],
        [proof.pi_b[1][1],            proof.pi_b[1][0]],
      ],
      c:   [proof.pi_c[0], proof.pi_c[1]],
    };
    return     solProof;
}

// Generate the complete proof for withdrawal async function
generateProof({ deposit , recipient ,
      relayer = '0x0000000000000000000000000000000000000000 00', fee = '0',
      refund = '0' }) {
    const { root , pathElements , pathIndices } = await
      generateMerkleProof(deposit);

    const witness = {
      root ,
      nullifierHash:          deposit.nullifierHash ,
      container ,
      relayer ,
      fee ,
      nullifier: BigNumber.from(deposit.nullifier). toBigInt(),

      pathElements ,
      pathIndices ,
    };

    const solProof = await proof(witness); return
    solProof;
}

// Input data const
deposit = {
    "nullifier ": "0x00000000000000000000000000000001a
      34f8546195f074f9e7380446ff3a",
    "commitment ": "0x0b6014c902cd1e73dde4d951cc276fe7ee5
      74fd39784a75a440911ae61c36f22",
    "nullifierHash ": "0x26f3b5d6175b61acd648a7c095b0c480
      eebe60297aab12b08980a5265da467b3"
};

const recipient = "0x58102161341811da3009e257618b1cc5c
```

```
    2c464c0";

// Execution
(async() => {
    try {
        const solProof = await generateProof ({ deposit , recipient });

        console.log('Final proof:', JSON.stringify( solProof , null , 2));

    } catch (error) {
        console.error('Error:', error);
    }
})();
```

Output:

```
Getting current state from tornado contract

Fetching events from block 22237544 to 22275523

Inserting leaves into Merkle Tree...

All leaves inserted

Adjusted leafIndex: 11

Root:
↪ 0x16216c3050e4c744e0a057779a4cc933999830a439ed091e2f518e2c43b9bb62

Is root valid: true

Is spent: false

Final proof: {
  "a": [
    "78624409140957194447513430300166747697270951452662454921821120956↩
     ↪ 17319191350",
    "20399187910052134371486872584403723224813942216176337451136616361↩
     ↪ 980246752812"
  ],
  "b": [
    [
      "38575358269974367590073065868762192776985744476204708466163131 3↩
       ↪ 811160781351",
      "13600705585898368067493640181325805925936606313956877656533809 3↩
       ↪ 86812346179126"
    ],
    [
```

```
        "28629139911293392109411060880132780576066973993061956968859808
        ↪ 0905916213309",
        "62954017402187977669554492444775786090699791594120230072381819
        ↪ 5039656728935"
      ]
    ],
    "c": [
      "25141816379742662381752235881846958302871113374344911115517075703
      ↪ 90284323099",
      "35363928920732017262383655707842285304229841231935468820674351644
      ↪ 86998942928"
    ]
}
```


5.2.5 Withdraw

Script:

```
const { Web3 } = require('web3');

// Configuration
const rpc = "wss://base -sepolia -rpc.publicnode.com"; const PRIVATE_KEY
= "REDACTED ";
const tornadoAddress = "0x3a6d3B0F0bA61d2Ca2e56129E522
    982fFf0a545e";

// Web3 initialization const web3 =
new Web3(rpc);
const contractJson = require(__dirname + '/../
    artifacts/contracts/ETHTornado.sol/ETHTornado.json ');

const tornado = new web3.eth.Contract(contractJson.abi
    , tornadoAddress);

// Account setup
const account = web3.eth.accounts.privateKeyToAccount
    ('0x' + PRIVATE_KEY);
web3.eth.accounts.wallet.add(account);
web3.eth.defaultAccount = account.address; const
senderAccount = account.address;

/**
  * Make an ETH withdrawal
  * @param {Object} params - Pickup parameters
  * @param {string} params.proof - Hexadecimal string of the zk-SNARK
  proof
```

```
* @param {Array} params.args - Array of additional parameters [root ,
    nullifierHash , recipient , relayer , fee , refund]

* @param {string} [params.refund='0 '] - Refund amount in
                   wee
* /
async function      withdraw(proof , root , nullifierHash ,
    recipient , relayer , fee = '0', refund = '0 ') { try
        {
        // Check the balance
        const balance = await
        web3.eth.getBalance( senderAccount);
        console.log('Account balance:', web3.utils.
    fromWei(balance , 'ether '), 'ETH ');

        // Check that the contract exists const code = await
        web3.eth.getCode( tornadoAddress);

        if (code === '0x') {
            throw new Error('No contract
    deployed to the specified address !');
        }

        const isKnownRoot = await tornado.methods.
        isKnownRoot(root).call();
        console.log('Root         known:', isKnownRoot); {
        if (! isKnownRoot)
            console.warn('Warning: The root provided
    it is not present in the contract!');
        }

        const isSpent = await
        tornado.methods.isSpent( nullifierHash).call();
        console.log('Nullifier already spent:', isSpent); if (isSpent) {

            throw new Error('The nullifier has already been
    spent!');
        }

        // Estimate the gas
        const gasEstimate = await tornado.methods.
    withdraw(proof , root , nullifierHash , relayer , fee).estimateGas ({

            from:    senderAccount ,
            value:     refund
        });
```

```
55              console.log('Estimated gas:', gasEstimate);
56
57              // Get the gas price
58              const gasPrice = await web3.eth.getGasPrice (); console.log('Gas
59              Price:', web3.utils. fromWei(gasPrice , 'gwei '), 'gwei ');
60
61              // Verify sufficient funds
62              const totalCost = BigInt(Number(gasEstimate))
    * BigInt(gasPrice) + BigInt(refund); if (BigInt(balance) <
63              totalCost) {
64                      throw new Error('Insufficient funds for
    the deposit and the gas!');
65              }
66
67              // Execute the transaction
68              console.log('Submitting                withdraw     transactions ')
    ;
69              const receipt = await tornado.methods.withdraw (proof , root ,
                        nullifierHash , recipient , relayer ,
    fee).send({
70                      from:     senderAccount ,
71                      value:      refund ,
72                      gas: gasEstimate ,
73                      gasPrice: gasPrice
74              })
75                      . on('transactionHash ', (txHash) => { console.log(`The
76                              transaction hash is $
    {txHash}`);
77                      })
78                      . on('error ', (e) => { console.error('Error while
79
    transaction:', e.message);
80                      });
81
82              console.log('Withdrawal           completed !');
83       } catch (error) {
84              console.error('Error while withdrawing:', error); if (error.receipt)
85              {
86                      console.error('Receipt Details:', error.
    receipt);
87              }
88              throw error;
89       }
90 }
91
```

