Tornado Cash

Emanuele Cornaggia, Alessandro Genova

February 2025

Contents

Abstract

L'ascesa della tecnologia blockchain ha introdotto livelli senza precedenti di trasparenza finanziaria grazie a registri pubblici come Ethereum. Tuttavia, questa trasparenza solleva anche sfide significative in termini di privacy. **Tornado Cash**, uno *smart contract* decentralizzato e non custodiale su Ethereum, affronta queste problematiche consentendo agli utenti di offuscare la cronologia delle transazioni attraverso avanzate tecniche crittografiche, come le *zero-knowledge proofs*.

Fin dal suo lancio, Tornado Cash ha alimentato un acceso dibattito all'interno della comunità blockchain e oltre. Se da un lato rappresenta uno strumento legittimo per rafforzare la privacy e la sovranità finanziaria, dall'altro ha attirato l'attenzione dei regolatori a causa del rischio di utilizzo per attività illecite, come il riciclaggio di denaro e l'elusione delle sanzioni economiche. La controversia ha raggiunto il suo apice nel 2022, quando l'Office of Foreign Assets Control (OFAC) del Dipartimento del Tesoro degli Stati Uniti ha imposto sanzioni contro il protocollo[21], segnando uno dei primi interventi legali nei confronti di un sistema basato su *smart contract* decentralizzati. Tuttavia, nel novembre 2024, un tribunale statunitense ha annullato le sanzioni, stabilendo che gli *smart contract* immutabili di Tornado Cash non possono essere considerati proprietà e, di conseguenza, non rientrano nelle restrizioni imposte dall'OFAC.

In questo lavoro andremo ad esporre il funzionamento di Tornado Cash, comprensivo di contratti e DAO. Vedremo le strutture matematiche sosttostanti. Andremo ad implementare i contratti su una Testnet e discuteremo delle controversie legate all'utilizzo del protocollo e lo stato attuale.

Introduzione

Una delle principali differenze tra la blockchain di Ethereum e quella di Bitcoin riguarda la gestione dei wallet, degli indirizzi di resto e dei modelli di transazione.

2.1 Modello UTXO di Bitcoin

Bitcoin adotta un modello di transazione basato sugli **UTXO** (Unspent Transaction Output), in cui ogni transazione spende uno o più output non ancora utilizzati (UTXO appunto) come input e genera un nuovo UTXO per ogni output. Quando un utente effettua una transazione, il wallet seleziona una serie di UTXO per coprire l'importo richiesto. Se la somma degli input eccede l'importo della transazione, il saldo residuo viene inviato a un indirizzo di resto (*change*), controllato dal wallet del mittente[18].

2.2 Modello Account-Based di Ethereum

Nel caso della blockchain di Bitcoin, l'elemento centrale è rappresentato dalla criptomoneta Bitcoin stessa, attorno alla quale si sviluppano tutte le operazioni di scambio. Al contrario, nella blockchain di Ethereum, la criptomoneta Ether svolge principalmente il ruolo di "carburante" necessario per il funzionamento della rete, mentre gli strumenti di maggiore rilevanza sono gli **smart contract** e i **token**, sia fungibili che non fungibili. Inoltre, attraverso specifici *smart contract*, è possibile vincolare temporaneamente una determinata quantità di token mediante un meccanismo noto come **staking**, comunemente impiegato all'interno di contratti di finanza decentralizzata (DeFi).

Ethereum utilizza un modello **account-based** (invece di UTXO) perché è più adatto alla gestione di *smart contract* complessi. Questo modello consente aggiornamenti diretti dello stato globale, inclusi saldi degli account, variabili dei contratti e altre informazioni persistenti. Inoltre, semplifica la programmabilità, facilitando l'interazione tra contratti e account ed evitando la frammentazione delle transazioni tipica del modello UTXO, che non gestisce uno stato persistente in modo nativo. Ogni indirizzo Ethereum possiede un saldo persistente che

viene direttamente modificato dalle transazioni. In questo sistema non esistono UTXO; le transazioni si limitano a detrarre l'importo dal saldo del mittente e ad accreditarlo al destinatario[10].

2.3 Privacy e Anonimato: Introduzione a Tornado Cash

Avere un account personale fisso implica che tutte le transazioni siano completamente visibili da chiunque, eliminando quasi completamente la privacy, che rimane solamente nel collegare l'indirizzo del wallet ad una persona fisica.

Tornado Cash nasce con l'idea di migliorare la privacy, garantendo anonimato sulle transazioni. Può essere particolarmente utile per attivisti, come quelli che operano in paesi con regimi repressivi, giornalisti che trattano temi sensibili, o anche aziende che vogliono proteggere le loro transazioni finanziarie. Inoltre, può supportare i donatori che contribuiscono a cause controverse o politicamente sensibili, senza rischiare di essere identificati o perseguiti.

È un "*non-custodial protocol*", ovvero gli utenti mantengono il pieno controllo dei propri fondi e chiavi private, senza doverli affidare a una terza parte. Grazie agli *smart contract*, questo protocollo è visibile a tutti e non è modificabile, nemmeno dallo sviluppatore. L'idea principale è la seguente:

1. L'utente manda un certo ammontare di criptomoneta che intende usare per effettuare il pagamento al contratto di Tornado Cash.
2. L'utente genera una prova di aver fatto un deposito.
3. Utilizzando questa prova, un qualsiasi altro utente può ritirare i fondi su un wallet diverso, senza che vi sia un collegamento diretto con il deposito iniziale.

Chiunque voglia monitorare le transazioni del nostro indirizzo potrà vedere che abbiamo inviato criptovaluta al contratto di Tornado Cash, ma non sarà in grado di risalire al destinatario del prelievo. Allo stesso modo, se riceviamo fondi da Tornado Cash, l'osservatore potrà vedere l'accredito ma non potrà identificare l'indirizzo del mittente.

2.4 Token e Chain Supportate

L'originale implementazione, oltre ad Ether (ETH), prevede l'utilizzo dei seguenti token (sempre sulla chain di Ethereum):

- DAI
- cDAI
- USDC

- USDT
- WBTC

Con l'ultima versione aggiornata al 2021, Tornado Cash è stato reso disponibile, oltre che per Ethereum, anche per le seguenti chain:

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

2.5 Versioni di Tornado Cash

Esistono due versioni di Tornado Cash:

- Tornado Cash Classic
- Tornado Cash Nova

In Tornado Cash Classic, quando un utente deposita la criptomoneta in una pool, viene generato un segreto. Questo segreto funziona come una chiave privata: un utente che ne è in possesso può in qualsiasi momento ritirare i fondi su un altro indirizzo.

"In Tornado Cash Nova, i fondi sono direttamente legati all'indirizzo usato per il deposito, senza ricorrere a un segreto. L'utente può ritirarli connettendosi alla pool tramite lo stesso indirizzo. La robustezza del protocollo si affida al numero di utenti e l'entità della pool (più transazioni avvengono, e più è difficile effettuare un analisi che ricolleghi le transazioni)

Primitive

3.1 zk-SNARKs

In questo paragrafo parleremo in modo sintetico delle zk-SNARKs, in quanto abbiamo dedicato un capitolo a parte per parlarne in modo molto più approfondito.

3.1.1 ZKP

Una Zero-Knowledge Proof (ZKP) è un protocollo crittografico che permette ad una parte (detta prover) di convincere un'altra parte (detta verifier) che un'affermazione è vera, senza rivelare alcuna informazione oltre il semplice fatto che l'affermazione è vera.

Questo concetto venne introdotto nel 1985 da Goldwasser, Micali e Rackoff nel loro paper "The Knowledge Complexity of Interactive Proof-Systems"[13].

Un protocollo di Zero-Knowledge Proof deve soddisfare tre proprietà essenziali:

- **Completezza**: se l'affermazione è vera e il prover segue correttamente il protocollo, il verifier accetterà la prova con alta probabilità.
- **Correttezza**: un prover disonesto non può ingannare il verifier facendogli accettare un'affermazione falsa, salvo una probabilità trascurabile.
- **Zero-Knowledge**: la prova non rivela alcuna informazione utile sul segreto sottostante, oltre alla veridicità dell'affermazione.

Queste proprietà garantiscono che il verifier possa essere certo della correttezza dell'affermazione senza ottenere dettagli che compromettano la riservatezza del segreto.

Le ZKP possono essere classificate in due principali categorie

- **Prove interattive**: richiedono uno scambio di messaggi tra prover e verifier per raggiungere la convinzione sulla validità dell'affermazione. Un esempio è il protocollo di Schnorr per l'autenticazione.

- **Prove non interattive (NIZK)**: eliminano la necessità di interazione, consentendo la verifica tramite un'unica trasmissione di dati. L'euristica Fiat–Shamir puó venir usata per trasformare alcune ZKP interattive in NIZK.

3.1.2 zk-SNARKs

La Zero-Knowledge Succinct Non-Interactive Argument of Knowledge sono una sottoclasse delle NIZK.

La principali carattestiche sono:

- **Zero-Knowledge**: la prova non rivela alcuna informazione sul segreto sottostante, oltre alla veridicità dell'affermazione.
- **Succintezza**: la prova è di dimensione ridotta e può essere verificata in tempo polinomiale rispetto alla dimensione dell'input.
- **Non interattivo**: il protocollo non richiede interazione tra prover e verifier.
- **Argument of Knowledge**: il prover può generare una prova valida solo se possiede effettivamente la conoscenza del witness (contiene valori e segreti che soddisfano una determinata affermazione o relazione matematica).

zk-SNARKs sono progettati in maniera da essere efficienti e da poter venir utilizzati per verificare un grande numero di statements contemporaneamente.

3.2 Pedersen Hash

[19] Sia G un gruppo ciclico di ordine q primo e caratteristica p, generato da un elemento g, con un secondo generatore h scelto in modo che il problema del logaritmo discreto (DLP) tra g e h sia computazionalmente intrattabile (in altre parole, deve essere molto difficile calcolare l'esponente x tale che $h = g^x$).

Il Pedersen Hash di un messaggio m viene definito come:

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

dove:

- m è il messaggio da hashare, considerato come un elemento numerico del campo finito sottostante
- r è un valore casuale scelto uniformemente in $\mathbb{Z}_q$, usato per introdurre entropia e ottenere proprietà di sicurezza
- g e h sono generatori del gruppo G

Proprietá:

- **Collision Resistance**: se il gruppo G è sufficientemente grande e il problema del logaritmo discreto è difficile, trovare due messaggi distinti m_1, m_2 tali che $H(m_1) = H(m_2)$ è computazionalmente difficile.
- **Hiding**: il valore di hash non rivela informazioni dirette sul messaggio m, poiché è mascherato dal valore casuale r.
- **Binding**: se r non è noto, è computazionalmente difficile trovare un altro messaggio m', e un valore casuale r' tali che $H(m) = H(m')$.

Pedersen hash è progettato per ottimizzare l'efficienza computazionale all'interno dei circuiti Zero-Knowledge.

L'hashing di un messaggio tramite la funzione di hash di Pedersen comprime i bit del messaggio in un punto su una curva ellittica denominata Baby Jubjub. Questa curva è definita nell'ambito della curva ellittica BN128, supportata da operazioni precompilate sulla rete Ethereum introdotte con l'EIP-196. Di conseguenza, le operazioni che utilizzano la curva Baby Jubjub, come l'hashing di Pedersen, risultano altamente efficienti in termini di gas.

Quando si calcola l'hash di Pedersen di un messaggio, il punto risultante sulla sua curva ellittica è estremamente efficiente da verificare, ma il processo di inversione per recuperare il messaggio originale è computazionalmente intrattabile.

3.3 Funzione hash MiMC

[1] Minimized Multiplicative Complexity è un algoritmo crittografico progettato specificamente per ottimizzare l'efficienza computazionale in contesti di zero-knowledge proofs (ZKP), come SNARKs e STARKs. La sua struttura è stata sviluppata per ridurre la complessità moltiplicativa, un fattore critico nei protocolli di prova a conoscenza zero, dove i costi computazionali delle moltiplicazioni modulari influenzano direttamente l'efficienza del sistema.

Nello specifico MiMC è una funzione di hash iterativa costruita su un approccio a rete di Feistel.

Sia x l'input del messaggio e k una chiave segreta (opzionale). La funzione di round è iterata su un numero T di round con la seguente trasformazione:

$$x_{i+1} = (x_i + C_i + k)^e \pmod{p}$$

dove:

- C_i é una costante di round predefinita e pubblica
- e é un piccolo esponente dispari
- p é un primo grande che definisce il campo sottostante $\mathbb{F}_p$

- k é un valore di chiave opzionale

Il valore hash finale è l'output dopo T iterazioni della trasformazione sopra descritta.

MiMC garantisce sicurezza crittografica sfruttando la non linearità introdotta dall'esponente e e la difficoltà del problema del logaritmo discreto nel campo finito $\mathbb{F}_p$. Le principali proprietà includono:

- **Resistenza alla preimmagine**: date le operazioni modulari con un esponente, la funzione è difficile da invertire.
- **Collision Resistant**: l'uso di costanti di round differenti garantisce che due input distinti non producano lo stesso hash con probabilità significativa. L'uso di un esponente dispari aiuta a evitare simmetrie (che avremmo con un esponente pari: $x^e = (-x)^e$).
- **Bassa complessità computazionale**: il costo delle operazioni modulari è ridotto grazie all'uso di un unico esponente fisso piccolo, minimizzando il numero di moltiplicazioni necessarie rispetto ad altre costruzioni crittografiche basate su permutazioni pi'u complesse.

