Tornado Privacy Solution
Cryptographic Review
Version 1.1

Dmitry Khovratovich and Mikhail Vladimirov
ABDK Consulting

November 29, 2019

1 Introduction

We have been approached by Tornado.Cash to review the protocol and the implementation they have designed. Tornado.Cash implements an Ethereum zero-knowledge mixer: a smart contract that accepts transactions in Ether (in future also in ERC-20 tokens) so that the amount can be later withdrawn with no reference to the original transaction.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 mixer protocol has the following functionality:

- Insert/deposit money to the mixer. 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 mixer. The N ETH is withdrawn with f Ether sent as a fee to the mixer owner and $(N-f)$ to the designated recipient. The value f is chosen by the sender.

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 [Pedb]. 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 16, 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 $\mathcal{S}[R, h, A, f]$ be the following statement of knowledge with public values R, h, A, f:

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

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

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

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 $\text{Prove}(d_p, \mathcal{T}, k, r, l, A, f) \to P$ be the proof constructor using d_p and $\text{Verify}(d_v, P, R, h, A, f)$ 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). If verification succeeds, the contract releases $(N - f)$ ETH to address A and fee f ETH to the mixer owner.
- 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 $f < N$;
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. Send an Ethereum transaction to contract $\mathcal{C}$ supplying R, h, A, f, P in transaction 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 mixer owner 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/peppersec/tornado-mixer/tree/master/contracts
[4] https://github.com/iden3/circomlib/blob/master/src/mimcsponge_gencontract.js
[5] https://github.com/peppersec/tornado-mixer/blob/master/cli.js

4 Security claims

Tornado claims the following security properties of Mixer:

- 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 Mixer 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.

5 Analysis

5.1 Hash function issues

Here we investigate the issues related to the use of hash functions in the protocol. There are two functions in use: Pedersen hash and MiMC.

Pedersen hash issues The Pedersen hash function is derived from vector Pedersen commitment $C_r(x_1, x_2, \ldots, x_m) = g_1^{x_1} g_2^{x_2} \cdots g_m^{x_m} g_{m+1}^{r}$ where g_i are generators of some group $\mathbb{G}$. It is known that Pedersen commitment is computationally binding and perfectly hiding, which makes it a good candidate for hashing (these properties have some similarity to collision and preimage resistance, respectively). It can be demonstrated that a collision for $C_r(x_1, x_2, \ldots, x_m)$ implies a discrete logarithm relation among generators, which reduces the collision resistance to the discrete logarithm complexity in the group. However, there is some subtlety in adopting Pedersen commitment to hashing arbitrary long bitstrings. Note that it is collision resistant only when all x_i and r have fixed length and do not exceed the group order.

A concrete instantiation by Iden3 and borrowed by Tornado does the following. A bitstring M is divided into segments M_i of 200 bits max. We need elliptic-curve points P_i as many as segments. Each segment is partitioned into 4-bit chunks $m_{i,j}$ interpreted as an integer in the set $[-8; 8] \setminus 0$. Then one computes [Pedb]:

$$H_1(M) = \sum_i P_i \left(\sum_{0 \le j < 50} 2^{5j} m_{i,j} \right). \tag{2}$$

where summation is the addition on the curve and multiplication is scalar. The actual implementation for the BN254 curve [Peda] deviates from [Pedb] as it uses only 10 points, so at most 2000 bits can be hashed.

Note the following properties of the resulting hash function:

1. It adds redundancy to the scalar: only 200 bits are used, whereas the BN254 group has order above 2^{253}, so 53 bits are unused.

2. It is homomorphic: if we denote by $\mathbf{P}$ the vector of generators used in H_1, then

$$H_1^{\mathbf{P}}(M_1) + H_1^{\mathbf{Q}}(M_2) = H_1^{\mathbf{P}||\mathbf{Q}}(M_1||M_2). \tag{3}$$

   This property is useful in commitments in many protocols, but for a hash function it is quite unexpected, and such a hash function should be used with great care. For example, if one computes a message authentication code (MAC) with this function as $H_1(K||M)$ where K is secret then, obviously, a MAC for $M_1||M_2$ can be easily obtained from MAC for M_1. Such a property, called length extension, has been used in many attacks[6]. It is also insecure to make several hash functions out of one by using domain separation prefix as $H_q(x) = H_1(q||x)$ as it becomes just a public addend in the formula.

3. During witdrawal the nullifier hash $H_1(k)$ is revealed. Looking at each previous deposit with commitment C, the attacker guesses that it is $H_1(k||r)$, and then he can obtain candidate $H_1'(r)$ using Equation (3).

$$H_1(k||r) = f_1(k)P_1 + (f_2(k) + f_3(r))P_2 + f_4(r)P_3; \tag{4}$$
$$H_1(k) = f_1(k)P_1 + f_2(k)P_2; \tag{5}$$
$$H_1'(r) = f_3(r)P_2 + f_4(r)P_3. \tag{6}$$

   for some arithmetic functions f_1, f_2, f_3, f_4. If r has low entropy, the attacker can check his guess and thus break the anonymity of the scheme.

4. It requires an elliptic curve arithmetic to be implemented. The current implementation of Pedersen hash used by Tornado relies on a very new elliptic curve BabyJubJub.

We found **no way to exploit these properties** in the concrete protocol by Tornado: the authors use the Pedersen hash in a proper way. For all future protocol that might rely on this one, the following steps must be taken:

- Always call Pedersen hash with same message length.
- If messages of different lengths must be used, generate a completely new set of generators for each length.

MiMC issues There is an attack on the blockcipher version of MiMC [Bon19], which does not apply to the MiMC hash function (and we do not see any way to do that).