```
92
93
94  const proof = {
95          "a": [
96              "786244091409571944475134303001667476972709514526624549218211209561731919135
                0",
97              "203991879100521343714868725844037232248139422161763374511366163619802467528
                12"
98          ],
99          "b": [
100             [
101                 "385753582699743675900730658687621927769857444762047084661631313811160781
                    351",
102                 "136007055858983680674936401813258059259366063139568776565338093868123461
                    79126"
103             ],
104             [
105                 "286291399112933921094110608801327805760669739930619569688598080090591621
                    3309",
106                 "629540174021879776695544924447757860906997915941202300723818192503965672
                    8935"
107             ]
108         ],
109         "c": [
110             "251418163797426623817522358818469583028711133743449111155170757039028432309
                9",
111             "353639289207320172623836557078422853042298412319354688206743516448699894292
                8"
112         ]
113 }
114
115 const root = "0x16216c3050e4c744e0a057779a4cc933999830
        a439ed091e2f518e2c43b9bb62";
116
117 const nullifierHash = "0x26f3b5d6175b61acd648a7c095b0c
        480eebe60297aab12b08980a5265da467b3"
118
119 const recipient = "0x58102161341811da3009e257618b1cc5c
        2c464c0";
120
121 const relayer = "0x0000000000000000000000000000000000
        00000";
122
123 // Perform withdrawal
124 withdraw(proof , root ,          nullifierHash ,          container ,
        relayer)
```

```
125 . then (() => console.log('Withdrawal successful'))
126 . catch(err => console.error('Final error:', err));
```

Output:

```
Account balance: 0.007567958903663304 ETH
Known Root: true
Nullifier already spent: false
Estimated gas: 315590n
Gas price: 0.00100032 gwei
Submitting withdraw transaction
The transaction hash is
0xef290555754da1fe69539bbc2c66f1d2ab1b080db26ad045c7a40b387a8ed997
Withdrawal completed!
Withdrawal successful
```

5.2.6 Comments

Deposit address: 0x75D610b953C60f9950be94a5cE82C8DC8bA41CC2

Address that made the withdrawal:

0x58102161341811dA3009e257618b1CC5C2c464C0 In the variablePRIVATE_KEYthe respective private keys were then entered.

A relayer was not used but the withdrawal was collected directly from the receiving address

Transactions on Sepolia Test can be verified at the following links:

Deposit: 0xf486e0c92708ef5794d851bc6698cafb7f40ec338b72907f053da7d1d8fe82d1

Withdraw: 0xef290555754da1fe69539bbc2c66f1d2ab1b080db26ad045c7a40b387a8ed997

Additions to the Basic Protocol

6.1 Relays

When a user makes a withdrawal on Tornado Cash, they must interact directly with the protocol's smart contract, which incurs transaction costs in terms ofgas fees.This is often a problem, as Tornado Cash's primary goal is to ensure transaction anonymity. As a result, the accounts receiving the funds are often empty of Ether (ETH), as purchasing ETH through an exchange and then transferring it to a new address could result in the user being identified.

To solve this problem, therelayer,third parties who act as intermediaries and offer to cover gas fees in exchange for a small fee (which varies, e.g. from 0.45% to 2.5% of the withdrawal, currently).

When a user wants to make awithdrawalusing a relayer, generate acryptographic proof (proof)which, in addition to including the deposit data, also incorporates the address of the relayer itself. Instead of sending the proof directly to the smart contract, the user transmits it to the relayer*off chain*, who is responsible for signing the transaction and covering the costs of its execution.

Thanks to the fact that the proof of thewithdrawalis generated by specifyingboth the relayer address and the recipient wallet address,The relayer has no ability to alter the behavior of the transaction to its advantage. The only action it can take is not to execute the transaction. In that case, the user can simply select another relayer and generate a new proof using the new intermediary's data.

A Relayer can become one by depositing a minimum of 5k TORN in stake. The more TORN he stakes, the higher his “score”, the higher his fee, the lower his score.

6.2 Anonymity Mining

The effectiveness of the Tornado Cash protocol depends heavily on the size of its anonymity set,that is, the number of users participating in the system.

If the protocol were used by a single individual, the anonymization mechanism would be completely useless. In that case, it would be enough to look at the transactions recorded on the blockchain to identify an address that made a deposit and one that made a withdrawal, making the association between the two operations trivial.

Even with a small number of users, it would be possible to conduct a probabilistic analysis to establish correlations between deposits and withdrawals. If the anonymity set is limited, the number of potential links between deposits and withdrawals is reduced, allowing an observer to infer, with some certainty, which transactions are related to each other.

Another critical factor is the liquidity within the pool. Even with a large number of users, if depositors withdrew assets immediately after depositing, the balance would fluctuate between zero and the most recent amount, rapidly resetting after each withdrawal. This would make it easier to link transactions, as observing the balance changes would allow one to deduce when a deposit is redeemed.

To mitigate these problems, the mechanism ofAnonymity Mining,whose operation is quite simple: a user who deposits assets into the pool receives a reward in TORN tokens proportional to the time the funds remain in the protocol. This reward, calculated through Anonymity Points (AP) earned per block based on the size of the deposit (e.g. 4 AP for 0.1 ETH or 400 AP for 100 ETH), serves to increase the anonymity set, more effectively mask temporal patterns and make it significantly more difficult to correlate deposits and withdrawals, thus incentivizing broader and more lasting participation.

A key aspect of the protocol is that it is not publicly visible whether a given deposit has been withdrawn or not. This ensures that no direct link can be established between a deposit and a withdrawal, strengthening the level of anonymity of the entire system.

The Anonymity Mining program ended in 2021, as Tornado Cash already had enough volume.

6.3 Tornado Cash Nova

Suppose a user deposits 0.423489389 ETH, the probability that another user deposits the same amount, choosing randomly, is practically zero. And therefore from a withdrawal it would be possible to trace the deposit simply from the value. For this reason it was chosen to use standard amounts for deposits and withdrawals, so as to eliminate this type of vulnerability. As for ETH, the standard values are 0.1, 1, 10, 100.

However, this solution introduces some rigidity, since now to deposit the hypothetical 0.423489389 ETH we would need to make 4 deposit transactions and 4 withdrawal transactions, paying the gas 8 times instead of 2 and leaving 0.023489389 ETH on the starting wallet.

Tornado Cash Novawas born with the idea of solving this problem.

To overcome this limitation, Tornado Cash Nova introduces the concept of shielded accountand the ability to deposit and withdraw completely arbitrary amounts. Unlike Tornado Cash Classic, where the amounts are fixed, Nova allows users to depositany amount you wantand make multiple withdrawals until the balance is exhausted.

This system offers more flexibility, but requires special care to maintain anonymity. If a user deposits and withdraws the exact same amount, they may be more easily traceable. However, splitting the withdrawal into multiple transactions with different amounts can make it more difficult to link the deposit and withdrawals, especially in a widely used pool.

From a technical point of view, Tornado Cash Nova uses a system of upgradable commitmentbased on Merkle Trees and zk-SNARKs. Each transaction modifies the user's balance state without explicitly revealing the amounts involved. This is done by generating a newcommitment which represents the remaining balance after a partial withdrawal. The new commitment is added to the Merkle tree, while the old one is excluded from future verifications via thenullifier,preventing possible double spending.

Tornado Cash Nova is not an upgrade of Tornado Cash Classic, but an alternative version with different features. While Nova allows for more flexibility in amounts, Classic provides easier anonymization thanks to standardized amounts. The choice between the two depends on the user's specific needs in terms of privacy and usability.

6.4 Tornado Proxy

When accessing Tornado.cash deposit contracts through the official interface, all transactions are executed through a proxy contract called Tornado Proxy. Since deposit contracts are immutable and many Tornado.cash features were added only after their initial deployment, Tornado Proxy allows for additional functionality to be integrated without having to replace the original, tried and tested instances of deposit contracts.

The two most notable features of Tornado Proxy are:

- On-chain backup of user deposits, through encrypted note accounts, thus ensuring greater security and recoverability of transactions.

- Queue management of deposits and withdrawals, facilitating their subsequent processing within the Tornado Trees contract for efficient verification via zero-knowledge proofs.

