Drying Money Laundering

Preserving privacy with Tornado Cash while disincentivizing malicious activity

13/03/2023 @high_byte

This post assumes you know what are blockchain, mixers, tornado cash and zero-knowledge proofs, but not necessarily all about the technical details.

Tornado Cash - the controversial cryptocurrency mixer (well, mostly controversial outside crypto circles) is, regardless, a marvel of technology.

However, subjectively, it is not without flaws. If I have to define it I might say it aligns with chaotic gravitating towards neutral good. But it is for the beholder to decide.

In this post I suggest an improvement to Tornado Cash such that it disincentivizes malicious actors while still preserving privacy of honest users.

But first, a foreword:

1. *I am not a US citizen or resident.*
2. *I have not deployed, operated, or intend to operate Tornado Cash, fork or similar service, present or future.*
3. *Here's the demo repo*

If you're familiar with a section feel free to skip to the next.

1. Intro to mixing
2. Intro to ZK
3. Tornado Cash internals
4. Disincentivizing malicious activity

Intro to mixing

Let’s say Alice is using a mixer. Typically it would look something like this:

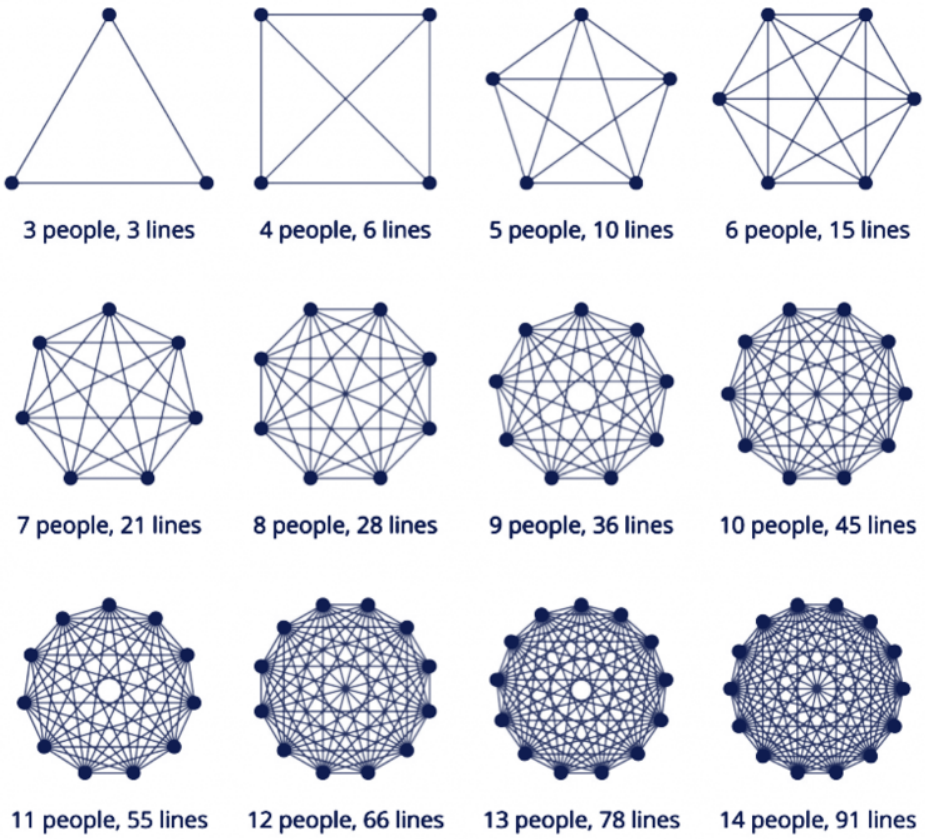
1. Alice deposits money from address 0xb0bee to the mixer
2. Magic
3. Alice withdraws money to address 0xc0ffee from the mixer

Normally there would be a connection between 0xb0bee and 0xc0ffee, even if they are not known to be associated with Alice it can be seen on the blockchain that these addresses interacted with each other.

However on a mixer this link is broken using zero-knowledge proofs. It is possible to tell that 0xb0bee deposited money to the mixer and that 0xc0ffee withdrew money out from the mixer, but there is no direct association between them.

The link between the value you put in and the value you take out is broken by leveraging zero-knowledge proofs.

The more people use the mixer, the larger the set of possible associations, until it is impractical to have real-world certainty about any association.



Of course if 0xb0bee deposits 15.2 ETH and 0xc0ffee withdraws exactly 15.2 ETH we can safely assume these addresses belong to the same entity, so it is common to wait some time before withdrawing.

Think about it like a laundry machine where people put in clothes, you put in a white sock, the machine mixes the clothes, and then you take out a red sock out, where those socks are fungible in their effectiveness to warm your feet. (read: they are equivalent)

Intro to ZK

For those who aren't familiar with ZK - think about this analogy:
You have a coin (or one sock if it's Christmas?) and a combination lock which is unlocked.
You put both of them inside the mixer's box. (super anonymously, under the darkness of the night, while wearing a ski mask on top of a fake mustache)
Let's imagine Ted (ie. Tornado Cash) coming every night to gather coins and locks.
Ted puts one coin behind each lock. But Ted doesn't know which coin belongs to which lock, since they were mixed.

In fact, now nobody can tell whose coin is behind which lock. But the lock owner knows the combination of the lock and they still have one coin behind it. And equally important, any owner can withdraw only as many coins as they deposited.