Multiple hash functions Tornado uses two hash functions in the design, and both are used in zero-knowledge proofs. This is redundant and increases the attack surface: an attack on either design can break the entire protocol. We thus recommend using only one hash function – MiMC or more efficient variants (see Section 5.4).

Elliptic curve issues The BN254 curve used for zkSNARKs has security level of only 100 bits according to recent cryptanalytic results [BD17]. An attacker with this capacity (though clearly inexistent nowadays) can forge SNARK proofs and withdraw arbitrary amount of coins from Mixer. We recommend using a curve with higher estimated security (such as BLS12-381) but unfortunately the pairing operation for such curves has not been added to Ethereum and thus would cost a great amount of gas if implemented directly.

5.2 User mistakes

In this section we investigate mistakes that are likely to occur in an implementation or done by users. We check if the protocol is robust to some of them.

[6] For example, an attack on Flickr http://netifera.com/research/flickr_api_signature_forgery.pdf.

- Consider the nullifier value k. If k has low entropy, an attacker can guess it, then submit his own commitment with the same k but different r, and withdraw it. As a result, a nullifier hash $H_1(k)$ will be added to the list and the user will be unable to withdraw his coin. The same situation occurs if k is stolen.

  If user for any reason repeats k (due to buggy software or accidental transaction doubling) or another user takes the same k, then only one coin with given k can be withdrawn (the first one tried to be withdrawn), and the others are effectively lost. We recommend to take the position in the tree as input when computing h, so same commitments in different leafs can still be withdrawn.

- Consider now the randomness value r. If r is stolen or has low entropy, the anonymity is broken as explained earlier in the text: an attacker can compute $H_1'(r)$ and for every submitted nullifier he computes the original commitment $H_1(k, r)$. In the low entropy case the attacker can verify his guess against the history of commitments.

  If r repeats, nothing bad happens, as it is used only for a proof and no function of it is published during the withdrawal. As long as k and r together have sufficient entropy, an attacker can not link two commitments together.

We checked the Tornado random number generator (RNG) in the supplied client code and **found that it yields enough entropy** for security. All third-party implementations must ensure a good RNG is used for secret generation.

5.3 Claim verification

In this section we check the original security claims by the designers.

1. Consider a successful withdrawal with nullifier hash h. A SNARK proof is sound and complete, so Prover knows k, r, l, O such that $h = H_1(k)$ and $O(l)$ opens $H_1(k, r)$ to root R. Assuming that MiMC is collision resistant, $H_1(k, r)$ must be a leaf in the tree, so for each withdrawal there must be a deposit in the tree it refers to. **True**.

2. Suppose some deposit C has been withdrawn twice. According to the SNARK proof, there must be a valid (k, r) pair in each case. As H_1 is collision resistant, the pairs are the same, but then nullifier hashes $H_1(k)$ must also collide, which is forbidden by the protocol. **True**.

3. If all k are different, a coin can always be withdrawn if its nullifier is unique. For the nullifier to repeat, a collision must be found in the nullifier hash function. This contradicts the assumption of MiMC being collision resistant. **True**.

4. If k or r are unknown, a coin can not be withdrawn as this would contradict the SNARK proof of knowledge of k and r. We have not found a way to front run the transaction when k is unknown. **True**.

5. A SNARK proof can not be used for another h as the circuit involves the calculation of h. The recipient address and fee can not be changed either, as long as they are in the SNARK proof context (this must be checked in the implementation too). **True**.

6. We have found that all primitives have at least 126-bit security, except for the BN254 curve. It does not make sense though to increase the security above the BN254 level, since finding discrete logarithms on it would imply signature forgery in the entire Ethereum, which would be a bigger problem than the Mixer insecurity. **True**.

7. It is obvious that if the contract has zero balance, then every earlier deposit has been withdrawn. We have, however, found one more such case.

   Suppose a tree with root R has n leafs, and there has been n withdrawals with proofs for R or an earlier root. Then all n deposits to R must have been t. **Mostly true**.

5.4 Optimality

We have found several ways to optimize the protocol:

- The value r can be optimized away, since there are two secret variables for each deposit which are not opened. As long as k has sufficient entropy and different functions are used to compute the commitment and the nullifier hash, one variable is sufficient. An alternative construction would be

$$\text{Commitment } C = H(k, 0);$$
$$\text{Nullifier hash } h = H(k, 1, l),$$

  where H is non-homomorphic hash function and l is the tree position. With a 15-byte k, everything can be packed into two field elements. This construction is also robust to nullifier reuse: in this case, deposits will be identical but both can still be withdrawn.

- The same hash function can be used for tree hashing, nullifier hashing, and commitment construction.

- The used hash functions are expensive, particularly Pedersen hash, which has a large codebase and largest number of SNARK constraints. If the gas costs become too high, an alternative might be to use a more recent (but not that thoroughly tested) Poseidon [GKK+19] with its 2:1 variant and the S-box x^5 adapted for the BN254 curve. It has 80 S-boxes per call (and 240 constraints) where it processes two field elements. Thus for the SNARK proof we need 18 calls to Poseidon, thus 4320 constraints in total, a 5x reduction over the current case.

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).

[BD17] Razvan Barbulescu and Sylvain Duquesne. "Updating key size estimations for pairings". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 334 (cit. on p. 4).

[Bon19] Xavier Bonnetain. *Collisions on Feistel-MiMC and univariate GMiMC*. Cryptology ePrint Archive, Report 2019/951. https://eprint.iacr.org/2019/951. 2019 (cit. on p. 4).

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, et al. "Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems". In: *IACR Cryptology ePrint Archive* 2019 (2019) (cit. on p. 6).

[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).

[Peda] *Circomlib: Pedersen Hash*. https://github.com/iden3/circomlib/blob/master/circuits/pedersen.circom. 2019 (cit. on p. 3).

[Pedb] *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 pp. 1, 3).