When you make a deposit through the Tornado Proxy, and subsequently a withdrawal through it, the proxy calls the corresponding methods in the Tornado Trees contract.

Registering a deposit involves the following steps:

1. The address of the deposit contract, the commitment and the current block number are acquired.

2. These values are ABI encoded and hashed keccak256.

3. The resulting hash is fed into a queue within the contract, to be subsequently grouped into a Merkle Tree of deposits (not to be confused with the deposit contract itself).

Registering a withdrawal follows the same process, except that instead of the commitment, the hash of the nullifier associated with the withdrawal is used. The resulting keccak256 hash is then placed in the withdrawal queue, to be subsequently grouped into the withdrawal Merkle Tree.

The registerDeposit and registerWithdrawal methods of the Tornado Trees contract emit the DepositData and WithdrawalData events, respectively, containing the same values used to compute the hash, plus an additional field indicating the order of entry into the queue.

6.4.1 Chunked Merkle Tree Update

Standard Merkle Trees are expensive to store and update, especially when managing a large number of leaves. Depositing a note into Tornado.cash's deposit contracts can cost over 1.2 million gas, translating into hundreds of dollars in ETH if the transaction occurs on the Ethereum mainnet. Much of this cost comes from placing a single commitment into the deposit contract's Merkle Tree.

Instead of spending all that gas, one could simply propose a new Merkle root, computed off-chain, and prove its validity via a Zero Knowledge Proof.

However, verifying Zero Knowledge Proofs is expensive. To reduce costs, instead of updating the Merkle Tree with each change, multiple insertions can be grouped into aggregate commitments that can be verified in a single operation.

Chunked Tree Structure

The store and withdraw trees are both fixed-size Merkle trees, 20 levels deep, with one special feature: the chunk size determines the level at which updates are processed in aggregate, rather than as individual inserts.

In the case of Tornado Trees, the chunk size is 256 (2_8), so each chunk covers 8 levels of the tree. The complete tree remains limited to 2_{20}leaves, but these are divided into chunks of 256 leaves each, for a total of 2^{12}chunk.

The hash function used to generate the node labels is Poseidon, which shares some similarities with the Pedersen hash used in the main escrow contract, as it is based on a curve hashing algorithm.

elliptical. The main difference is that Poseidon operates on the BN128 curve, while Pedersen uses Baby Jubjub. Additionally, in zero-knowledge proofs, Pedersen requires about 1.7 constraints per bit, while Poseidon uses between 0.2 and 0.45 constraints per bit, making it significantly more efficient.

Events

To compute a tree update, you need to know its existing structure. To get it, you can query the contract logs and retrieve previously emitted DepositData or WithdrawalData events, depending on the tree you are updating. Using the index indicated by lastProcessedDepositLeaf or lastProcessedWithdrawalLeaf, you can split the events into two sets of leaves: “committed” and “pending”. As the names suggest, the leaves of the first set have already been committed into the Merkle Tree, while the leaves of the second set have yet to be committed.

Tree Update

Using the already confirmed events, it is possible to reconstruct the current state of the Merkle Tree by calculating the Poseidon hash for each of the existing leaves. This is done by taking as input the address of the Tornado instance, the hash of the commitment or nullifier, and the block number.

The initial state of the Merkle Tree is that each leaf node is labeled with a “zero value” equal to keccak256(“tornado”) % BN254 FIELD SIZE, similar to the behavior of zero nodes in the main storage loop, but using the BN254 elliptic curve. This approach ensures that all paths in the tree are invalid until a valid commit is inserted into a leaf, and provides a constant and predictable label for each node whose children are zero.

The leaves of the tree are then populated from left to right with the leaf hashes corresponding to the confirmed events. Afterwards, the non-leaf nodes are updated up to the root. If everything was done correctly, the resulting root should match the one currently stored as depositRoot or withdrawalRoot in the Tornado Trees contract. Now that we have the “old root”, we can select a chunk of pending events (256 elements), compute its Poseidon hashes, and insert them into the tree. After updating the non-leaf nodes up to the root, we will have the “new root”.

Finally, you need to collect a list of path elements from the leaves of the subtree, along with an array of 0/1 values indicating whether each path element is to the left or right of its parent node.

Argument hash

Finally, in order to calculate a proof, it is necessary to hash the list of arguments that will be passed into the circuit to obtain the proof.

To build the message, you need to concatenate the following fields:

1. The old root label
2. The label of the new root
3. Path indices as integer, zero-padded on the left
4. For each event waiting to be entered:
    - The hash of the commitment/nullifier
    - The address of the Tornado instance (relative to the pool)
    - The block number

Once the message is created, the SHA-256 hash of it is calculated and the modulus of the hash is applied against the modulus of the BN128 group, which is found in the SNARK FIELD constant of the Tornado Trees contract.

Input for a Tree Update witness

The Batch Tree Update circuit takes as input the newly computed SHA-256 hash and the following private parameters:

1. The old root
2. The new root
3. Path indices as integer, zero-padded on the left
4. The elements of the path
5. For each event waiting to be entered:
    - The hash of the commitment/nullifier
    - The address of the Tornado instance (relative to the pool)
    - The block number

Prove Claim

To prove that the tree update was successful, it is not necessary to provide a proof that covers the entire structure. It is sufficient to show that a subtree of size equal to an 8-level block, containing a specific list of leaves, was inserted to replace the leftmost “leaf zero” of a 12-level tree.

Proof of Argument Hash

Instead of specifying all the inputs publicly, we can take the previously computed Args Hash and compare it with the result of computing the same hash inside the ZK circuit, using the private inputs. This makes the proof verification run much more efficient.

Subtree Construction

Taking the three fields of each pending event in order (instance, commit/nullifier, block number), we compute the Poseidon hash of each leaf. We then build the Merkle Tree for just the subtree we are updating. Since the subtree is complete, we don't need to worry about zero leaves.

Subtree Insertion Check

Finally, it is verified that inserting the root of the subtree into the proposed position results in the transformation of the old root into the new root. This process works essentially the same as the Merkle Tree Check in the main repository circuit, except that here Poseidon is used instead of MiMC.

The Merkle Tree Updater first checks whether the specified path contains a zero leaf by calculating what the root would be given the path elements, the path indices, and a zero leaf. It compares this result to the old specified root, and then repeats the process with the given subtree leaf, comparing the resulting root to the new root.

6.4.2 Completing Tree Update

After generating witness and test, the corresponding updateDepositTree or updateWithdrawalTree method in the Tornado Trees contract is called.

We move on to the update method:

- the proof
- the hash of the arguments
