Tornado Cash Privacy Solution
Version 1.4

Alexey Pertsev, Roman Semenov, Roman Storm

December 17, 2019

1 Introduction

Tornado.Cash implements an Ethereum zero-knowledge privacy solution: a smart contract that accepts transactions in Ether (in future also in ERC-20 tokens) so that the amount can be later withdrawn with no reference to the original transaction.

2 Protocol description

The protocol has the following functionality:

- Insert/deposit money to the smart contract. This can be done in a single transaction with a fixed amount (denoted by N) of Ether. The N-ETH note is called a *coin*.
- Remove/withdraw money from the smart contract can be done in 2 ways:
  - The N ETH is withdrawn through a Relayer with f Ether sent as a fee to the Relayer address t and $(N - f)$ to the designated recipient. The value f and t is chosen by the sender. In this case the withdraw transaction is initiated by the Relayer and it pays the Gas fee that is supposed to be covered by f.
  - The N ETH is withdrawn to the designated recipient, the transaction is initiated by the recipient. The recipient should have enough ETH to pay Gas fee for the transaction. In that case fee f is considered to be equal to 0.

2.1 Setup

Let $\mathbb{B} = \{0, 1\}$. Let e be the pairing operation used in SNARK proofs, which is defined over groups of prime order q.

Let $H_1 : \mathbb{B}^* \to \mathbb{Z}_p$ be a Pedersen hash function defined in [Ped]. Let $H_2 : (\mathbb{Z}_p, \mathbb{Z}_p) \to \mathbb{Z}_p$ be the MiMC hash function [AGR+16] defined as a MiMC permutation in the Feistel mode in a sponge mode of operation[1].

Let $\mathcal{T}$ be a Merkle tree of height 20, where each non-leaf node hashes its 2 children with H_2. It is initialized with all leafs being 0 values. Later the zero values are gradually replaced with other values from $\mathbb{Z}_p$. Let $O(\mathcal{T}, l)$ be the Merkle opening for leaf with index l (value of sister nodes on the way from leaf l to the root, denoted by R) in tree $\mathcal{T}$.

Let us call $k \in \mathbb{B}^{248}$ a *nullifier* and $r \in \mathbb{B}^{248}$ a *randomness*. Let us denote an Ethereum address of the coin recipient by A.

Let $\mathcal{S}[R, h, A, f, t]$ be the following statement of knowledge with public values R, h, A, f, t:

$$\mathcal{S}[R, h, A, f, t] = \{\text{I KNOW } k, r \in \mathbb{B}^{248}, l \in \mathbb{B}^{16}, O \in Z_p^{16} \text{ SUCH THAT } h = H_1(k) \\ \text{AND } O \text{ is the opening of } H_2(k||r) \text{ at position } l \text{ to } R\} \quad (1)$$

[1] https://github.com/iden3/circomlib/blob/master/src/mimcsponge_gencontract.js

where A and f are included into the context of the statement. Here h is called *nullifier hash* and $||$ is concatenation of bitstrings.

Let $\mathcal{D} = (d_p, d_v)$ be the ZK-SNARK [Gro16] proving-verifying key pair for $\mathcal{S}$ created using some trusted setup procedure. Let $\text{Prove}(d_p, \mathcal{T}, k, r, l, A, f, t) \rightarrow P$ be the proof constructor using d_p and $\text{Verify}(d_v, P, R, h, A, f, t)$ be the proof verifier.

Let $\mathcal{C}$ be the smart contract that has the following functionality:

- It stores the last $n = 100$ root values in the history array. For the latest Merkle tree $\mathcal{T}$ it also stores the values of nodes on the path from the last added leaf to the root that are necessary to compute the next root.
- It accepts payments for N ETH with data $C \in \mathbb{Z}_p$. The value C is added to the Merkle tree, the path from the last added value and the latest root is recalculated. The previous root is added to the history array.
- It verifies the alleged proof P against the submitted public values (R, h, A, f, t). If verification succeeds, the contract releases $(N - f)$ ETH to address A and fee f ETH to the Relayer address t.
- It verifies that the coin has not been withdrawn before by checking that the nullifier hash from the proof has not appeared before and if so, adds it to the list of nullifier hashes.

2.2 Deposit

To deposit a coin, a user proceeds as follows:

1. Generate two random numbers $k, r \in \mathbb{B}^{248}$ and computes $C = H_1(k||r)$
2. Send Ethereum transaction with N ETH to contract $\mathcal{C}$ with data C interpreted as an unsigned 256-bit integer. If the tree is not full, the contract accepts the transaction and adds C to the tree as a new non-zero leaf.

2.3 Withdrawal

To withdraw a coin (k, r) with position l in the tree a user proceeds as follows:

1. Select a recipient address A and fee value $f \leq N$;
2. Select a root R among the stored ones in the contract and compute opening $O(l)$ that ends with R.
3. Compute nullifier hash $h = H_1(k)$.
4. Compute proof P by calling Prove on d_p.
5. Perform the withdrawal in one of the following ways:
   - Send an Ethereum transaction to contract $\mathcal{C}$ supplying R, h, A, f, t, P in transaction data.
   - Send a request to Relayer supplying transaction data R, h, A, f, t, P. The Relayer is then supposed to make a transaction to contract $\mathcal{C}$ with supplied data.

The contract verifies the proof and uniqueness of the nullifier hash. In the successful case it sends $(N - f)$ to A and f to the Relayer t and adds h to the list of nullifier hashes.

3 Implementation

The cryptographic functions for off-chain use are implemented in the circomlib library[2]. The Solidity implementation of Merkle tree, deposit, and withdraw logic is by the authors[3]. The Solidity implementation of MiMC is by iden3[4]. The SNARK keypair and the Solidity verifier code are generated by the authors using SnarkJS. The other protocol logic (e.g., Ethereum transaction composition, SNARK proof construction calls) is by the authors[5].

[2] https://github.com/iden3/circomlib/tree/master/circuits
[3] https://github.com/tornadocash/tornado-core/tree/master/contracts
[4] https://github.com/iden3/circomlib/blob/master/src/mimcsponge_gencontract.js
[5] https://github.com/tornadocash/tornado-core/blob/master/cli.js

4 Security claims

Tornado claims the following security properties:

- Only coins deposited into the contract can be withdrawn;
- No coin can be withdrawn twice;
- Any coin can be withdrawn once if its parameters (k, r) are known unless a coin with the same k has been already deposited and withdrawn.
- If k or r is unknown, a coin can not be withdrawn. If k is unknown to the attacker, he can not prevent the one who knows (k, r) from withdrawing the coin (this includes all cases of front-running a transaction).
- The proof is binding: one can not use the same proof with a different nullifier hash, another recipient address, or a new fee amount.
- The cryptographic primitives used by Tornado have at least 126-bit security (except for the BN254 curve where the discrete logarithm problem has something like 100-bit security), and the security does not degrade because of their composition.
- For each withdrawal every deposit since the last moment when the contract has zero Ether till the formation of the root in the proof can be a potential coin, though some coins are more likely to be withdrawn depending on the user behaviour.

References

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: *ASIACRYPT (1)*. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 191–219 (cit. on p. 1).

[Gro16] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *EUROCRYPT 2016*. Vol. 9666. LNCS. Springer, 2016, pp. 305–326 (cit. on p. 2).

[Ped] *Iden3: Pedersen Hash*. https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/pedersen-hash/pedersen.html. 2019 (cit. on p. 1).