3.4 Merkle Tree

[17] Noto anche come albero di hash binario, è una struttura dati gerarchica utilizzata in crittografia e informatica distribuita per garantire l'integrità e la verificabilità delle informazioni in modo efficiente e sicuro. Esso prende il nome dal suo inventore, Ralph Merkle, che lo ha introdotto negli anni '80 come parte della sua ricerca su metodi sicuri di verifica dei dati in contesti distribuiti.

L'albero di Merkle è essenzialmente un albero binario in cui i nodi foglia rappresentano gli hash di dati individuali, mentre i nodi interni contengono gli hash dei nodi figli. La radice dell'albero, nota come radice di Merkle, rappresenta un hash aggregato di tutti i dati sottostanti, che può essere usato per verificare l'integrità dell'intero set di dati in modo compatto.

Un albero di Merkle è costruito iterativamente in una maniera che riduce il numero di hash necessari per verificare l'integrità dei dati. La sua struttura è la seguente:

- **Nodi foglia**: Ogni nodo foglia dell'albero rappresenta un hash del dato originale, denotato come $H(d_i)$, dove d_i è l'elemento di dati originale.
- **Nodi interni** Ogni nodo interno è ottenuto calcolando l'hash della concatenazione degli hash dei suoi figli. Ad esempio, se un nodo ha due figli A e B, il nodo padre conterrà l'hash $H(A \mid\mid B)$ (dove $\mid\mid$ è l'operatore di concatenazione).

- **Mertkle Tree Root** è l'hash finale dell'albero. Rappresenta un'unica stringa di dati che può essere utilizzata per verificare l'integrità dell'intero albero. Contiene un'informazione su tutti i nodi foglia.

L'albero di Merkle è particolarmente utile per strutture dati grandi, poiché riduce notevolmente la quantità di dati necessari per verificare l'integrità di un singolo elemento.

Proprietà:

- **Integrità dei dati**: la radice di Merkle fornisce una sintesi compatta dell'intero set di dati, e qualsiasi modifica a un singolo dato (ad esempio, a una foglia dell'albero) causerà un cambiamento significativo nell'hash della radice. Pertanto, se la radice rimane invariata, l'integrità dei dati sottostanti è garantita.
- **Verificabilità Efficiente**: una delle principali forze dell'albero di Merkle è la possibilità di verificare l'integrità di un singolo dato senza la necessità di scaricare o verificare l'intero albero. Per verificare un singolo dato, è sufficiente conoscere l'hash della radice e una "prova di Merkle", che consiste negli hash dei nodi interni lungo il percorso dalla foglia alla radice. Questa riduzione dei dati da verificare rende gli alberi di Merkle ideali per applicazioni con grandi quantità di dati distribuiti.
- **Resistenza alle Collisioni**: se la funzione di hash sottostante è resistente alle collisioni, allora l'albero di Merkle sarà anch'esso resistente alle collisioni. In altre parole, sarà computazionalmente difficile per un attaccante trovare due set di dati distinti che generano la stessa radice di Merkle.

Funzionamento Smart Contract

In questa sezione analizziamo il funzionamento tecnico di Tornado Cash, esaminando i meccanismi crittografici e le strutture dati che garantiscono la privacy delle transazioni. Approfondiremo le seguenti componenti fondamentali:

- **Deposit**: il meccanismo con cui gli utenti bloccano i fondi nel contratto.
- **Merkle Tree**: la struttura utilizzata per organizzare i depositi e verificare la validità delle prove.
- **Generazione della prova**: il processo crittografico che permette di dimostrare il possesso di un impegno senza rivelarne il valore.
- **Withdrawal**: la fase di prelievo, in cui un utente può riscattare i fondi in modo anonimo su un indirizzo diverso.

Ogni smart contract su Ethereum è identificato da un indirizzo univoco, noto come **Contract Address (CA)**. Esistono molteplici contratti che forniscono diverse funzionalitá. Il contratto con cui si deve interagire per depositare e ritirare criptomoneta si chiama TornadoRouter. Questo contratto fa da punto di ingresso unico: quando si interagisce con esso, si passa come input il contratto relativo alla pool criptomoneta che si vuole depositare o prelevare. Il vantaggio di questa scelta é che i contratti delle pool possono rimanere fissi, e se c'é bisogno di applicare un cambiamento al protocollo, si puó deployare un altro TornadoRouter. Inoltre c'é da notare che nella versione Classic di Tornado Cash, le pool rappresentano la criptomoneta ed un taglio prefissato (per ETH esistono le pool 0.1, 1, 10 e 100 ETH). Questa scelta é stata fatta per rendere piú difficile qualsiasi tipo di analisi in cui si vuole ricollegare lo stesso ammontare di criptomoneta di un deposito a quella di un ritiro. L'indirizzo di TornadoRouter è il seguente:

0xd90e2f925DA726b50C4Ed8D0Fb90Ad053324F31b

Il contratto è stato distribuito sulla blockchain attraverso una specifica transazione, identificata da un **Transaction Hash**. Questo hash rappresenta l'operazione di **creazione dello smart contract** sulla rete Ethereum. La transazione corrispondente è:

0x51baa3dbb7e8756d52ba1b863f41e7af38373d0bb9943b9585d7f0db2d419e6e

Analizziamo il codice della prima pool creata, infatti si chiama TornadoCash_eth e non specifica il taglio di ETH, il quale è 10. Per analizzare direttamente il codice sorgente del contratto, è possibile consultarlo su Etherscan al seguente 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.
3     @param _commitment the note commitment, which is
     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  }
```

Un utente genera due numeri primi con lunghezza 31 bits. Il primo numero lo chiamiamo **nullifier** mentre il secondo **secret** (in quanto rappresenterà il segreto per riscattare la somma). Nullifier e secret vengono concatenati e al loro risultato viene applicata la funzione di **hash di Pedersen**. Questo hash rappresenta il nostro **committment** (il parametro `_committment` della funzione `deposit`).

Questo commitment da noi generato viene passato come paramentro della funzione `_insert`, la quale andrà ad inserirlo nel **MerkleTree** e ci restituirà l'**indice** in cui il commitment è stato inserito.

`commitments [ _commitment ] = true ;` va ad aggiungere il commtiment nel dizionario dei commitment, impostandone il valore a true. Questo passaggio viene fatto per evitare di inserire lo stesso commitment due volte nell'albero. Il controllo avviene appena sopra: `require(!commitments[_commitment])`

`_processDeposit();` serve per controllare che i fondi sono stati mandati alla pool giusta.

Infine, tramite la chiamata

`emit Deposit(_commitment, insertedIndex, block.timestamp);`, viene generato un evento di tipo Deposit, il quale viene registrato nel log delle transazioni. Poiché gli eventi non sono direttamente accessibili dal contratto una volta emessi, questa registrazione consente di mantenere una traccia permanente dell'operazione sulla blockchain. Tramite questo evento viene ottenuto come l'indice della foglia del Merkle Tree in cui è stato inserito il commitment.

4.2 Merkle Tree

La costruzione del Merkle Tree in Tornado Cash viene effettuata una sola volta al momento della distribuzione del contratto sulla blockchain. Questo avviene nel costruttore `constructor(uint32 _treeLevels)` della classe `MerkleTreeWithHistory`, dove viene inizializzato l'albero con un livello di profondità predefinito (nel caso di Tornado Cash, 20 livelli). Questo consente di poter utilizzare la pool per 2^{20} depositi, poi ne dovrà venire creata una nuova.

Dopo questa fase, l'albero è pronto per ricevere nuovi depositi, che saranno inseriti in tempo reale tramite la funzione `_insert()`, aggiornata ogni volta che un nuovo *commitment* viene registrato.

4.2.1 Costruzione

```
1  contract MerkleTreeWithHistory {
2    uint256 public constant FIELD_SIZE = 218882428718392
       7522224640574525727508854836440041603434369820418657
       5808495617;
3    uint256 public constant ZERO_VALUE = 216638390044169
       3294538235590879059922526650182290791145750497851557
       8255421292; // = keccak256("tornado") % FIELD_SIZE
4
5    uint32 public levels;
6
7    // the following variables are made public for
       easier testing and debugging and
8    // are not supposed to be accessed in regular code
9    bytes32[] public filledSubtrees;
10   bytes32[] public zeros;
11   uint32 public currentRootIndex = 0;
12   uint32 public nextIndex = 0;
13   uint32 public constant ROOT_HISTORY_SIZE = 100;
14   bytes32[ROOT_HISTORY_SIZE] public roots;
15
16   constructor(uint32 _treeLevels) public {
```

```
17      require(_treeLevels > 0, "_treeLevels should be
        greater than zero");
18      require(_treeLevels < 32, "_treeLevels should be
        less than 32");
19      levels = _treeLevels;
20
21      bytes32 currentZero = bytes32(ZERO_VALUE);
22      zeros.push(currentZero);
23      filledSubtrees.push(currentZero);
24
25      for (uint32 i = 1; i < levels; i++) {
26        currentZero = hashLeftRight(currentZero,
        currentZero);
27        zeros.push(currentZero);
28        filledSubtrees.push(currentZero);
29      }
30
31      roots[0] = hashLeftRight(currentZero, currentZero)
        ;
32    }
33
34    // Altre funzioni seguono nel codice originale
```

Il Merkle Tree utilizzato da Tornado Cash prende come input un livello di profonditá, il quale é compreso tra 0 e 32 estremi esclusi.

Per poter ottenere questa informazione abbiamo due opzioni:

- Interagendo col contratto e richiedendo il valore del campo `uint32 public levels`.
- Andare nella transazione di creazione del contratto e guardare gli input forniti.

Utilizzando questi due metodi si scopre che l'altezza che è stata decisa di attribuire al Merkle Tree è di 20.

Il valore `ZERO_VALUE` è calcolato nel seguente modo: `keccak256("tornado") % FIELD_SIZE`. Ciò puó essere facilmente verificato con il seguente script python:

```
1  import sha3     # install safe-pysha3
2
3  FIELD_SIZE = 21888242871839275222246405745257275088548
       364400416034343698204186575808495617
4
5  keccak = sha3.keccak_256()
6  keccak.update(b"tornado")
7
8  print(int(keccak.hexdigest(), 16) % FIELD_SIZE)
```

Il codice

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

crea un valore iniziale e lo usa per riempire l'albero andandolo ad aggiungere agli array `zeros` e `filledSubtrees`. Questi array vengono usati per rappresentare i **nodi vuoti** e i **sottoalberi riempiti**, rispettivamente.

Il codice procede iterativamente alla costruzione dei MerkleTree calcolando l'hash delle foglie (che al momento hanno valore `currentZero`) per ottenere l'hash genitore delle due foglie, per poi calcolare l'hash del rootMerkleTree.

Gli hash vengono calcolati dall'algoritmo MiMC sfruttando l'implementazione della libreria `Hasher.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) < FIELD_SIZE, "_right
   should be 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 Inserimento

Precedentemente abbiamo visto che quando effettuiamo un deposito e passiamo il commitment alla funzione `deposit`, lo smart contract chiama il metodo `_insert`, per andare ad inserirlo nel 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;
27      return nextIndex - 1;
28    }
```

La funzione `_insert` accetta come input il **commitment** e restituisce l'**indice della foglia** in cui è stato inserito. L'aggiornamento dell'albero consiste nell'inserire commitment nella prima foglia disponibile, ricalcolando successivamente gli hash MiMC intermedi lungo il percorso dalla foglia alla radice.

Il processo utilizza un meccanismo basato su `currentIndex` e `filledSubtrees`. A ogni livello, se `currentIndex` è pari, il nuovo hash viene assegnato al nodo sinistro e il destro è impostato a `zeros[i]`, aggiornando `filledSubtrees[i]` con il nuovo valore. Se `currentIndex` è dispari, il nodo sinistro è recuperato da `filledSubtrees[i]` e il nuovo hash diventa il destro. Questo sistema consente di mantenere traccia dei sottoalberi completi e di aggiornare efficientemente l'albero.

Il Merkle Tree ha una profondità di 20, permettendo un massimo di $2^{20} \approx 1.048.576$ foglie. Il numero di foglie attualmente occupate è indicato dalla variabile pubblica `nextIndex`, che rappresenta sia il contatore dei depositi effettuati sia l'indice della prossima foglia disponibile. Quando `nextIndex` raggiunge 2^{20}, l'albero è pieno e ulteriori depositi non sono più possibili. Attualmente, il valore medio di `nextIndex` è circa 50.000, ben al di sotto del limite massimo.

4.2.3 Esempio di esecuzione

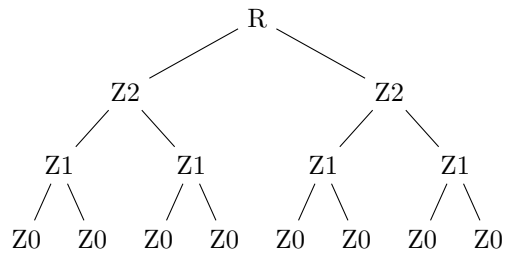
Consideriamo un Merkle Tree di 4 livelli (profondità 3, con $2^3 = 8$ foglie). Analizziamo sinteticamente 6 inserimenti, mostrando lo stato di `filledSubtrees` e l'albero. Definiamo:

- Livelli: 0 (foglie) a 3 (radice).
- `filledSubtrees`: array di dimensione 3 per i livelli 0, 1, 2. Memorizza l'hash del sottoalbero sinistro completo a ogni livello, usato per calcolare i nodi superiori.
- $Z_i = \text{zeros}[i]$: hash predefiniti per nodi vuoti.
- N_i: hash intermedi (es. $N1 = \text{hashLeftRight}(C0, C1)$).