- the old root
- the new root
- the path indexes
- a list of events to be batched

The update method then checks that:

1. The specified old root is actually the current root of the corresponding tree.
2. The specified Merkle path points to the first available zero leaf in the subtree.
3. The specified argument hash matches the supplied inputs and proposed events.

4. The proof is valid according to the circuit verifier, using the hash of the arguments as public input.

If these conditions are met, each inserted event is deleted from the queue. The corresponding fields in the contract, indicating the current and previous roots, are updated, as is a pointer to the last leaf of the event.

There is a special branch of logic in this method that also handles event migration from the Tornado Trees V1 contract. After the initial migration, these paths are never visited again and can be safely ignored.

6.4.3 Tornado Router

The TornadoProxy contract 0x722122df12d4e14e13ac3b6895a86e84145b6967

It was then replaced by the TornadoRouter contract 0xd90e2f925da726b50c4ed8d0fb90ad053324f31b thus replacing the entry point to access the protocol.

This update enabled the introduction of Tornado Nova and brought some other minor improvements across the board.

DAO

The Tornado Cash ecosystem includes aERC-20 tokens, TORN,which performs a function ofgovernance.It gives holders the right to propose and vote on changes regarding the development of the protocol through the DAO (Decentralized Autonomous Organization).

The initial distribution of the TORN token follows the following pattern:

- 5% (500,000 RUB):distributed viaairdropto early adopters who used Tornado.Cash ETH pools, as a form of recognition for their early support of the protocol.
- 10% (1,000,000 ROUND):assigned to the program of “anonymity mining”to incentivize users to keep funds in the pools, thus increasing the anonymity set. These tokens are distributed linearly over a year and, once they run out, Tornado Cash is expected to have reached a critical mass of users such that it no longer needs this incentive. The anonymity mining program has in fact ended in 2021.
- 55% (5,500,000 RUB):reserved for theDAO treasure,with linear release over five years (after an initial three-month “cliff” period, during which the funds remain locked). These funds are managed through votes of the DAO, which can decide to use them for future developments, security audits, incentives for the growth of the protocol or other.
- 30% (3,000,000 ROUND):intended for developersfoundersand to the first supporters of the project, with a linear release over three years (after a one-year “cliff” period, during which the tokens cannot yet be redeemed).

7.1 How to submit a proposal?

In order to participate in the governance of Tornado.Cash, a user must first put his TORN instake.If a user votes or creates a proposal, the tokens

cannot be unlocked until the end of the process, which lasts8.25 days from the creation of the proposal. To create a proposal, a user must have at least 1.000 TORN (currently there are around 6500€).Each proposal must be a smart contract whose code has been verified and which is executed by the governance smart contract via the functiondelegatecall.The period for voting on a proposal is5 days.A proposal will be successful if it is approved by the majority and if at least25,000 votes (1 TORN = 1 vote). After a proposal is approved, it is subject to a period of timelockof 2 days. After this period, any user can execute the proposal. However, if the proposal is not executed by any user within 3 days after the timelock expires, it is considered expired and can no longer be executed.

A proposal may concern the following operations:

- The addition of a new pool (where Tornado Cash users deposit or withdraw funds) in theproxy,which is a central smart contract that stores pool addresses in a mutable data structure, allowing new pools to be added through governance.
- The modification of the parameters relating to the interest ratesreward.
- Token activation or deactivationRETURN.
- The modification of some major mining contracts, such as the TornadoTrees contract.
- A combination of all the above operations.

7.2 Staking

With the approval of the 10th governance proposal, the TORN token has acquired a new functionality related to thestaking,expanding its utility in the Tornado Cash ecosystem.

Staking is a mechanism that, in addition to the ability to vote on proposals, encourages users to lock up a quantity of TORN in exchange for rewards proportional to the amount locked. These rewards consist of an additional amount of TORN, distributed based on the amount staked.

Where do these rewards come from?

TORN staking rewards are not derived from an inflationary process (i.e. the creation of new tokens), but from the fees generated by using the protocol. This is made possible by the introduction of adecentralized relayer registry,which regulates the operation of the relayers.

Relayers are entities that facilitate user withdrawals in the protocol. To be included in the Tornado Cash UI, they must deposit a set amount (minimum 5k, and any higher increases the relayer’s “score”) of TORN as stake and maintain a sufficient balance to

cover the fees due to the staking contract. When a user makes a withdrawal through a relayer, the relayer charges the user a fee for the service. Separately, a fee equal to the0.3%of the withdrawal value (calculated in ETH) is converted into TORN at the current market price and deducted from the relayer's stake. This amount of TORN is then redistributed to users who participate in staking.

This system creates a balance of incentives: relayers gain visibility in the interface and collect fees from withdrawals, while users who stake TORN receive proportional rewards, supported by fees paid by relayers to the protocol.

7.3 Community Participation

ThereDAOOfTornado CashIt needs contributions from various figures to continue to grow and develop, including:

- Developerswhich continue and improve the development of the protocol.
- Auditorswho perform code reviews to find bugs and vulnerabilities.
- Content creatorwho promote and disseminate the protocol.

In June 2021, with proposal number 7, the community decided to create a fund to reward those who contribute to the development of the DAO. The funds are managed through aMulti-signature Walleton Gnosis Safe (an Ethereum smart contract that allows the creation of multi-sig wallets). The keys for this wallet were distributed to5 userschosen by voting. To validate a transaction, you need4 signatures (4-of-5). These users do not decide autonomously, but their task is to give or deny their signature based on community votes. To ensure the honesty of these parties, these users receive 100 TURNSper month from the community fund. The community can decide toto revokethe role of any of these users at any time.

Controversies and Current Status

Tornado Cash, the decentralized protocol for anonymous transactions on Ethereum, is at the center of a heated debate between privacy and illicit activities. Born in 2019, it guarantees anonymity by mixing transactions, but it has also been accused of facilitating activities such as money laundering and fraud.

8.1 Disputes

According to the*US Department of the Treasury's Office of Foreign Assets Control (OFAC)*,from 2019 to 2022,*Tornado Cash*hasover 7 billion dollars launderedin cryptocurrencies[21].

Its peak of fame came in 2021, during the explosion of the market of*NFT*(*Non-Fungible Token*), unique tokens known for exorbitant prices, such as the *Bored Ape Yacht Club*(from USD 250 to USD 1.6 million for rare pieces). However, many novice investors have been fooled by scams, and bad guys have exploited*Tornado Cash*to hide illegally obtained earnings.