Then when you want to withdraw (after some time as I recommended earlier) you simply use your combination to unlock one coin. (super anonymously, under the darkness of the night, while wearing a ski mask on top of a fake mustache and hope everybody else do the same)

In this analogy Ted is a trusted entity and can steal all the coins. So for sake of completeness let's replace Ted with Jimmy, an open source machine whose function everyone can verify and is powered by an AI model with no desire to steal.

Tornado Cash internals

Now that we have the building blocks for a mixer, let's take a look at a real-world example.

Users of Tornado Cash generate a proof on a ZK circuit written in Circom using some math magic.
This proof is then verified in Solidity with some dark sorcery which was buried in the literature for decades until it was unleashed onto the world by the wizards of cryptography.

pictured: the concept of zero knowledge proofs
(he's actually a really nice guy when you meet him in person)

In the previous analogy I used locks and coins, while in practice the lock is a commitment and the coin… well it's a coin.

To secure the mixer, Tornado Cash ensures that:

1. Exactly one coin per deposit & withdrawal

   Tornado Cash deployed several versions of the contracts which allow deposit & withdrawal of exactly 0.1 ETH, 1 ETH, 10 ETH and 100 ETH (to which I refer to as “a coin” regardless of units) in order to prevent deanonymization. (as mentioned earlier)

2. One withdrawal per deposit (prevent double-spending)

   To prevent double-spending the contract also ensures the commitment can only be used once with a simple mapping (commitment => boolean).

```
28    // we store all commitments just to prevent accidental deposits with the same commitment
29    mapping(bytes32 => bool) public commitments;
```

   The commitment is basically a hash of some random secret bits. You can send this hash onto the chain, but only you would know what the underlying secret is.

   When you deposit a coin then this commitment is inserted into a Merkle tree, and once you wish to withdraw the contract verifies the ZK proof that your commitment is in the Merkle tree.

```
46        component tree = MerkleTreeChecker(levels);
47        tree.leaf <== hasher.commitment;
48        tree.root <== root;
49        for (var i = 0; i < levels; i++) {
50            tree.pathElements[i] <== pathElements[i];
51            tree.pathIndices[i] <== pathIndices[i];
52        }
```

   And that the (hash of the) secret of that commitment is only used once (hence the commitment is only used once), namely the nullifier (hash):

```
174   function withdraw(
175     bytes calldata _proof,
176     bytes32 _root,
177     bytes32 _nullifierHash,
178     address payable _recipient,
179     address payable _relayer,
180     uint256 _fee,
181     uint256 _refund
182   ) external payable nonReentrant {
183     require(_fee <= denomination, "Fee exceeds transfer value");
184     require(!nullifierHashes[_nullifierHash], "The note has been already spent");
185     require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use a recent one
```

3. Only depositor can withdraw (ie. no frontrunning)

   (To be precise: only the entity that generated the commitment proof can withdraw)

> Finally this is where the magic really happens.
> Rather than revealing the secret on chain (and thus revealing who is the depositor), instead a ZK proof is supplied to verify that you know a secret which would hash to a commitment inside the Merkle tree, without actually revealing which commitment it is!
>
> To reiterate the analogy from before: you have a lock (commitment) and you prove that you know the combination without actually giving away the combination.

An interesting optimization of Tornado Cash is leveraging the fact that the tree is of a sequential list, ie. the leaves all represent strictly increasing indices.

Because of this, instead of building the entire tree from scratch on each deposit (2^N nodes where N is depth of the tree - unrealistically expensive) then only the parent branches of each deposit are updated (N nodes - relatively very cheap)

Understanding this will become important in the following section.

Disincentivizing malicious activity

The idea is simple: prevent withdrawals from addresses that are known to be associated with hacks or malicious activity.

How to create a fair system for this is for another post, but let's assume some entity called a Revoker decides the fate of a deposit, which may be an admin, an oracle or a DAO, and it is fair enough not to deter honest users.

Now since we need some time to verify exploits and blacklist them, there must be some delay to deposits before they can be withdrawn. But this is okay since the best practice is to wait some time anyway.

To implement this in practice may look trivial at first, but not so trivial at second glance:
The first intuition might be to blacklist commitment sourced in malicious activity from the Merkle tree. But withdrawals are not associated with their commitments! Hence they cannot be blocked this way.
So the solution is to remove the commitment from the Merkle tree. However, to do so requires us to rebuild the Merkle tree and as mentioned before - it is incredibly costly!

So instead I ask the Revorker to rebuild the new tree off-chain, supply a proof for the new tree and verify on-chain that only this one commitment was modified in the new tree.

I know this sounds complex, because it is a bit complex, but it is the way to integrate secure revoke capabilities into Tornado Cash.

[See my implementation of the revoke() function here.]

Final words

At the time of writing these words, the developer of Tornado Cash, Alexey Pertsev, remains in Dutch jail for several months now, in what I and many others believe is without acceptable justification.

At the time I was uninformed and had to forge my own opinion on the matter and so I went to research Tornado Cash. My conclusion is that besides developing and publishing Tornado Cash onto the chain, he did not operate, monetized or had any administrative access to Tornado Cash, nor is it likely a vulnerability exists which he or others could exploit, and hence without clear charges or a hint of evidence there is no justification keeping him another day in jail.

Stay strong.