Costruzione Iniziale

Il Merkle Tree viene inizializzato con `levels = 3`.

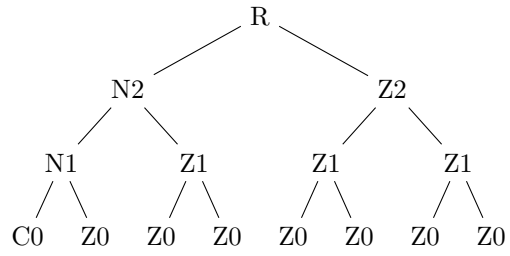
- `nextIndex` $= 0$
- `filledSubtrees` $= [Z_0, Z_1, Z_2]$, dove $Z_0 =$ ZERO_VALUE, $Z_1 = h(Z_0, Z_0)$, $Z_2 = h(Z_1, Z_1)$
- Albero:



- Radice iniziale: $R = h(Z_2, Z_2)$

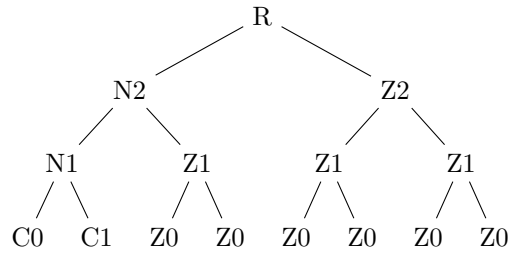
Inserimento 1: Foglia $C0$ (indice 0)

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



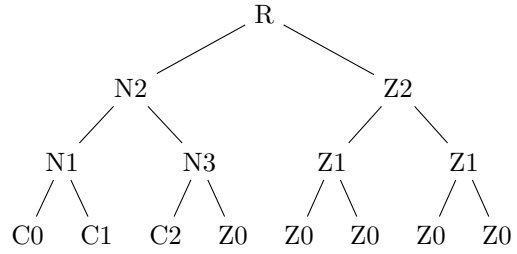
Inserimento 2: Foglia $C1$ (indice 1)

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



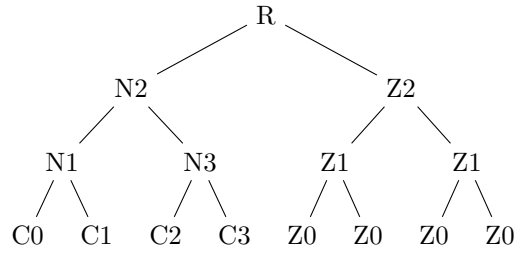
Inserimento 3: Foglia $C2$ (indice 2)

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



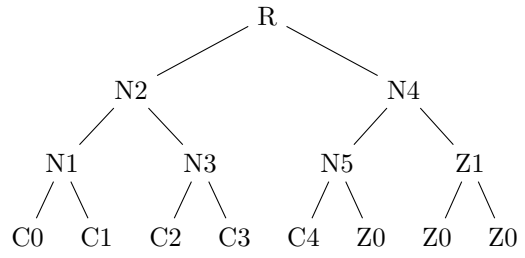
Inserimento 4: Foglia $C3$ (indice 3)

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



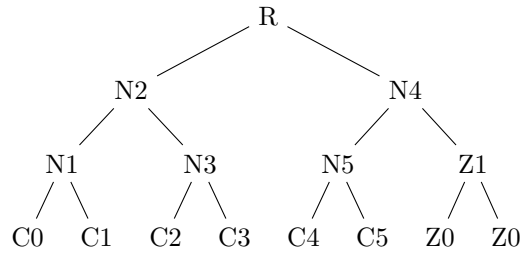
Inserimento 5: Foglia $C4$ (indice 4)

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



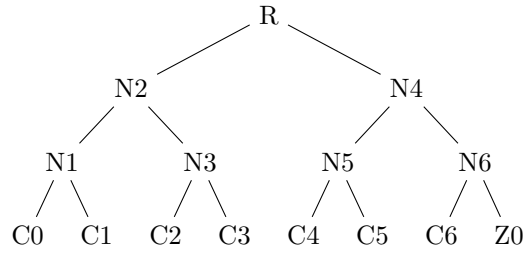
Inserimento 6: Foglia $C5$ (indice 5)

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



Inserimento 7: Foglia $C6$ (indice 6)

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



Dove:

- $N1 = \text{hashLeftRight}(C0, C1)$,

- $N3 = \text{hashLeftRight}(C2, C3)$,
- $N5 = \text{hashLeftRight}(C4, C5)$,
- $N6 = \text{hashLeftRight}(C6, Z0)$,
- $N2 = \text{hashLeftRight}(N1, N3)$,
- $N4 = \text{hashLeftRight}(N5, N6)$,
- $R = \text{hashLeftRight}(N2, N4)$ (ultimo stato).

4.3 Generazione della prova

In questo step avviene il passaggio chiave di tutto il protocollo, bisogna dimostrare di avere conscenza della preimmagine di un commitment presente nel Merkle Tree, senza poterla rivelare, in quanto questo porterebbe a poter calcolare il commitment e quindi collegare l'azione di deposit con quella di whitrdraw, andando ad eliminare totalmente tutto il lavoro fatto fino ad ora, per poter garantire l'anonimato.

Questo passaggio avviene tramite l'utilizzo di algoritmo zk-SNARK.

Il **witness** é l'insieme di dati che serve per generare una prova che dimostra che siamo al corrente dei dati privati, senza peró rivelarli.

Questo passaggio non puó avvenire direttamente nello smart contract di Tornado Cash, in quanto i parametri passati in input devono rimanere privati. Esso viene quindi eseguito dall'utente off-chain sul proprio device (la repository github di Tornado Cash fornisce dei tools per poter fare questo passaggio da linea di comando o da interfaccia grafica, per facilitare il lavoro all'utente).

Andiamo ad analizzare il codice sorgente della CLI:
`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
3   const { root, pathElements, pathIndices } = await
     generateMerkleProof(deposit, currency, amount);
4
5   // Prepare circuit input
6   const input = {
7     // Public snark inputs
8     root: root,
9     nullifierHash: deposit.nullifierHash,
10    recipient: bigInt(recipient),
11    relayer: bigInt(relayerAddress),
12    fee: bigInt(fee),
13    refund: bigInt(refund),
```

```
14
15      // Private snark inputs
16      nullifier: deposit.nullifier,
17      secret: deposit.secret,
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  }
```

Il witness del deposito é dato dai sequenti dati privati:

- **secret**
- **nullifier**
- **pathElements**: i nodi appartenenti al path dalla leaf dal deposito, alla Merkle Tree Root
- **pathIndices**: gli indici dei nodi di pathElements

e dai seguenti dati pubblici:

- **root**: una qualsiasi merkleRoot contenuta nella variable `roots`
- **nullifierHash**: il Pedersen Hash del nullifier
- **relayer**: opzionale, l'indirizzo del relayer
- **fee**: opzionale, le fee da dare al relayer

- **refund**: opzionale, il refund da dare al relayer

La SNARK proof viene generata dalla libreria `websnark` mediante la funzione `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 spends
      - the recipient of funds
      - optional fee that goes to the transaction sender (usually a relay)
  */
  function withdraw(bytes calldata _proof, bytes32 _root, bytes32 _nullifierHash, address payable _recipient, address payable _relayer, uint256 _fee, uint256 _refund) external payable nonReentrant {
    require(_fee <= denomination, "Fee exceeds transfer value");
    require(!nullifierHashes[_nullifierHash], "The note has been already spent");
    require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use 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, _relayer, _fee, _refund);
    emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
  }
```

Una volta effettuato un deposito, per poterlo ritirare, bisogna invocare la funzione whitdraw.

I dati che deve avere in possesso un utente per poter ritrare i fondi sono i parametri pubblici utilizzati per generare la proof, e la proof stessa.

Il contratto andrá ad eseguire delle operazioni di controllo:

1. Le relayer fees non devono superare l'ammontare del deposito (per evitare underflow).
2. Il nullifierHash non deve essere presente nel dizionario nullifierHashes, qualora fosse presente indicherebbe che l'ammontare é gia stato ritirato (double spending).
3. La Merkle Tree Root è presente nella variabile root del Merkle Tree.
4. Controlla che la proof sia effettivamente valida.

Una volta che le condizioni sono valide, viene aggiunto l'hash del nullifier al dizionario nullifierHashes e viene emesso l'evento Withdrawal.

Parte pratica

In questa sezione andiamo ad analizzare una transazione usando Tornado Cash. Non potendo fare ciò, per questioni legali, col contratto originale. Abbiamo deployato una copia del contratto sulla TestNet Sepolia Base.

5.1 Sepolia Base Testnet

La Sepolia Base Testnet è una rete di prova pubblica utilizzata per sviluppare e testare applicazioni sulla blockchain Base, un Layer 2 di Ethereum creato da Coinbase[5]. Questa testnet fornisce un ambiente sicuro per testare smart contract e dApp.

Questa rete replica il comportamento di Ethereum, consentendo agli sviluppatori di testare le proprie applicazioni con la certezza che i risultati saranno coerenti una volta che verranno distribuite sulla rete principale di Ethereum.

La principale differenza riguarda il protocollo di consenso. Poiché le monete su Sepolia Base non hanno valore reale, non esiste un incentivo economico per garantire il corretto funzionamento della rete attraverso il meccanismo di Proof of Stake. Per questo motivo, i validatori vengono selezionati direttamente da Coinbase, rendendo la rete centralizzata. Tuttavia, questo non rappresenta un problema, poiché Sepolia Base è un ambiente di test progettato esclusivamente per lo sviluppo e la sperimentazione.

Per ottenere Ether su Sepolia Base, è necessario utilizzare le faucet, servizi che distribuiscono gratuitamente una piccola quantità di ETH ai wallet richiedenti. Questi ETH di test vengono utilizzati per pagare le commissioni di transazione, consentendo agli sviluppatori di interagire con la blockchain senza costi reali.

5.2 Poseidon Hash

La versione di Tornado Cash che abbiamo creato ha un miglioramento rispetto alla versione base: l'utilizzo di Poseidon Hash al posto di Pedersen Hash. Questa miglioria è stata proposta da ABDK Consulting, quando è stato chiesto l'audit del contratto di Tornado Cash.

Un audit è una verifica della sicurezza di un contratto. Uno sviluppatore può richiedere ad un'azienda un audit, la quale dopo aver studiato il codice ne garantisce la sicurezza, fornendone un certificato. Questo è uno step cruciale nelle blockchain, in quanto in un ambiente decentralizzato se interagisci col contratto sbagliato rischi di perdere i soldi e non c'è nessun'entità centrale che può cancellare la transazione, quindi i progetti fanno affidamento su questi audit per dimostrare la sicurezza dei proprio contratti.

5.2.1 Parte tecnica su Poseidon Hash

Poseidon Hash è una funzione di hashing crittografica progettata per garantire alta efficienza nei circuiti a conoscenza zero (Zero-Knowledge, ZK) e nelle prove di conoscenza zero (Zero-Knowledge Proofs, ZKPs), come SNARKs e STARKs. È stata introdotta nel 2019 nel paper *"Poseidon: A New Hash Function for Zero-Knowledge Proof Systems"* [14], come parte di una famiglia di hash ottimizzate per circuiti aritmetici. Nel 2023, gli stessi autori hanno proposto una versione aggiornata, Poseidon2 [15], che migliora ulteriormente le prestazioni.

Poseidon Hash è costruita come una funzione a spugna (*sponge function*) che utilizza la permutazione POSEIDON$^{\pi}$, una variante della strategia HadesMiMC. Mappa stringhe su un campo finito $\mathbb{F}_p$ (dove p è un numero primo approssimativamente $2^n > 2^{31}$) in stringhe a lunghezza fissa sullo stesso campo. La permutazione POSEIDON$^{\pi}$ si basa su una struttura SPN (*Substitution-Permutation Network*), combinando strati di S-box non lineari (ad esempio x^{α}, con $\alpha = 5$ o 3) e strati lineari (moltiplicazioni con matrici MDS). Questa struttura è ottimizzata per i sistemi R1CS (*Rank-1 Constraint Systems*), comunemente usati nei circuiti SNARKs, riducendo significativamente il numero di vincoli rispetto ad altre funzioni come Pedersen Hash e MiMC.

Il paper originale sottolinea un vantaggio chiave:

> "La nostra funzione di hash POSEIDON utilizza fino a 8 volte meno vincoli per bit di messaggio rispetto a Pedersen Hash."

Questa efficienza è cruciale per applicazioni ZK, dove il numero di vincoli determina la complessità computazionale della generazione e verifica delle prove.

Poseidon Hash è suggerita per tutte le applicazioni che richiedono funzioni di hashing compatibili con ZK, in particolare:

1. **Commitments:** Per protocolli in cui si dimostra la conoscenza di un valore impegnato senza rivelarlo. Si consiglia l'uso di Poseidon-128 con permutazioni a chiamata singola e larghezze (*width*) da 2 a 5 elementi di campo. Rispetto a Pedersen Hash, Poseidon è più veloce e può essere utilizzato anche in schemi di firma, riducendo l'impronta del codice.