Some cyber attacks have exploited*Tornado Cash*to launder funds. Among these, the attack on*Axie Infinity*of March 2022, in which the North Korean hacker group*Lazarus Group*stole millions of dollars in funds, laundering $455 million through*Tornado Cash*. They follow the*Horizon Bridge Exploit*of 2022, in which $100 million was stolen, and the*Bitmart Hack*of 2021, which resulted in a loss of $196 million.

TheMay 20, 2023,an anonymous hacker – who some suspect may be Gozzy, a developer associated with*Tornado Cash*–has compromised the *Tornado Cash DAO*via a malicious proposal, disguised as a legitimate update, containing malicious code. By exploiting the voting system, the attacker gained full control of the governance, assigning himself 1.2 million fake votes: with this power, hestolen483,000 TORN tokens from governance vaults, worth approximately$2.17 million,partially recycled through the protocol itself. The hacker subsequently returned control to the governance (perhaps satisfied with the loot), allowing the DAO balance to cover the losses and return the funds to the staked users. This episode, known as the

The first major attack on Tornado Cash revealed vulnerabilities in decentralized voting processes.

In a second accident, inFebruary 2024,an individual known as*Butterfly Effect*, presented as a community developer, introduced a governance proposal (issue 47) containing malicious JavaScript code. This modification compromised the protocol, redirecting users' deposit notes to a server controlled by the attacker, putting their ETH deposits at risk. The developers of*Tornado Cash*responded by confirming the breach, advising users to withdraw and replace the compromised notes, and urging token holders to revoke their votes for the proposal by deleting the malicious code. These two attacks highlight the inherent risks to trust in decentralized governance proposals.

8.2 Sanctions and legal battles

THE'August 8, 2022,the*OFAC*hassanctioned*Tornado Cash*, banning its use by US citizens[21]. The official domain was taken down, and GitHub removed the repository (later restored).*Edward Snowden*he called this move "*profoundly illiberal and authoritarian*".

TheNovember 26, 2024,there*US Court of Appeals*has established that smart contracts of*Tornado Cash*They are NOT punishable "properties" [12], and theJanuary 21, 2025sanctions have been lifted in Texas. The legal process to completely lift the sanctions is underway, making it likely thatlegality Of *Tornado Cash*in the USA once completed.

Developers, however, still face processes:

- Alexey Pertsev iswas arrested in the Netherlands in 2022, sentenced to 5 years and 4 months in 2024 [8], but temporarily released from house arrest onFebruary 8, 2025,declaring: "*Freedom is priceless, but my freedom cost a lot of money*".
- *Roman Storm*And*Roman Semenov*, two other major developers, have been charged in the US with laundering $1 billion.*Edward Snowden*he defended them, stating: "*Privacy is not a crime*".

8.3 Between privacy and dangers

*Tornado Cash*It doesn't only serve illicit purposes:*Vitalik Buterin*he used it for anonymous donations in Ukraine (https://x.com/VitalikButerin/status/1556925602233569280), protecting the identities of those supporting humanitarian causes in sensitive contexts. Activists in authoritarian regimes can receive funds without tracking, and individuals or workers paid in cryptocurrency can hide their earnings from hackers or data analytics firms. Small businesses can also mask transactions to protect business strategies from competitors, considering privacy a fundamental right.

However, the30%of the volume entered is illicit, [4], fueling the debate between freedom and security.

8.4 Today: a complicated access

Today, numerous scam Telegram sites and groups are proliferating. The official frontend is only accessible via IPFS links, a decentralized and censorship-resistant storage system, such as:

- https://ipfs.io/ipfs/bafybeie5mqhgqew2qbgzzjtt6c6ipntp37cwwkpd3zsezl44dgugezwlcm/
- https://2.torndao.eth.limo/

These links are verifiable in the pinned message of the official Telegram group. Centralized sites such ashttps://tornadoeth.cashare scams, with UI code that steals users' deposit notes. Safe alternatives include running locally *tornado-cli*or using the frontend locally*tornado-classic-ui*.

Risks:Centralized exchanges (CEX) lock funds coming directly from *Tornado Cash*, but moving them across multiple chains reduces this likelihood. With sanctions being lifted, CEXs may stop blocking funds and even re-list the TORN token, with*Coinbase*[5] among the first candidates, having supported the lawsuit. RPC providers such as*It rages*[16] block transactions of*Tornado Cash*, but alternatives exist to circumvent these restrictions. Some decentralized exchanges such as Uniswap block the purchase of TORN.

8.5 Usage statistics

[9][20]

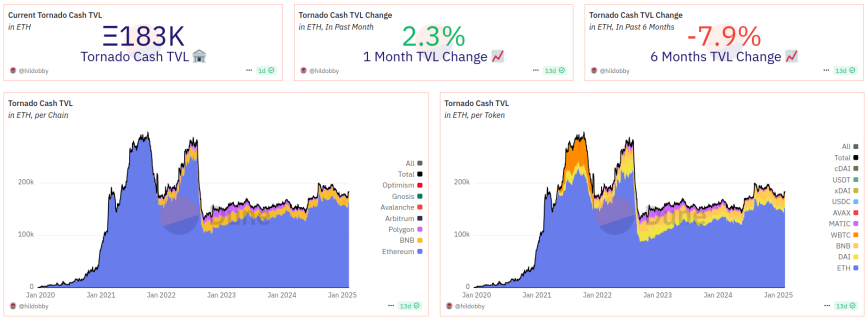


Figure 8.1: TVL of 183k ETH ~€456M

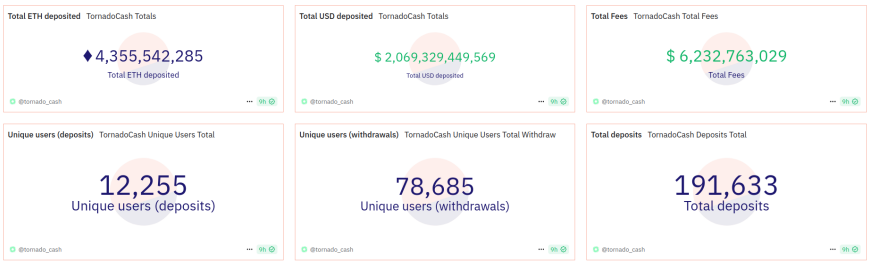


Figure 8.2: Total usage to date

Figure 8.3: Despite sanctions, it is still used

8.6 Community and developments

The community of*Tornado Cash*Andactive. The new official Telegram group was established through Governance Proposal #57, proposed onNovember 25, 2024,to counter scam groups. This proposal is officially recognized on *CoinMarketCap*[7] and*CoinGecko*[6], by inserting the link into the Telegram icon. The old Twitter/X account is inactive, but the DAO actively manages the protocol. Telegram groups with over 10k members are often scams with inactive accounts bought.

Crucial question: How to ban decentralized code, such as smart contracts *Tornado Cash*, which are run on local virtual machines? Despite US sanctions (in the process of being revoked and removed),*Tornado Cash*resists, used by those seeking privacy in sensitive contexts.

Insight: zk-SNARKs

9.1 Circuit

A circuit in the context of zk-SNARK, used in Tornado Cash, is a sequence of vertices and edges that forms a closed loop. More precisely:

- The initial vertex and the final vertex coincide
- There are no repetitions of intermediate edges or vertices (except for the start/end vertex)
- The sequence is closed, forming a sort of ring

In practice, the circuit is a cyclic graph, where data flows through vertices (operations) connected by edges (relationships between variables).

In Tornado Cash circuits are used as a framework for generating witnesses.

Circom is a declarative language based on R1CS (Rank-1 Constraint System)[3], a mathematical model for representing arithmetic circuits. A circuit written in Circom is therefore a sequence of operations that are translated/compiled into a system of mathematical constraints that can be used to generate zk-SNARKs or zk-STARKs.

An R1CS is a way to express a computation or arithmetic problem as a set of quadratic equations of rank 1. In practice, it transforms a program or a mathematical relation into a system of constraints that can be verified efficiently.

Each constraint is of the form: $TO \cdot x \cdot B \cdot x = C \cdot x$ Where:

- x is a vector of variables (which includes both the input and the intermediate and output variables of the calculation).
- A, B and C are vectors of coefficients (often represented as matrices when considering multiple constraints). The symbol "·" indicates the scalar product.

In this context, “rank 1” means that every constraint can be expressed as a product of two linear expressions ($TO \cdot x$ And $B \cdot x$) which, when multiplied, must equal a third linear expression ($C \cdot x$).

The circuit is compiled in the following manner:

1. A calculation is broken down into a sequence of basic operations.
2. Each operation is translated into an R1CS constraint.
3. The vector x contains a “witness”, that is, the values that satisfy all the constraints at the same time.

9.1.1 Witness generation