2. **Hashing di Oggetti Multipli:** Per codificare campi multipli come elementi di campo e dimostrarne proprietà in ZK. Si suggerisce l'uso di una spugna a lunghezza variabile con Poseidon-128 o Poseidon-80 e larghezza 5 (con *rate* 4).

3. **Alberi di Merkle:** Per consentire prove ZK della conoscenza di una foglia in un albero, con eventuali affermazioni sul contenuto della foglia. Si raccomanda un'arità 4 (larghezza 5) con Poseidon-128 per le migliori prestazioni, sebbene siano supportate arità più convenzionali (es. 2 o 8).

Nei SNARKs, il campo primo è spesso il campo scalare di una curva ellittica *pairing-friendly*. La permutazione POSEIDON$^{\pi}$ può essere rappresentata come un circuito con un numero relativamente basso di *gate*, ma i suoi parametri devono essere adattati al valore di p. In particolare, dopo aver fissato p, si verifica se x^{α} è invertibile in $\mathbb{F}_p$, condizione soddisfatta se $p \mod \alpha \neq 1$.

Poseidon Hash si distingue per la sua efficienza rispetto a Pedersen Hash e MiMC, due funzioni di hashing usate in contesti ZK. La tabella seguente riporta il numero di vincoli R1CS necessari per dimostrare la conoscenza di una foglia in un albero di Merkle con 2^{30} elementi, confrontando Poseidon-128 con Pedersen Hash e MiMC-2p/p (Feistel):

Poseidon-128				
Arità	Larghezza	R_F	R_P	Vincoli Totali
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: Confronto dei vincoli R1CS per un albero di Merkle con 2^{30} elementi.

- **Poseidon-128:** Con arità 4:1 (larghezza 5), richiede solo 4500 vincoli totali, grazie a un numero ridotto di round completi ($R_F = 8$) e parziali ($R_P = 60$).
- **Pedersen Hash:** Richiede 41.400 vincoli, un valore significativamente più alto, dovuto alla sua struttura basata su logaritmi discreti, meno adatta ai circuiti aritmetici.
- **MiMC-2p/p:** Con 19.440 vincoli, è più efficiente di Pedersen ma meno di Poseidon, a causa del maggior numero di round (324) necessari per garantire la sicurezza.

5.2.2 Implementazione

Per implementare i contratti su Sepolia Base, è stato utilizzato il codice fornito da Chih Cheng Liang, uno sviluppatore della Ethereum Foundation, che ha reso pubblica la sua versione (`https://github.com/ChihChengLiang/poseidon-tornado`)

Posiedon Hash viene utilizzato al posto di MiMC per fare l'hash delle foglie e al posto del Pedersen Hash per fare il commitment.

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

Il commitment non utilizza più il valore secret ma solamente il nullifier:
`commitment = PoseidonHash(nullifier, 0)`
Per generare prova e witness utilizziamo il nullifierHash, calcolato nella seguente maniera:
`nullifierHash = PoseidonHash(nullifier, 1, leafIndex)`
I dati forniti da Chih Cheng Liang sono i seguenti:
Per compilare il progetto bisogna avere il seguenti programmi installati:

- Node.js (versione non recente, per questo progetto è stata usata v16.20.2)
- Circom versione 2.0.2

Una volta scaricata la repository i comandi da eseguire sono i seguenti:

```
npm install
npm run build
npm test
```

Una volta eseguita la build correttamente, si sarà creata una cartella artifact contenente il file .json dei contratti compilati.

Un problema che è stato riscontrato nel fare il deploying dei contratti su testnet è che il contratto Hasher.json aveva bytecode vuoto. Quindi è stato compilato il contratto sfruttando `genContract.js` di circolibjs con il seguente codice:

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

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

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

  const poseidonContract = await circomlibjs.
    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 circomlibjs.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);
});
```

Una volta ottenuti Verifier.json, Hasher.json ed ETHTornado.json, bisogna per deployare i contratti.

Il contratto con cui si andrà ad interagire per i depositi ed i withdrawal sarà quello generato dal deployment di ETHTornado.json. La maniera con cui i tre contratti sono collegati, avviene col deployment di quest'ultimo, il quale costruttore prende come input l'indirizzo dei due contratti (Hasher e Verifier).

I contratti sono stati deployati grazie alla libreria web3, che ci permette di collegarci ad una rpc e mandare transazioni alla rete, tramite il seguente script:

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

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

```
    publicnode.com');

// Account setup
const privateKey = 'REDACTED';
const hasherAddress = '0x0ffc6149238E8c153e5ad30E71f37e50B9C87caC';
const verifierAddress = '0x9A25Fb8207086912ee3af781277e14767f81BD71';
const account = web3.eth.accounts.privateKeyToAccount(privateKey);
web3.eth.accounts.wallet.add(account);

// Load contract data
const contractData = JSON.parse(fs.readFileSync('../artifacts/contracts/ETHTornado.sol/ETHTornado.json', 'utf8'));
const contractABI = contractData.abi;
const contractBytecode = contractData.bytecode;

const contract = new web3.eth.Contract(contractABI);

async function deployContract() {
    try {
        // Network info
        const chainId = await web3.eth.getChainId();
        console.log('Chain ID:', chainId); // Should be 84532

        // Account info
        const balance = await web3.eth.getBalance(account.address);
        console.log('Account:', account.address);
        console.log('Balance:', web3.utils.fromWei(balance, 'ether'), 'ETH');
        if (balance === '0') throw new Error('Account has no funds');

        // Check hasher address
        const hasherCode = await web3.eth.getCode(hasherAddress);
        console.log('Hasher code length:', hasherCode.length);
        if (hasherCode === '0x') throw new Error('Hasher address is not a deployed contract');

        // 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();
46          console.log('Gas price:', web3.utils.fromWei(
    gasPrice, 'gwei'), 'gwei');
47
48          // Deploy
49          const deployedContract = await contract
50              .deploy({
51                  data: 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) => {
63                  console.log('Tx receipt:', receipt);
64              })
65              .on('error', (error) => {
66                  console.error('Tx error:', error);
67              });
68
69          console.log('Deployed at:', deployedContract.
    options.address);
70          return deployedContract;
71      } catch (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();
```

Il costruttore di ETHTornado.json prende in input 4 parametri, in ordine:

- **_verifier** L'indirizzo a cui è stato deployato il contratto Verifier
- **_denomination** Rappresenta il valore fisso dei depositi e prelievi, nel nostro caso è 1000000000000000 wei (0.001ETH)
- **_merkleTreeHeight** L'altezza del Merkle Tree, nel nostro caso 20
- **_hasher** L'indirizzo a cui è stato deployato il contratto Hasher

Per il deploying di Hasher e Verifier è stato usato lo stesso script ma con arguments la lista vuota "[ ]" e con input Verifier.json e l'Hasher.json generato con lo script precedente.

Per controllare transazioni ed indirizzi su Sepolia Base si può utilizzare il seguente sito: `https://sepolia.basescan.org`[2]

I contratti sono stati deployati ai seguenti indirizzi:

- **Hasher:** 0x0ffc6149238E8c153e5ad30E71f37e50B9C87caC
- **Verifier:** 0x9A25Fb8207086912ee3af781277e14767f81BD71
- **ETHTornado** 0x3a6d3B0F0bA61d2Ca2e56129E522982fFf0a545e

Vediamo ora un esempio di transazione:

5.2.3 Deposito

Script:

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

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

// Inizializzazione Web3
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 // Configurazione account
17 const account = web3.eth.accounts.privateKeyToAccount
       ('0x' + PRIVATE_KEY);
18 web3.eth.accounts.wallet.add(account);
19 web3.eth.defaultAccount = account.address;
20 const senderAccount = account.address;
21
22 function toHex(number, length = 32) {
23     const str = number instanceof Buffer ? number.
       toString('hex') : BigInt(number).toString(16);
24     return '0x' + str.padStart(length * 2, '0');
25 }
26
27 // Funzione Poseidon Hash che restituisce un bytes32
28 function poseidonHash(poseidon, inputs) {
29     const hash = poseidon(inputs.map((x) => BigNumber.
       from(x).toBigInt()));
30     const hashStr = poseidon.F.toString(hash); //
       Converte nel campo finito
31     const hashHex = BigNumber.from(hashStr).
       toHexString(); // Converti in esadecimale
32     return ethers.utils.hexZeroPad(hashHex, 32); //
       Padding a 32 byte
33 }
34
35 async function createDeposit(amount) {
36     // Inizializza Poseidon
37     const poseidon = await circomlib.buildPoseidon();
38
39     // Genera un nullifier casuale (15 byte)
40     const nullifierBytes = ethers.utils.randomBytes(15
       );
41     const nullifierHex = ethers.utils.hexlify(
       nullifierBytes);
42
43     // Calcola il commitment con [nullifier, 0]
44     const commitment = poseidonHash(poseidon, [
       nullifierHex, 0]);
45
46     // Restituisci in formato JSON
47     return {
48         nullifier: ethers.utils.hexZeroPad(
       nullifierHex, 32), // Nullifier come bytes32
49         commitment: commitment
50     };
```

```
51 }
52
53 /**
54  * Effettua un deposito
55  * @param {Object} params - Parametri del deposito
56  * @param {string} params.currency - Valuta (es. "eth
      ")
57  * @param {number} params.amount - Importo del
      deposito
58  */
59 async function deposit({ currency, amount }) {
60     try {
61         // Controlla il saldo
62         const balance = await web3.eth.getBalance(
      senderAccount);
63         console.log('Saldo account:', web3.utils.
      fromWei(balance, 'ether'), 'ETH');
64
65         // Verifica che il contratto esista
66         const code = await web3.eth.getCode(
      tornadoAddress);
67         if (code === '0x') {
68             throw new Error('Nessun contratto
      deployato all\'indirizzo specificato!');
69         }
70
71         const deposit = await createDeposit({ amount
      });
72
73         // Calcola il valore da inviare (amount in wei
      )
74         const value = web3.utils.toWei(amount.toString
      (), 'ether');
75         console.log('Valore da inviare:', web3.utils.
      fromWei(value, 'ether'), 'ETH');
76
77         const commitment = deposit.commitment
78
79         // Stima il gas
80         const gasEstimate = await tornado.methods.
      deposit(commitment).estimateGas({
81             from: senderAccount,
82             value: value
83         });
84         console.log('Gas stimato:', gasEstimate);
85
```

```
86          // Ottieni il gas price
87          const gasPrice = await web3.eth.getGasPrice();
88          console.log('Prezzo del gas:', web3.utils.
    fromWei(gasPrice, 'gwei'), 'gwei');
89
90          // Verifica fondi sufficienti
91          const totalCost = BigInt(Number(gasEstimate))
    * BigInt(gasPrice) + BigInt(value);
92          if (BigInt(balance) < totalCost) {
93              throw new Error('Fondi insufficienti per
    il deposito e il gas!');
94          }
95
96          // Esegui la transazione
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('Transazione completata! Hash:',
    receipt.transactionHash);
106         const data = receipt.logs[0].data;
107         const firstHalf = '0x' + data.slice(2, 66);
108         const leafIndex = parseInt(firstHalf);
109         console.log('leafIndex:', leafIndex);
110         const poseidon = await circomlib.buildPoseidon
    ();
111         const nullifierHash = poseidonHash(poseidon, [
    deposit.nullifier, 1, leafIndex]);
112         const nullifierHex = deposit.nullifier;
113         return { nullifierHex, commitment,
    nullifierHash };
114     } catch (error) {
115         console.error('Errore durante il deposito:',
    error);
116         if (error.receipt) {
117             console.error('Dettagli receipt:', error.
    receipt);
118         }
119         throw error;
120     }
121 }
```

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

Output:

```
Saldo account: 0.083258091675057736 ETH

Valore da inviare: 0.001 ETH

Gas stimato: 828077n

Prezzo del gas: 0.001000319 gwei

Submitting deposit transaction

Transazione completata!

Hash:
↪ 0xf486e0c92708ef5794d851bc6698cafb7f40ec338b72907f053da7d1d8fe82d1

leafIndex: 11

nullifier:
↪ 0x00000000000000000000000000000001a34f8546195f074f9e7380446ff3a

commitment:
↪ 0x0b6014c902cd1e73dde4d951cc276fe7ee574fd39784a75a440911ae61c36f22

nullifierHash:
↪ 0x26f3b5d6175b61acd648a7c095b0c480eebe60297aab12b08980a5265da467b3
```

5.2.4 Generazione prova

Per scrivere questo script è stato necessario compilare il file `/src/MerkleTree.ts` in un file javascript tramite il comando `npx tsc`. Ottenendo il file `/dist/src/merkleTree.js`

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

```
const circomlib = 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 = "0x3a6d3B0F0bA61d2Ca2e56129E522982fFf0a545e";

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]);
```

```
41    }
42  }
43
44  // Fetch events in batches
45  async function getEventsInBatches(fromBlock, toBlock,
        batchSize = 49999n) {
46    const events = [];
47    const latestBlock = toBlock === 'latest' ? await web
        3.eth.getBlockNumber() : BigInt(toBlock);
48    for (let start = BigInt(fromBlock); start <=
        latestBlock; start += batchSize) {
49      const end = latestBlock < start + batchSize - 1n ?
        latestBlock : start + batchSize - 1n;
50      console.log(`Fetching events from block ${start}
        to ${end}`);
51      const batch = await tornado.getPastEvents('Deposit
        ', {
52        fromBlock: start.toString(),
53        toBlock: end.toString(),
54      });
55      events.push(...batch);
56    }
57    return events;
58  }
59
60  // Genera la prova Merkle
61  async function generateMerkleProof(deposit) {
62    console.log('Getting current state from tornado
        contract');
63    const events = await getEventsInBatches(22237544, '
        latest');
64
65    const leaves = events
66      .sort((a, b) => {
67        const indexA = BigInt(a.returnValues.leafIndex);
68        const indexB = BigInt(b.returnValues.leafIndex);
69        return indexA < indexB ? -1 : indexA > indexB ?
        1 : 0;
70      })
71      .map(e => e.returnValues.commitment);
72
73    if (!leaves.includes(deposit.commitment)) {
74      throw new Error(`Deposit commitment ${deposit.
        commitment} not found in the fetched leaves`);
75    }
76
```

```
77    const poseidon = await circomlib.buildPoseidon();
78    const tree = new MerkleTree(MERKLE_TREE_HEIGHT, '
       tornado', new PoseidonHasher(poseidon));
79
80    console.log('Inserting leaves into Merkle Tree...');
81    for (const leaf of leaves) {
82      await tree.insert(leaf);
83    }
84    console.log('All leaves inserted');
85
86    const leafIndex = leaves.indexOf(deposit.commitment)
       ;
87    console.log('Adjusted leafIndex:', leafIndex);
88    assert(leafIndex >= 0, 'The deposit is not found in
       the tree');
89
90    const root = await tree.root();
91    const rootHex = toHex(root);
92    const isValidRoot = await tornado.methods.
       isKnownRoot(rootHex).call();
93    const isSpent = await tornado.methods.isSpent(
       deposit.nullifierHash).call();
94
95    console.log('Root:', rootHex);
96    console.log('Is valid root:', isValidRoot);
97    console.log('Is spent:', isSpent);
98
99    assert(isValidRoot === true, 'Merkle tree is
       corrupted');
100   assert(isSpent === false, 'The note is already spent
       ');
101
102   const { path_elements: pathElements, path_index:
       pathIndices } = await tree.path(leafIndex);
103
104   return { root, pathElements, pathIndices };
105 }
106
107 // Genera la prova SNARK
108 async function prove(witness) {
109   const wasmPath = path.join(__dirname, '../build/
       withdraw_js/withdraw.wasm');
110   const zkeyPath = path.join(__dirname, '../build/
       circuit_final.zkey');
111
112   const { proof } = await groth16.fullProve(witness,
```

```
    wasmPath, zkeyPath);
  const solProof = {
    a: [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;
}

// Genera la prova completa per il ritiro
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,
    recipient,
    relayer,
    fee,
    nullifier: BigNumber.from(deposit.nullifier).
    toBigInt(),
    pathElements,
    pathIndices,
  };

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

// Dati di input
const deposit = {
  "nullifier": "0x0000000000000000000000000000000001a
    34f8546195f074f9e7380446ff3a",
  "commitment": "0x0b6014c902cd1e73dde4d951cc276fe7ee5
    74fd39784a75a440911ae61c36f22",
  "nullifierHash": "0x26f3b5d6175b61acd648a7c095b0c480
    eebe60297aab12b08980a5265da467b3"
};

const recipient = "0x58102161341811da3009e257618b1cc5c
```

```
    2c464c0";

// Esecuzione
(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 valid root: 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');

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

// Inizializzazione Web3
const web3 = new Web3(rpc);
const contractJson = require(__dirname + '/../
   artifacts/contracts/ETHTornado.sol/ETHTornado.json
   ');
const tornado = new web3.eth.Contract(contractJson.abi
   , tornadoAddress);

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

/**
 * Esegue un prelievo ETH
 * @param {Object} params - Parametri del prelievo
 * @param {string} params.proof - Stringa esadecimale
   della prova zk-SNARK
```

```
23  * @param {Array} params.args - Array di parametri
       aggiuntivi [root, nullifierHash, recipient, relayer
       , fee, refund]
24  * @param {string} [params.refund='0'] - Importo di
       rimborso in wei
25  */
26 async function withdraw(proof, root, nullifierHash,
       recipient, relayer, fee = '0', refund = '0') {
27     try {
28         // Controlla il saldo
29         const balance = await web3.eth.getBalance(
       senderAccount);
30         console.log('Saldo account:', web3.utils.
       fromWei(balance, 'ether'), 'ETH');
31 
32         // Verifica che il contratto esista
33         const code = await web3.eth.getCode(
       tornadoAddress);
34         if (code === '0x') {
35             throw new Error('Nessun contratto
       deployato all\'indirizzo specificato!');
36         }
37 
38         const isKnownRoot = await tornado.methods.
       isKnownRoot(root).call();
39         console.log('Root conosciuta:', isKnownRoot);
40         if (!isKnownRoot) {
41             console.warn('Attenzione: la root fornita
       non e presente nel contratto!');
42         }
43 
44         const isSpent = await tornado.methods.isSpent(
       nullifierHash).call();
45         console.log('Nullifier gia speso:', isSpent);
46         if (isSpent) {
47             throw new Error('Il nullifier e gia stato
       speso!');
48         }
49 
50         // Stima il gas
51         const gasEstimate = await tornado.methods.
       withdraw(proof, root, nullifierHash, recipient,
       relayer, fee).estimateGas({
52             from: senderAccount,
53             value: refund
54         });
```

```
55          console.log('Gas stimato:', gasEstimate);
56
57          // Ottieni il gas price
58          const gasPrice = await web3.eth.getGasPrice();
59          console.log('Prezzo del gas:', web3.utils.
    fromWei(gasPrice, 'gwei'), 'gwei');
60
61          // Verifica fondi sufficienti
62          const totalCost = BigInt(Number(gasEstimate))
    * BigInt(gasPrice) + BigInt(refund);
63          if (BigInt(balance) < totalCost) {
64              throw new Error('Fondi insufficienti per
    il deposito e il gas!');
65          }
66
67          // Esegui la transazione
68          console.log('Submitting withdraw transaction')
    ;
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) => {
76                  console.log(`The transaction hash is $
    {txHash}`);
77              })
78              .on('error', (e) => {
79                  console.error('Errore durante la
    transazione:', e.message);
80              });
81
82          console.log('Prelievo completato!');
83      } catch (error) {
84          console.error('Errore nel prelievo:', error);
85          if (error.receipt) {
86              console.error('Dettagli receipt:', 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         "385753582699743675900730658687621927769857444762047084661631313811160781351",
102         "136007055858983680674936401813258059259366063139568776565338093868123461791 26"
103       ],
104       [
105         "286291399112933921094110608801327805760669739930619569688598080090591621330 9",
106         "629540174021879776695544924447757860906997915941202300723818192503965672893 5"
107       ]
108     ],
109     "c": [
110       "251418163797426623817522358818469583028711133743449111155170757039028432309 9",
111       "353639289207320172623836557078422853042298412319354688206743516448699894292 8"
112     ]
113 }
114
115 const root = "0x16216c3050e4c744e0a057779a4cc933999830a439ed091e2f518e2c43b9bb62";
116
117 const nullifierHash = "0x26f3b5d6175b61acd648a7c095b0c480eebe60297aab12b08980a5265da467b3"
118
119 const recipient = "0x58102161341811da3009e257618b1cc5c2c464c0";
120
121 const relayer = "0x0000000000000000000000000000000000000000";
122
123 // Esegui il prelievo
124 withdraw(proof, root, nullifierHash, recipient, relayer)
```

```
125 .then(() => console.log('Prelievo eseguito con
        successo'))
126 .catch(err => console.error('Errore finale:', err));
```

Output:

```
Saldo account: 0.007567958903663304 ETH

Root conosciuta: true

Nullifier già speso: false

Gas stimato: 315590n

Prezzo del gas: 0.00100032 gwei

Submitting withdraw transaction

The transaction hash is

0xef290555754da1fe69539bbc2c66f1d2ab1b080db26ad045c7a40b387a8ed997

Prelievo completato!

Prelievo eseguito con successo
```

5.2.6 Commenti

Indirizzo che ha effettuato il deposito: 0x75D610b953C60f9950be94a5cE82C8DC8bA41CC2

Indirizzo che ha effettuato il prelievo: 0x58102161341811dA3009e257618b1CC5C2c464C0

Nella variabile `PRIVATE_KEY` sono state quindi inserite le rispettive chiavi private.

Non è stato utilizzato un relayer ma il prelievo è stato ritirato direttamente dell'indirizzo ricevente

Le transazioni su Sepolia Test si possono verificare ai seguenti link:

Deposit: 0xf486e0c92708ef5794d851bc6698cafb7f40ec338b72907f053da7d1d8fe82d1

Withdraw: 0xef290555754da1fe69539bbc2c66f1d2ab1b080db26ad045c7a40b387a8ed997

Aggiunte al Protocollo di Base

6.1 Relayer

Quando un utente effettua un prelievo su Tornado Cash, deve interagire direttamente con lo smart contract del protocollo, il che comporta costi di transazione in termini di **gas fees**. Ciò rappresenta spesso un problema, poiché l'obiettivo principale di Tornado Cash è garantire l'anonimato delle transazioni. Di conseguenza, gli account destinatari dei fondi sono frequentemente privi di Ether (ETH), in quanto l'acquisto di ETH tramite un exchange e il successivo trasferimento su un nuovo indirizzo potrebbe comportare l'identificazione dell'utente.

Per risolvere questo problema, entrano in gioco i **relayer**, soggetti terzi che agiscono come intermediari e si offrono di coprire le gas fees in cambio di una piccola tariffa (che varia, ad es da 0.45% a 2.5% del prelievo, attualmente).

Quando un utente desidera effettuare un **prelievo** utilizzando un relayer, genera una **prova crittografica (proof)** che, oltre a includere i dati del deposito, incorpora anche l'indirizzo del relayer stesso. Invece di inviare direttamente la prova allo smart contract, l'utente la trasmette al relayer *off-chain*, il quale si occupa di firmare la transazione e coprirne i costi di esecuzione.

Grazie al fatto che la prova del **prelievo** viene generata specificando **sia l'indirizzo del relayer sia quello del wallet destinatario**, il relayer non ha alcuna possibilità di alterare il comportamento della transazione a proprio vantaggio. L'unica azione che può intraprendere è non eseguire la transazione. In tal caso, l'utente può semplicemente selezionare un altro relayer e generare una nuova prova utilizzando i dati del nuovo intermediario.

Un Relayer può diventarlo depositando minimo 5k TORN in stake. Piu TORN mette in stake, più alto sarà il suo "score", più alta è la sua tariffa, più basso sarà lo score.

6.2 Anonymity Mining

L'efficacia del protocollo Tornado Cash dipende fortemente dalla dimensione del suo **anonymity set**, ovvero il numero di utenti che partecipano al sistema.

Se il protocollo fosse utilizzato da un solo individuo, il meccanismo di anonimizzazione risulterebbe completamente inutile. In tal caso, sarebbe sufficiente osservare le transazioni registrate sulla blockchain per individuare un indirizzo che ha effettuato un deposito e uno che ha eseguito un prelievo, rendendo banale l'associazione tra le due operazioni.

Anche in presenza di un numero ridotto di utenti, sarebbe possibile condurre un'analisi probabilistica per stabilire correlazioni tra depositi e prelievi. Se l'anonimity set è limitato, il numero di potenziali collegamenti tra depositi e prelievi si riduce, permettendo a un osservatore di inferire, con un certo grado di certezza, quali transazioni siano collegate tra loro.

Un ulteriore fattore critico è la liquidità all'interno della pool. Anche con un numero elevato di utenti, se i depositanti prelevassero gli asset subito dopo il deposito, il saldo oscillerebbe tra zero e l'importo più recente, azzerandosi rapidamente dopo ogni prelievo. Questo renderebbe più facile collegare le transazioni, poiché l'osservazione delle variazioni del saldo permetterebbe di dedurre quando un deposito viene riscattato.

Per mitigare questi problemi, è stato introdotto il meccanismo di **Anonymity Mining**, il cui funzionamento è piuttosto semplice: un utente che deposita asset nella pool riceve una ricompensa in token TORN proporzionale al tempo di permanenza dei fondi nel protocollo. Questa ricompensa, calcolata tramite Punti di Anonimato (AP) guadagnati per blocco in base alla dimensione del deposito (es. 4 AP per 0,1 ETH o 400 AP per 100 ETH), serve per aumentare l'anonimity set, mascherare più efficacemente i pattern temporali e rendere significativamente più difficile correlare depositi e prelievi, incentivando così una partecipazione più ampia e duratura.

Un aspetto fondamentale del protocollo è che non è pubblicamente visibile se un determinato deposito è stato prelevato o meno. Questo garantisce che non sia possibile stabilire un collegamento diretto tra un deposito e un prelievo, rafforzando il livello di anonimato dell'intero sistema.

Il programma di Anonymity Mining è terminato nel 2021, visto che Tornado Cash aveva già abbastanza volume.

6.3 Tornado Cash Nova

Supponiamo che un utente depositi 0.423489389 ETH, la probabilità che un altro utente depositi la stessa cifra, scegliendo in modo casuale, è praticamente nulla. E quindi da un prelievo si riuscirebbe a risalire al deposito semplicemente dal valore. Per questo motivo è stato scelto di utilizzare degli importi standard per depositi e prelievi, così da eliminare questo tipo di vulnerabilità. Per quanto riguarda ETH i valori standard sono 0.1, 1, 10, 100.

Questa soluzione però introduce della rigidità, in quanto ora per depositare gli ipotetici 0.423489389 ETH avremmo bisogno di fare 4 transazioni di deposito e 4 di prelievo, andando a pagare il gas 8 volte al posto di 2 e lasciando 0.023489389 ETH sul wallet di partenza.