```
1   // Verifies that commitment that corresponds to given secret and
    nullifier is included in the merkle tree of deposits

2   templateWithdraw(levels)                {
3       signal      input     root;
4       signal      input     nullifierHash;
5       signal input recipient; // not taking part in any computations

6       signal input relay;                 // not taking part in any
    computations
7       signal input fee;                   // not taking part in any
    computations
8       signal input refund;                // not taking part in any
    computations
9       signal private input nullifier; signal private
10      input secret;
11      signal private input pathElements[levels ]; signal private
12      input pathIndices[levels ];
13
14      component hasher = CommitmentHasher ();
15      hasher.nullifier            <== nullifier;
16      hasher.secret <==           secret;
17      hasher.nullifierHash            === nullifierHash;
18
19      component tree = MerkleTreeChecker(levels); tree.leaf
20                      <==   hasher.commitment;
21      tree.root       <==   root;
22      for (var i = 0; i < levels; i++) {
23              tree.pathElements[i]            <==   pathElements[i];
24              tree.pathIndices[i]             <==   pathIndices[i];
25      }
26
```

```
27        // Add hidden signals to make sure that tampering with recipient or
          fee will invalidate the snark proof

28        // Most likely it is not required , but it 's better to stay on the safe side
           and it only takes 2 constraints