Tornado Cash Nova nasce con l'idea di risolvere questo problema.

Per superare questa limitazione, Tornado Cash Nova introduce il concetto di **shielded account** e la possibilità di depositare e prelevare importi completamente arbitrari. A differenza di Tornado Cash Classic, dove gli importi sono fissi, Nova permette agli utenti di depositare **qualsiasi cifra desiderata** e di effettuare prelievi multipli fino a esaurire il saldo.

Questo sistema offre maggiore flessibilità, ma richiede un'attenzione particolare per mantenere l'anonimato. Se un utente deposita e preleva la stessa cifra esatta, potrebbe essere più facilmente rintracciabile. Tuttavia, dividere il prelievo in più transazioni con importi differenti può rendere più complesso il collegamento tra il deposito e i prelievi, soprattutto in una pool molto utilizzata.

Dal punto di vista tecnico, Tornado Cash Nova utilizza un sistema di **commitment aggiornabile** basato su Merkle Trees e zk-SNARKs. Ogni transazione modifica lo stato del saldo dell'utente senza rivelare esplicitamente gli importi coinvolti. Questo avviene attraverso la generazione di un nuovo **commitment** che rappresenta il saldo residuo dopo un prelievo parziale. Il nuovo commitment viene aggiunto al Merkle tree, mentre il vecchio viene escluso dalle future verifiche tramite il **nullifier**, impedendo un eventuale double spending.

Tornado Cash Nova non è un aggiornamento di Tornado Cash Classic, ma una versione alternativa con caratteristiche diverse. Mentre Nova consente maggiore flessibilità negli importi, Classic garantisce un'anonimizzazione più semplice grazie agli importi standardizzati. La scelta tra le due dipende dalle esigenze specifiche dell'utente in termini di privacy e usabilità.

6.4 Tornado Proxy

Quando si accede ai contratti di deposito di Tornado.cash tramite l'interfaccia ufficiale, tutte le transazioni vengono eseguite attraverso un contratto proxy denominato Tornado Proxy. Poiché i contratti di deposito sono immutabili e molte funzionalità di Tornado.cash sono state aggiunte solo dopo il loro deployment iniziale, il Tornado Proxy consente di integrare funzionalità aggiuntive senza dover sostituire le istanze originali dei contratti di deposito, già collaudate e affidabili.

Le due funzionalità più rilevanti del Tornado Proxy sono:

- Backup on-chain dei depositi degli utenti, attraverso account di note crittografate, garantendo così una maggiore sicurezza e recuperabilità delle transazioni.
- Gestione della coda di depositi e prelievi, facilitando il loro successivo processamento all'interno del contratto Tornado Trees per una verifica efficiente tramite prove a conoscenza zero.

Quando si effettua un deposito tramite il Tornado Proxy, e successivamente un prelievo attraverso lo stesso, il proxy chiama i metodi corrispondenti nel contratto Tornado Trees.

La registrazione di un deposito prevede i seguenti passaggi:

1. Si acquisiscono l'indirizzo del contratto di deposito, il commitment e il numero del blocco corrente.

2. Questi valori vengono codificati tramite ABI e sottoposti a un hashing keccak256.

3. L'hash risultante viene inserito in una coda interna al contratto, per essere successivamente raggruppato in un Merkle Tree dei depositi (da non confondere con il contratto di deposito stesso).

La registrazione di un prelievo segue lo stesso procedimento, con la differenza che al posto dell'commitment viene utilizzato l'hash del nullifier associato al prelievo. L'hash keccak256 risultante viene quindi inserito nella coda dei prelievi, per essere successivamente raggruppato nel Merkle Tree dei prelievi.

I metodi registerDeposit e registerWithdrawal del contratto Tornado Trees emettono rispettivamente gli eventi DepositData e WithdrawalData, contenenti gli stessi valori utilizzati per calcolare l'hash, oltre a un campo aggiuntivo che indica l'ordine di ingresso nella coda.

6.4.1 Aggiornamento Chunked Merkle Tree

I Merkle Tree standard sono costosi da memorizzare e aggiornare, soprattutto quando si vuole gestire un elevato numero di foglie. Depositare una nota nei contratti di deposito di Tornado.cash può costare oltre 1,2 milioni di gas, traducendosi in centinaia di dollari in ETH se l'operazione avviene sulla mainnet di Ethereum. Gran parte di questo costo deriva dall'inserimento di un singolo impegno nel Merkle Tree del contratto di deposito.

Invece di spendere tutto quel gas, si potrebbe semplicemente proporre una nuova radice di Merkle, calcolata off-chain, e dimostrarne la validità tramite una Zero Knowledge Proof.

Tuttavia, verificare le Zero Knowledge Proofs è un'operazione onerosa. Per ridurre i costi, anziché aggiornare il Merkle Tree a ogni modifica, è possibile raggruppare più inserimenti in commitment aggregati, verificabili in un'unica operazione.

Struttura Chunked Tree

Gli alberi di deposito e prelievo sono entrambi Merkle Tree di dimensione fissa, profondi 20 livelli, con una caratteristica particolare: il chunk size determina il livello a cui gli aggiornamenti vengono elaborati in modo aggregato, anziché come inserimenti singoli.

Nel caso di Tornado Trees, il chunk size è 256 (2^8), quindi ogni chunk copre 8 livelli dell'albero. L'albero completo rimane limitato a 2^{20} foglie, ma queste sono suddivise in chunk da 256 foglie ciascuno, per un totale di 2^{12} chunk.

La funzione di hash utilizzata per generare le etichette dei nodi è Poseidon, che condivide alcune somiglianze con l'hash di Pedersen impiegato nel contratto di deposito principale, in quanto si basa su un algoritmo di hashing su curve

ellittiche. La principale differenza è che Poseidon opera sulla curva BN128, mentre Pedersen utilizza Baby Jubjub. Inoltre, nelle prove a conoscenza zero, Pedersen richiede circa 1,7 vincoli per bit, mentre Poseidon ne utilizza tra 0,2 e 0,45 per bit, rendendolo significativamente più efficiente.

Eventi

Per calcolare un aggiornamento dell'albero, è necessario conoscere la sua struttura esistente. Per ottenerla, si possono interrogare i log del contratto e recuperare gli eventi DepositData o WithdrawalData emessi in precedenza, a seconda dell'albero che si sta aggiornando. Utilizzando l'indice indicato da lastProcessedDepositLeaf o lastProcessedWithdrawalLeaf, è possibile suddividere gli eventi in due insiemi di foglie: "committed" e "pending". Come suggeriscono i nomi, le foglie del primo insieme sono già state confermate all'interno del Merkle Tree, mentre quelle del secondo insieme devono ancora essere inserite.

Tree Update

Utilizzando gli eventi già confermati, è possibile ricostruire lo stato attuale del Merkle Tree calcolando l'hash Poseidon per ciascuna delle foglie esistenti. Questo viene fatto prendendo come input l'indirizzo dell'istanza Tornado, l'hash dell'impegno o del nullifier e il numero di blocco.

Lo stato iniziale del Merkle Tree prevede che ogni nodo foglia sia etichettato con un "valore zero" pari a keccak256("tornado") % BN254_FIELD_SIZE, analogamente al funzionamento dei nodi zero nel circuito di deposito principale, ma utilizzando la curva ellittica BN254. Questo approccio garantisce che tutti i percorsi dell'albero siano invalidi finché non viene inserito un impegno valido in una foglia, e fornisce un'etichetta costante e prevedibile per ciascun nodo i cui figli sono zero.

Le foglie dell'albero vengono quindi popolate da sinistra a destra con gli hash delle foglie corrispondenti agli eventi confermati. Successivamente, i nodi non foglia vengono aggiornati fino alla radice. Se il procedimento è stato eseguito correttamente, la radice risultante dovrebbe coincidere con quella attualmente memorizzata come depositRoot o withdrawalRoot nel contratto Tornado Trees. A questo punto, avendo ottenuto la "vecchia radice", è possibile selezionare un chunk di eventi in sospeso (256 elementi), calcolarne gli hash Poseidon e inserirli nell'albero. Dopo aver aggiornato i nodi non foglia fino alla radice, si otterrà la "nuova radice".

Infine, è necessario raccogliere un elenco di elementi di percorso a partire dalle foglie del sottoalbero, insieme a un array di valori 0/1 che indicano se ciascun elemento di percorso si trovi a sinistra o a destra rispetto al proprio nodo padre.

Hash degli argomenti

Infiner per poter calcolare una prova è necessario l'hash della lista di argomenti che verra passata nel circuito per ottenere la proof.

Per costruire il messaggio, è necessario concatenare i seguenti campi:

1. L'etichetta della vecchia radice
2. L'etichetta della nuova radice
3. Gli indici del percorso come intero, zero-padded a sinistra
4. Per ogni evento in attesa di venire inserito:
    - L'hash dell'commitment/nullifier
    - L'indirizzo dell'istanza Tornado (relativo alla pool)
    - Il numero del blocco

Una volta creato il messaggio, si calcola l'hash SHA-256 di questo e si applica il modulo dell'hash rispetto al modulo del gruppo BN128, che si trova nella costante SNARK_FIELD del contratto Tornado Trees.

Input per una witness di Aggiornamento dell'Albero

Il circuito Batch Tree Update prende in input l'hash SHA-256 appena calcolato ed i seguenti parametri privati:

1. La vecchia radice
2. La nuova radice
3. Gli indici del percorso come intero, zero-padded a sinistra
4. Gli elementi del percorso
5. Per ogni evento in attesa di venire inserito:
    - L'hash dell'commitment/nullifier
    - L'indirizzo dell'istanza Tornado (relativo alla pool)
    - Il numero del blocco

Prover Claim

Per dimostrare che l'aggiornamento dell'albero è stato eseguito correttamente, non è necessario fornire una dimostrazione che copra l'intera struttura. È sufficiente dimostrare che un sottoalbero di dimensione pari a un blocco di 8 livelli, contenente un elenco specifico di foglie, è stato inserito in sostituzione della "foglia zero"più a sinistra di un albero di 12 livelli.

Prova dell'Hash degli Argomenti

Invece di specificare tutti gli input pubblicamente, possiamo prendere l'Args Hash calcolato in precedenza e confrontarlo con il risultato del calcolo dello stesso hash all'interno del circuito ZK, utilizzando gli input privati. Questo rende l'esecuzione della verifica della prova molto più efficiente.

Costruzione del Sottoalbero

Prendendo i tre campi di ogni evento in sospeso nell'ordine (istanza, impegno/nullifier, numero di blocco), si calcola l'hash Poseidon di ciascuna foglia. Successivamente, si costruisce il Merkle Tree solo per il sottoalbero che si sta aggiornando. Poiché il sottoalbero è completo, non è necessario preoccuparsi di foglie zero.

Verifica dell'Inserimento del Sottoalbero

Infine, si verifica che inserire la radice del sottoalbero nella posizione proposta comporti la trasformazione della vecchia radice nella nuova radice. Questo processo funziona sostanzialmente allo stesso modo del Merkle Tree Check nel circuito di deposito principale, con la differenza che qui si utilizza Poseidon invece di MiMC.

Il Merkle Tree Updater verifica per prima cosa che il percorso specificato contenga una foglia zero calcolando quale sarebbe la radice dato gli elementi del percorso, gli indici del percorso e una foglia zero. Confronta questo risultato con la vecchia radice specificata, e quindi ripete il processo con la foglia del sottoalbero proposta, confrontando la radice risultante con la nuova radice.

6.4.2 Completamento Tree Update

Dopo aver generato witness e prova, si chiama il metodo corrispondente updateDepositTree o updateWithdrawalTree nel contratto Tornado Trees.

Si passa al metodo di aggiornamento:

- la prova
- l'hash degli argomenti
- la vecchia radice
- la nuova radice
- gli indici del percorso
- un elenco di eventi che da inserire in batch

Il metodo di aggiornamento verifica quindi che:

1. La vecchia radice specificata sia effettivamente la radice attuale dell'albero corrispondente.
2. Il percorso Merkle specificato punti alla prima foglia zero disponibile nel sottoalbero.
3. L'hash degli argomenti specificato corrisponda agli input forniti e agli eventi proposti.

4. La prova sia valida secondo il verificatore del circuito, utilizzando l'hash degli argomenti come input pubblico.

Se queste condizioni sono soddisfatte, ogni evento inserito viene eliminato dalla coda. I campi corrispondenti nel contratto, che indicano le radici attuali e precedenti, vengono aggiornati, così come un puntatore all'ultima foglia dell'evento.

Esiste un ramo speciale di logica in questo metodo che gestisce anche la migrazione degli eventi dal contratto Tornado Trees V1. Dopo la migrazione iniziale, questi percorsi non vengono mai più visitati e possono essere ignorati in sicurezza.

6.4.3 Tornado Router

Il contrato TornadoProxy 0x722122df12d4e14e13ac3b6895a86e84145b6967 è stato poi sostituito dal contratto TornadoRouter 0xd90e2f925da726b50c4ed8d0fb90ad053324f31b sostituendo quindi il punto di ingresso per accedere al protocollo.

Questo aggiornamento ha reso possibile l'introduzione di Tornado Nova e ha comportato ulteriori miglioramenti minori a livello generale.

DAO

L'ecosistema di Tornado Cash include un **token ERC-20**, **TORN**, il quale svolge una funzione di **governance**. Esso conferisce ai detentori il diritto di proporre e votare modifiche riguardanti lo sviluppo del protocollo attraverso la **DAO** (Decentralized Autonomous Organization).

La distribuzione iniziale del token TORN segue il seguente schema:

- **5% (500.000 TORN)**: distribuiti tramite **airdrop** ai primi utenti che hanno utilizzato i pool ETH di Tornado.Cash, come forma di riconoscimento per il loro supporto iniziale al protocollo.
- **10% (1.000.000 TORN)**: assegnati al programma di "**anonymity mining**" per incentivare gli utenti a mantenere i fondi nei pool, aumentando così l'anonymity set. Questi token vengono distribuiti linearmente nell'arco di un anno e, una volta esauriti, si prevede che Tornado Cash abbia raggiunto una massa critica di utenti tale da non necessitare più di questo incentivo. Il programma di anonymity mining è infatti concluso dal 2021.
- **55% (5.500.000 TORN)**: riservati al **tesoro della DAO**, con rilascio lineare in cinque anni (dopo un periodo iniziale di "cliff" di tre mesi, durante il quale i fondi rimangono bloccati). Questi fondi sono gestiti attraverso votazioni della DAO, che può decidere di utilizzarli per sviluppi futuri, audit di sicurezza, incentivi alla crescita del protocollo o altro.
- **30% (3.000.000 TORN)**: destinati agli sviluppatori **fondatori** e ai primi sostenitori del progetto, con rilascio lineare in tre anni (dopo un periodo di "cliff" di un anno, in cui i token non possono essere ancora riscattati).

7.1 Come presentare una proposta?

Per poter partecipare alla governance di Tornado.Cash, un utente deve prima mettere i suoi TORN in **stake**. Se un utente vota o crea una proposta, i token

non possono essere sbloccati fino alla fine del processo, che dura **8.25 giorni** dalla creazione della proposta. Per creare una proposta, un utente deve avere almeno **1.000 TORN** (attualmente sono circa 6500€). Ogni proposta deve essere uno smart contract il cui codice è stato verificato e che viene eseguito dallo smart contract di governance tramite la funzione **delegatecall**. Il periodo per votare una proposta è di **5 giorni**. Una proposta avrà successo se viene approvata dalla maggioranza e se vengono effettuati almeno **25.000 voti** (1 TORN = 1 voto). Dopo l'approvazione di una proposta, essa è soggetta a un periodo di **timelock** di 2 giorni. Al termine di tale periodo, qualsiasi utente può eseguire la proposta. Tuttavia, se la proposta non viene eseguita da nessun utente entro i 3 giorni successivi alla scadenza del timelock, essa viene considerata scaduta e non può più essere eseguita.

Una proposta può riguardare le seguenti operazioni:

- L'aggiunta di un nuovo pool (dove gli utenti di Tornado Cash depositano o ritirano fondi) nel **proxy**, che è uno smart contract centrale che memorizza gli indirizzi delle pool in una struttura dati modificabile, permettendo l'aggiunta di nuovi pool attraverso la governance.
- La modifica dei parametri relativi ai tassi di **ricompensa**.
- L'attivazione o disattivazione del token **TORN**.
- La modifica di alcuni contratti di mining principali, come il contratto TornadoTrees.
- Una combinazione di tutte le operazioni sopra menzionate.

7.2 Staking

Con l'approvazione della decima proposta di governance, il token TORN ha acquisito una nuova funzionalità legata allo **staking**, ampliandone l'utilità nell'ecosistema Tornado Cash.

Lo staking è un meccanismo che, oltre alla possibilità di votare proposte, incoraggia gli utenti a bloccare una quantità di TORN ricevendo in cambio ricompense proporzionali all'importo immobilizzato. Queste ricompense consistono in un quantitativo aggiuntivo di TORN, distribuito in base alla quantità stakata.

Da dove provengono queste ricompense?

Le ricompense per lo staking di TORN non derivano da un processo inflazionario (cioè dalla creazione di nuovi token), ma dalle commissioni generate dall'uso del protocollo. Questo è reso possibile dall'introduzione di un **registro decentralizzato dei relayer**, che regola l'operatività dei relayer.

I relayer sono entità che facilitano i prelievi degli utenti nel protocollo. Per essere inclusi nell'interfaccia utente di Tornado Cash, devono depositare una quantità prestabilita (minimo 5k, e una quantità superiore fa aumentare lo "score" del relayer) di TORN come stake e mantenere un saldo sufficiente per

coprire le commissioni dovute al contratto di staking. Quando un utente effettua un prelievo tramite un relayer, quest'ultimo addebita una tariffa all'utente per il servizio. Separatamente, una commissione pari allo **0,3%** del valore del prelievo (calcolata in ETH) viene convertita in TORN al prezzo di mercato corrente e detratta dallo stake del relayer. Questa quantità di TORN viene poi redistribuita agli utenti che partecipano allo staking.

Questo sistema crea un equilibrio di incentivi: i relayer guadagnano visibilità nell'interfaccia e incassano tariffe dai prelievi, mentre gli utenti che stakano TORN ricevono ricompense proporzionali, sostenute dalle commissioni pagate dai relayer al protocollo.

7.3 Partecipazione della community

La **DAO** di **Tornado Cash** ha bisogno di contributi da parte di diverse figure per continuare a crescere e svilupparsi, tra cui:

- **Sviluppatori** che proseguono e migliorano lo sviluppo del protocollo.
- **Auditori** che eseguono revisioni del codice per individuare bug e vulnerabilità.
- **Content creator** che promuovono e diffondono il protocollo.

A giugno 2021, con la proposta numero 7, la community ha deciso di creare un fondo per premiare coloro che contribuiscono allo sviluppo della DAO. I fondi sono gestiti tramite un **Multi-signature Wallet** su Gnosis Safe (uno smart contract di Ethereum che permette la creazione di multi-sig wallet). Le chiavi di questo wallet sono state distribuite a **5 utenti** scelti tramite votazione. Per validare una transazione sono necessarie **4 firme** (4-of-5). Questi utenti non decidono autonomamente, ma il loro compito è quello di dare o negare la propria firma in base alle votazioni della community. Per garantire l'onestà di queste parti, questi utenti ricevono **100 TORN** al mese dal fondo community. La community può decidere di **revocare** il ruolo di uno di questi utenti in qualsiasi momento.

Controversie e Stato Attuale

Tornado Cash, il protocollo decentralizzato per transazioni anonime su Ethereum, è al centro di un acceso dibattito tra privacy e illeciti. Nato nel 2019, garantisce anonimato mescolando le transazioni, ma è stato anche accusato di facilitare attività come il riciclaggio di denaro e le truffe.

8.1 Controversie

Secondo il *U.S. Department of the Treasury's Office of Foreign Assets Control (OFAC)*, **dal 2019 al 2022**, *Tornado Cash* ha **riciclato oltre 7 miliardi di dollari** in criptovalute[21].

Il suo picco di notorietà è arrivato nel 2021, durante l'esplosione del mercato degli *NFT* (*Non-Fungible Token*), token unici noti per prezzi esorbitanti, come il *Bored Ape Yacht Club* (da 250 USD a 1,6 milioni per i pezzi rari). Tuttavia, molti investitori alle prime armi sono stati ingannati da truffe, e i malintenzionati hanno sfruttato *Tornado Cash* per nascondere i guadagni ottenuti illegalmente.

Alcuni attacchi informatici hanno sfruttato *Tornado Cash* per riciclare fondi. Tra questi, l'attacco ad *Axie Infinity* del marzo 2022, in cui il gruppo hacker nordcoreano *Lazarus Group* ha rubato milioni di dollari di fondi, riciclandone 455 milioni tramite *Tornado Cash*. Seguono l'*Horizon Bridge Exploit* del 2022, in cui sono stati rubati 100 milioni di dollari, e il *Bitmart Hack* del 2021, che ha comportato una perdita di 196 milioni di dollari.

Il **20 maggio 2023**, un hacker anonimo – che alcuni sospettano possa essere Gozzy, uno sviluppatore associato a *Tornado Cash* – ha compromesso la *DAO di Tornado Cash* tramite una proposta malevola, camuffata da aggiornamento legittimo, contenente codice dannoso.Sfruttando il sistema di voto, l'attaccante ha ottenuto il controllo totale della governance, assegnandosi 1,2 milioni di voti fittizi: con tale potere, ha **rubato** 483.000 token TORN dai vault di governance, per un valore di circa **2,17 milioni di dollari**, parzialmente riciclati tramite il protocollo stesso. Successivamente, l'hacker ha restituito il controllo alla governance (forse soddisfatto del bottino), permettendo al saldo della DAO di coprire le perdite e restituire i fondi agli users in stake. Questo episodio, noto come il

primo grande attacco a Tornado Cash, ha rivelato le vulnerabilità nei processi di voto decentralizzati.

In un secondo incidente, nel **febbraio 2024**, un individuo noto come *Butterfly Effect*, presentato come sviluppatore della comunità, ha introdotto una proposta di governance (numero 47) contenente codice JavaScript malevolo. Questa modifica ha compromesso il protocollo, reindirizzando le note di deposito degli utenti a un server controllato dall'attaccante, mettendo a rischio i loro depositi in ETH. I sviluppatori di *Tornado Cash* hanno reagito confermando la violazione, consigliando agli utenti di ritirare e sostituire le note compromesse, e invitando i possessori di token a revocare i voti per la proposta, eliminando il codice dannoso. Questi due attacchi evidenziano i rischi intrinseci alla fiducia nelle proposte di governance decentralizzata.

8.2 Sanzioni e battaglie legali

L'**8 agosto 2022**, l'*OFAC* ha **sanzionato** *Tornado Cash*, vietandone l'uso ai cittadini USA[21]. Il dominio ufficiale è stato oscurato, e GitHub ha rimosso il repository (poi ripristinato). *Edward Snowden* ha definito questa mossa "*profondamente illiberale e autoritaria*".

Il **26 novembre 2024**, la *Corte d'Appello USA* ha stabilito che gli smart contract di *Tornado Cash* NON sono "proprietà" sanzionabili [12], e il **21 gennaio 2025** le sanzioni sono state revocate in Texas. Il processo legale per revocare completamente le sanzioni è in corso, rendendo probabile la **legalità** di *Tornado Cash* negli USA una volta concluso.

Gli sviluppatori, però, affrontano ancora processi:

- **Alexey Pertsev** è stato arrestato nei Paesi Bassi nel 2022, condannato a 5 anni e 4 mesi nel 2024 [8], ma rilasciato temporaneamente dagli arresti domiciliari l'**8 febbraio 2025**, dichiarando: "*La libertà non ha prezzo, ma la mia libertà è costata un sacco di soldi*".

- *Roman Storm* e *Roman Semenov*, altri due principali sviluppatori, sono stati accusati negli USA di riciclaggio per 1 miliardo di dollari. *Edward Snowden* li ha difesi, affermando: "*La privacy non è un crimine*".

8.3 Tra privacy e pericoli

Tornado Cash non serve solo illeciti: *Vitalik Buterin* lo ha usato per donazioni anonime in Ucraina (`https://x.com/VitalikButerin/status/1556925602233569280`), proteggendo l'identità di chi sostiene cause umanitarie in contesti sensibili. Attivisti in regimi autoritari possono ricevere fondi senza tracciamento, e individui o lavoratori pagati in criptovalute possono nascondere i propri guadagni da hacker o aziende di analisi dati. Piccole imprese, inoltre, possono mascherare transazioni per proteggere strategie commerciali da concorrenti, considerando la privacy un diritto fondamentale.

Tuttavia, il **30%** del volume immesso è illecito, [4], alimentando il dibattito tra libertà e sicurezza.

8.4 Oggi: un accesso complicato

Oggi, numerosi siti e gruppi Telegram scam proliferano. Il frontend ufficiale è accessibile solo tramite link IPFS, un sistema di archiviazione decentralizzato e resistente alla censura, come:

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

Questi link sono verificabili nel messaggio pinned del gruppo Telegram ufficiale. Siti centralizzati come `https://tornadoeth.cash` sono scam, con codice UI che ruba le note di deposito degli utenti. Alternative sicure includono l'esecuzione in locale di *tornado-cli* o l'utilizzo del frontend in locale *tornado-classic-ui*.

Rischi: gli exchange centralizzati (CEX) bloccano fondi provenienti direttamente da *Tornado Cash*, ma trasferirli attraverso più catene riduce questa probabilità. Con la revoca delle sanzioni in corso, i CEX potrebbero smettere di bloccare i fondi e persino ri-listare il token TORN, con *Coinbase*[5] tra i primi candidati, avendo supportato la causa legale. Provider RPC come *Infura*[16] bloccano le transazioni di *Tornado Cash*, ma alternative esistono per aggirare tali restrizioni. Alcuni exchange decentralizzati come Uniswap bloccano l'acquisto di TORN.