29        // Squares are used to prevent optimizer from removing
                    those constraints
30        signal      recipientSquare;
31        signal      feeSquare;
32        signal      relayerSquare;
33        signal      refundSquare;
34        recipientSquare <== recipient * recipient; feeSquare <==
35        fee * fee;
36        relayerSquare <== relayer * relayer; refundSquare
37        <== refund * refund;
38  }
```

The Withdraw circuit aims to:

1. Verify that a certain commitment (based on nullifier and secret) is valid and consistent with the public nullifierHash.

2. Confirm that this commitment is present in a Merkle tree with a certain root.

3. Ensure that additional parameters (such as recipient, fee, etc.) are bound in the proof.

and it does so in the following manner:

1.Nullifier Hash Check:The circuit takes the private nullifier and secret inputs and passes them to a component (CommitmentHasher) that computes two Pedersen hashes:

- Thecommitment hash:the hash of the entire commitment message (which combines nullifier and secret).
- Thenullifier hash:the hash of the nullifier alone.

Verify:The circuit compares the computed nullifier hash and the nullifierHash provided as public input, and enforces that they are equal (hasher.nullifierHash === nullifierHash)

2.Merkle Tree Check:The circuit uses the commitment hash (computed above), the public root of the Merkle tree (root), and the private inputs pathElements and pathIndices (the path in the tree) to verify that the commitment is a valid leaf of the tree.

- It starts from the leaf (the commitment hash) and uses a component called Muxer to combine the commitment with the first element of the path (pathElements[0]).
- The input pathIndices[0] (0 or 1) indicates whether the commitment hash is on the left or right in the next level hash computation, using a MiMC hasher.
- This process repeats level by level, using the hash of the previous level and the next pathElements[i], until a final hash is calculated.
- The circuit verifies that this final hash is equal to the public Merkle root (tree.root <== root).

3. Extra Withdrawal Parameter Check:The circuit takes the public inputs recipient, relayer, fee, and refund and calculates their squares (e.g. recipientSquare <== recipient * recipient). Even though these values are not used in the main calculations, adding them to the witness creates constraints that "lock" the transaction parameters. If someone tries to change them after the proof is generated, the constraints will no longer be satisfied and the proof will be invalid.

After creating the circuit, the task of generating the witness is entrusted to the calculateWitness function (https://github.com/tornadocash/snarkjs/blob/master/src/calculateWitness.js).This function takes as input the circuit and the set of input signals provided by the user. Its goal is to calculate and return an array, which contains the values of all the signals in the circuit needed to satisfy the defined constraints.

Generating the witness means executing the circuit step by step: assigning the inputs, calculating the intermediate values, and verifying that all the constraints are met. The result is an ordered list of numbers that represents the "solution" of the circuit, ready to be used as the basis for a ZKP.

This witness is then passed to a system like SNARK (snarkjs) to generate a compact cryptographic proof. This proof can be publicly verified by anyone to confirm that the circuit constraints are satisfied, without revealing the private data contained in the witness.

9.1.2 Proof calculation

Groth16 is a zk-SNARK zero-knowledge proof system. It was developed by Jens Groth in 2016 and is one of the most widely used schemes to prove that a computation was performed correctly without revealing the private data used in the computation.

Groth16 transforms an arithmetic circuit into a Quadratic Arithmetic Program (QAP), which represents constraints as polynomials. The proof shows that a witness exists w which satisfies the QAP polynomial equation, without revealing it.

The test is composed of three elements $\pi_{TO} \in G_1$, $\pi_B \in G_2$ And $\pi_C \in G_1$ Where G_1 And G_2 they are groups on an elliptic curve (the one used by Tornado Cash is BN254).

Setup and QAP

The circuit is converted to a QAP with polynomials $TO_{the}(x), B_{the}(x), C_{the}(x)$ for each signal w_{the} in the witness, and a divisor polynomial $Z(x)$, said *polynomial target* The QAP condition is:

$$\sum_{the=0}^{m-1} w_{the} TO_{the}(x) \cdot \sum_{the=0}^{m-1} w_{the} B_{the}(x) = \sum_{the=0}^{m-1} w_{the} C_{the}(x) + H(x) \cdot Z(x) \quad (9.1)$$

Where:

- m: total number of signals,
- w_{the}: witness values,
- $H(x)$: quotient polynomial, calculated by the prover.

Proving Key

The proving key contains pre-calculated values:

- $\alpha, \beta, \delta \in F_r$: random scalars,
- $[TO_{the}]_1 = [TO_{the}(t)]_1$, $[B_{the}]_1 = [B_{the}(t)]_1$, $[B_{the}]_2 = [B_{the}(t)]_2$, $[C_{the}]_1 = [C_{the}(t)]_1$: evaluations of polynomials at a secret point t, mapped on G_1 And G_2,
- $[\alpha]_1$, $[\beta]_1$, $[\beta]_2$, $[\delta]_1$, $[\delta]_2$: points generated by α, β, δ,
- $[H_{the}]_1$: points for the coefficients of $H(x)$, Where $H(x) = \sum h_{the} x_{the}$.

9.1.3 Generating the test

The test combines the witness w with the proving key, adding randomization to ensure zero-knowledge.

1. Initial calculation of the basic terms

- TO:

$$TO = \sum_{the=0}^{m-1} w_{the} [TO_{the}(t)]_1 \quad (9.2)$$

  Sum of signals w_{the} multiplied by the points $[TO_{the}]_1$.

- B:

$$B_1 = \sum_{the=0}^{n-1} w_{the}[B_{the}(t)]_1, \qquad B_2 = \sum_{the=0}^{n-1} w_{the}[B_{the}(t)]_2 \tag{9.3}$$

  Two sums: one in G_1 (B_1) and one in G_2 (B_2).

- C:

$$C = \sum_{the=n_{pub}+1}^{n-1} w_{the}[C_{the}(t)]_1 \tag{9.4}$$

  Sum of private signals (excluding public signals and the “one” signal).

- H:

$$H = \sum_{the=0}^{d-1} h_{the}[x^{the}]_1, \quad \text{Where} \quad H(x) = \frac{TO(x) \cdot B(x) - C(x)}{Z(x)} \tag{9.5}$$

  $H(x)$ is the quotient polynomial, with d degree of $Z(x)$.

Two random scalars are chosen $r, s \in \mathbb{F}_r$, used to mask the evidence.

- π_{TO}:

$$\pi_{TO} = TO + [\alpha]_1 + r \cdot [\delta]_1 \tag{9.6}$$

- π_B:

$$\pi_B = B_2 + [\beta]_2 + s \cdot [\delta]_2 \tag{9.7}$$

- π_C:

$$\pi_C = C + H + s \cdot \pi_{TO} + r \cdot B_1 - r \cdot s \cdot [\delta]_1 \tag{9.8}$$

9.1.4 Verifica

The proof check in Groth16 checks whether a proof (π_{TO}, π_B, π_C), generated for a *Quadratic Arithmetic Program* (QAP), satisfies the constraints of an arithmetic circuit without revealing the witness w. The verification is based on a coupling equation (*pairing*) on elliptic curves G_1 And G_2, using the *verifying key* and the public signs of the witness.

Verification equation:

The test checks the coupling equation:

$$And(\pi_{TO}, \pi_B) = And([\alpha]_1, [\beta]_2) \cdot And\left(\sum_{the=0}^{n_{pub}} w_{the}[IC_{the}]_1, [\gamma]_2 \right) \cdot And(\pi_C, [\delta]_2) \tag{9.9}$$

Where $And: G_1 \times G_2 \to G_T$ And a bilinear coupling function.

The proof is valid if the equation is satisfied, that is, if both sides are equal in G_T.

Linear combination calculation vk_x:

$$vk_x = [IC_0]_1 + \sum_{the=1}^{nPub} w_{the}[IC_{the}]_1 \tag{9.10}$$

vk_xrepresents the public signals weighted by the points of the verifying key.

Checking the pairing

It is calculated:

- Left side:$And(-\pi_{TO}, \pi_B)$,
- Right side:$And([\alpha]_1,[\beta]_2)\cdot And(vk_x,[\gamma]_2)\cdot And(\pi_C,[\delta]_2)$,

Where$-\pi_{TO}$Andthe denial ofπ_{TO}inG_1The test is valid if:

$$And(-\pi_{TO}, \pi_B) = And([\alpha]_1,[\beta]_2)\cdot And(vk_x,[\gamma]_2)\cdot And(\pi_C,[\delta]_2) \tag{9.11}$$

Bibliography

[1] Martin Albrecht et al. “MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity”. In:*International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 191–219.

[2] Basescan.*Basescan: Base Sepolia Network Testnet Explorer*.https://sepolia.basescan.org.Accessed: 24 February 2025. 2025.

[3] Eli Ben-Sasson et al. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge”. In:*Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II 33*. Springer. 2013, pp. 90–108.

[4] Chainalysis Team. “Understanding Tornado Cash, Its Sanctions Implications, and Key Compliance Questions.” In:*Chainalysis Blog*(Aug. 2022). url:https://www.chainalysis.com/blog/tornado-cash-sanctionchallenges/.

[5] Coinbase.*Coinbase - Buy, Sell, and Manage Cryptocurrency*. Accessed: 2025-02-24. 2025.url:https://www.coinbase.com/.

[6] CoinGecko.*CoinGecko - Cryptocurrency Prices and Market Data*. Accessed: 2025-02-24. 2025.url:https://www.coingecko.com/.

[7] CoinMarketCap.*CoinMarketCap - Cryptocurrency Prices, Charts, and Market Capitalizations*. Accessed: 2025-02-24. 2025.url:https://coinmarketcap.com/.

[8] “Developer Freed: Tornado Cash's Alexey Pertsev Out of Prison.” In:*AInvest*(2024). url:https://www.ainvest.com/news/developer-freedtornado-cash-s-alexey-pertsev-out-of-prison-2502101060ac71806a874f06/.

[9] Dune Analytics.*Dune - On-chain Crypto Analytics*. Accessed: 2025-02-24. 2025.url:https://dune.com/.

[10] Ethereum Foundation.*Ethereum - Decentralized Blockchain Platform*. Accessed: 2025-02-24. 2025.url:https://ethereum.org/.

[11] Etherscan.*Etherscan: The Ethereum Block Explorer*.https://etherscan.io. Accessed: 24 February 2025. 2025.

[12] “Federal Appeals Court Tosses OFAC Sanctions on Tornado Cash and Limits Federal Government's Ability to Police Crypto Transactions.” In: *Mayer Brown*(2024).url:https://www.mayerbrown.com/en/insights/publications/2024/12/federal-appeals-court-tosses-ofac-sanctions-on-tornado-cash-and-limits-federal-governmentsability-to-police-crypto-transactions.

[13] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In:*SIAM Journal on Computing*18.1 (1989), pp. 186–208.two:10.1137/0218012.url:https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/ Proof % 20Systems / The _ Knowledge _ Complexity _ Of _ Interactive _ Proof_Systems.pdf.

[14] Lorenzo Grassi et al. “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems.” In:*30th USENIX Security Symposium*. Cryptology ePrint Archive, Report 2019/458. 2019. eprint:2019/458.

[15] Lorenzo Grassi et al. “Poseidon2: A Faster Implementation of Poseidon for Zero-Knowledge Proof Systems”. In:*Cryptology ePrint Archive*(2023). Report 2023/323.

[16] It rages.*Infura - Ethereum API and Blockchain Infrastructure*. Accessed: 2025-02-24. 2025.url:https://www.infura.io/.

[17] Ralph C Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In:*Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 369–378.

[18] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Bitcoin.org*(2008).url:https://bitcoin.org/bitcoin.pdf.

[19] Torben Pryds Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In:*Annual International Cryptology Conference*. Springer. 1991, pp. 129–140.

[20] TRM Labs.*Tornado Cash Volume Dramatically Reduced Post-Sanctions, But Illicit Actors Are Still Using the Mixer*. Accessed: 2025-02-24. 2023. url:https : / / www . trmlabs . com/post/tornado-cash-volume-dramatically-reduced-post-sanctions-but-illicit-actors-arestill-using-the-mixer.

[21] US Department of the Treasury.*US Treasury Announces First US Strategy on Combating Corruption*. 2023.url:https://home.treasury. gov/news/press-releases/jy0916 (visited on 02/24/2025).