8.5 Statistiche di utilizzo

[9][20]

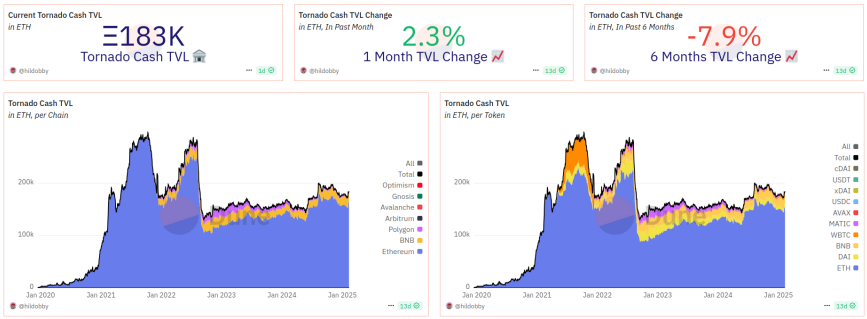


Figure 8.1: TVL di 183k ETH ∼ €456M

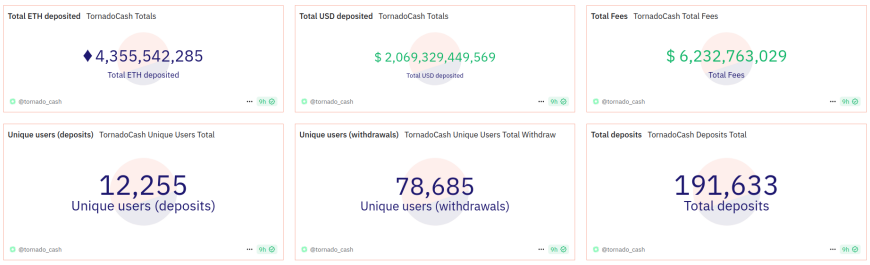


Figure 8.2: Utilizzo totale fino ad oggi

Figure 8.3: Nonostante le sanzioni, è ancora utilizzato

8.6 Community e sviluppi

La community di *Tornado Cash* è attiva. Il nuovo gruppo Telegram ufficiale è stato istituito tramite la Proposal di governance #57, proposta il **25 novembre 2024**, per contrastare i gruppi scam. Questa proposta è riconosciuta ufficialmente su *CoinMarketCap*[7] e *CoinGecko*[6], inserendo il link nell'icona di Telegram. Il vecchio account Twitter/X è inattivo, ma la DAO gestisce attivamente il protocollo. Gruppi Telegram con oltre 10k membri sono spesso scam con account inattivi comprati.

Domanda cruciale: come vietare codice decentralizzato, come gli smart contract di *Tornado Cash*, che sono eseguiti su macchine virtuali locali? Nonostante le sanzioni degli USA (in corso di revoca e rimozione), *Tornado Cash* resiste, utilizzato da chi cerca privacy in contesti sensibili.

Approfondimento: zk-SNARKs

9.1 Circuito

Un circuito nel contesto di zk-SNARK, usato in Tornado Cash, è una sequenza di vertici e archi che forma un anello chiuso. Più precisamente:

- Il vertice iniziale e il vertice finale coincidono
- Non ci sono ripetizioni di archi o vertici intermedi (eccetto per il vertice iniziale/finale)
- La sequenza è chiusa, formando una sorta di anello

In pratica, il circuito è un grafo ciclico, dove i dati fluiscono attraverso i vertici (operazioni) collegati da archi (relazioni tra variabili).

In Tornado Cash i circuiti vengono usati come struttura per la generazione di witnesses.

Circom è un linguaggio dichiarativo basato su R1CS (Rank-1 Constraint System)[3], un modello matematico per rappresentare circuiti aritmetici. Un circuito scritto in Circom è quindi una sequenza di operazioni che vengono tradotte/compilate in un sistema di vincoli matematici che possono essere utilizzati per generare prove zk-SNARKs o zk-STARKs.

Un R1CS è un modo per esprimere un calcolo o un problema aritmetico come un insieme di equazioni quadratiche di rango 1. In pratica, trasforma un programma o una relazione matematica in un sistema di vincoli che può essere verificato in modo efficiente.

Ogni vincolo è della forma: $A \cdot x \cdot B \cdot x = C \cdot x$ dove:

- x è un vettore di variabili (che include sia le variabili di input che quelle intermedie e di output del calcolo).
- A, B e C sono vettori di coefficienti (spesso rappresentati come matrici quando si considerano più vincoli). Il simbolo "$\cdot$" indica il prodotto scalare.

In questo contesto, "rango 1" significa che ogni vincolo può essere espresso come un prodotto di due espressioni lineari ($A \cdot x$ e $B \cdot x$) che, moltiplicate, devono eguagliare una terza espressione lineare ($C \cdot x$).

La compilazione del circuito avviene nella seguente maniera:

1. Un calcolo viene scomposto in una sequenza di operazioni di base.
2. Ogni operazione viene tradotta in un vincolo R1CS.
3. Il vettore x contiene una "testimonianza" (witness), ossia i valori che soddisfano tutti i vincoli contemporaneamente.

9.1.1 Generazione del witness

```
1     // Verifies that commitment that corresponds to
      given secret and nullifier is included in the
      merkle tree of deposits
2  template Withdraw(levels) {
3      signal input root;
4      signal input nullifierHash;
5      signal input recipient; // not taking part in any
      computations
6      signal input relayer;  // 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;
10     signal private input secret;
11     signal private input pathElements[levels];
12     signal private 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);
20     tree.leaf <== 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;
35      feeSquare <== fee * fee;
36      relayerSquare <== relayer * relayer;
37      refundSquare <== refund * refund;
38  }
```

Il circuito Withdraw ha lo scopo di:

1. Verificare che un certo commitment (basato su nullifier e secret) sia valido e coerente con il nullifierHash pubblico.
2. Confermare che questo commitment sia presente in un albero di Merkle con una certa radice (root).
3. Assicurare che parametri aggiuntivi (come recipient, fee, ecc.) siano vincolati nella prova.

e lo fa nella seguente maniera:

1. **Nullifier Hash Check**: il circuito prende gli input privati nullifier e secret e li passa a un componente (CommitmentHasher) che calcola due hash Pedersen:
   - Il **commitment hash**: l'hash dell'intero messaggio di commitment (che combina nullifier e secret).
   - Il **nullifier hash**: l'hash del solo nullifier.

   Verifica: Il circuito confronta il nullifier hash calcolato e il nullifierHash fornito come input pubblico, e impone che siano uguali (hasher.nullifierHash === nullifierHash)
2. **Merkle Tree Check**: il circuito usa il commitment hash (calcolato sopra), la radice pubblica dell'albero di Merkle (root), e gli input privati pathElements e pathIndices (il percorso nell'albero) per verificare che il commitment sia una foglia valida dell'albero.

- Parte dalla foglia (il commitment hash) e usa un componente chiamato Muxer per combinare il commitment con il primo elemento del percorso (pathElements[0]).
- L'input pathIndices[0] (0 o 1) indica se il commitment hash è a sinistra o a destra nel calcolo dell'hash del livello successivo, usando un hasher MiMC.
- Questo processo si ripete livello per livello, usando l'hash del livello precedente e il successivo pathElements[i], fino a calcolare un hash finale
- Il circuito verifica che questo hash finale sia uguale al Merkle root pubblico (tree.root <== root).

3. **Extra Withdrawal Parameter Check**: il circuito prende gli input pubblici recipient, relayer, fee, e refund e calcola i loro quadrati (es. recipientSquare <== recipient * recipient). Anche se questi valori non sono usati nei calcoli principali, aggiungerli al witness crea vincoli che "bloccano" i parametri della transazione. Se qualcuno cercasse di modificarli dopo la generazione della prova, i vincoli non sarebbero più soddisfatti e la prova risulterebbe invalida.

Dopo aver creato il circuito, il compito di generare il witness viene affidato alla funzione calculateWitness (`https://github.com/tornadocash/snarkjs/blob/master/src/calculateWitness.js`). Questa funzione prende in input il circuito e l'insieme di segnali di input forniti dall'utente. Il suo obiettivo è calcolare e restituire un array, che contiene i valori di tutti i segnali del circuito necessari per soddisfare i vincoli definiti.

Generare il witness significa eseguire il circuito passo per passo: assegnare gli input, calcolare i valori intermedi, e verificare che tutti i vincoli siano rispettati. Il risultato è una lista ordinata di numeri che rappresenta la "soluzione" del circuito, pronta per essere usata come base per una ZKP.

Questo witness viene poi passato a un sistema come SNARK (snarkjs) per generare una prova crittografica compatta. Tale prova può essere verificata pubblicamente da chiunque per confermare che i vincoli del circuito sono soddisfatti, senza rivelare i dati privati contenuti nel witness.

9.1.2 Calcolo della proof

Groth16 è un sistema di zero-knowledge proofs di tipo zk-SNARK. È stato sviluppato da Jens Groth nel 2016 ed è uno degli schemi più utilizzati per dimostrare che un calcolo è stato eseguito correttamente senza rivelare i dati privati usati nel calcolo.

Groth16 trasforma un circuito aritmetico in un Quadratic Arithmetic Program (QAP), che rappresenta i vincoli come polinomi. La prova dimostra che esiste un witness w che soddisfa l'equazione polinomiale del QAP, senza rivelarlo.

La prova è composta da tre elementi $\pi_A \in G_1, \pi_B \in G_2$ e $\pi_C \in G_1$ dove G_1 e G_2 sono gruppi su una curva ellittca (quella utilizzata da Tornado Cash è BN254).

Setup e QAP

Il circuito viene convertito in un QAP con polinomi $A_i(x)$, $B_i(x)$, $C_i(x)$ per ogni segnale w_i nel witness, e un polinomio divisore $Z(x)$, detto *target polynomial*. La condizione del QAP è:

$$\sum_{i=0}^{m-1} w_i A_i(x) \cdot \sum_{i=0}^{m-1} w_i B_i(x) = \sum_{i=0}^{m-1} w_i C_i(x) + H(x) \cdot Z(x) \tag{9.1}$$

dove:

- m: numero totale di segnali,
- w_i: valori del witness,
- $H(x)$: polinomio quoziente, calcolato dal prover.

Proving Key

La proving key contiene valori precalcolati:

- $\alpha, \beta, \delta \in \mathbb{F}_r$: scalari casuali,
- $[A_i]_1 = [A_i(\tau)]_1$, $[B_i]_1 = [B_i(\tau)]_1$, $[B_i]_2 = [B_i(\tau)]_2$, $[C_i]_1 = [C_i(\tau)]_1$: valutazioni dei polinomi in un punto segreto τ, mappate su G_1 e G_2,
- $[\alpha]_1$, $[\beta]_1$, $[\beta]_2$, $[\delta]_1$, $[\delta]_2$: punti generati da α, β, δ,
- $[H_i]_1$: punti per i coefficienti di $H(x)$, dove $H(x) = \sum h_i x^i$.

9.1.3 Generazione della prova

La prova combina il witness w con la proving key, aggiungendo randomizzazione per garantire zero-knowledge.

1. Calcolo iniziale dei termini base

- A:

$$A = \sum_{i=0}^{m-1} w_i [A_i(\tau)]_1 \tag{9.2}$$

  Somma dei segnali w_i moltiplicati per i punti $[A_i]_1$.

- B:
$$B_1 = \sum_{i=0}^{m-1} w_i[B_i(\tau)]_1, \quad B_2 = \sum_{i=0}^{m-1} w_i[B_i(\tau)]_2 \tag{9.3}$$
Due somme: una in G_1 (B_1) e una in G_2 (B_2).

- C:
$$C = \sum_{i=n_{pub}+1}^{m-1} w_i[C_i(\tau)]_1 \tag{9.4}$$
Somma dei segnali privati (escludendo i pubblici e il segnale "one").

- H:
$$H = \sum_{i=0}^{d-1} h_i[x^i]_1, \quad \text{dove} \quad H(x) = \frac{A(x) \cdot B(x) - C(x)}{Z(x)} \tag{9.5}$$
$H(x)$ è il polinomio quoziente, con d grado di $Z(x)$.

Si scelgono due scalari casuali $r, s \in \mathbb{F}_r$, usati per mascherare la prova.

- π_A:
$$\pi_A = A + [\alpha]_1 + r \cdot [\delta]_1 \tag{9.6}$$

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

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

9.1.4 Verifica

La verifica della prova in Groth16 controlla se una prova (π_A, π_B, π_C), generata per un *Quadratic Arithmetic Program* (QAP), soddisfa i vincoli di un circuito aritmetico senza rivelare il witness w. La verifica si basa su un'equazione di accoppiamento (*pairing*) sulle curve ellittiche G_1 e G_2, utilizzando la *verifying key* e i segnali pubblici del witness.

Equazione di verifica:

La verifica controlla l'equazione di accoppiamento:

$$e(\pi_A, \pi_B) = e([\alpha]_1, [\beta]_2) \cdot e\left(\sum_{i=0}^{n_{pub}} w_i[IC_i]_1, [\gamma]_2\right) \cdot e(\pi_C, [\delta]_2) \tag{9.9}$$

dove $e : G_1 \times G_2 \to G_T$ è una funzione di accoppiamento bilineare.

La prova è valida se l'equazione è soddisfatta, cioè se entrambi i lati sono uguali in G_T.

Calcolo della combinazione lineare vk_x:

$$vk_x = [IC_0]_1 + \sum_{i=1}^{n_{pub}} w_i [IC_i]_1 \tag{9.10}$$

vk_x rappresenta i segnali pubblici pesati dai punti della verifying key.

Verifica dell'accoppiamento

Si calcola:

- Lato sinistro: $e(-\pi_A, \pi_B)$,
- Lato destro: $e([\alpha]_1, [\beta]_2) \cdot e(vk_x, [\gamma]_2) \cdot e(\pi_C, [\delta]_2)$,

dove $-\pi_A$ è la negazione di π_A in G_1. La prova è valida se:

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

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-sanctions-challenges/`.

[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-freed-tornado-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-governments-ability-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. DOI: 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] Infura. *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-are-still-using-the-mixer.

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