HarryR, Barry Whitehat

Thanks to Wei Jie Koh for review and feedback.

Intro

Privacy on Ethereum has evolved a lot in the past months. With mixers likeTornado Cash in production and others in progress. Its important to note that issues like user advocacy, trust, and education remain a problem that needs to be prioritized.

Here we propose a system that merges the anonimity set of multiple tokens and denominations to create a larger anonimity set.

The system is architected such that it can be deployed on Ethereum today.

Leaf format

Inside the snark a merkle tree holds the mapping of funds to owners. Where users only know the value of their leaves and every other leaf is hidden. The format of the leaf is a hash.

leaf = hash(hash (public_key, relayer_gas, id_salt), token_type, balance, leaf_salt)

hash (public_key, gas, id_salt)

is the identity commitment, which is secret to the user.

public_key

is the leaf owner's EdDSA public key.

relayer_gas

is the amount of ETH that can be paid out as rewards to relayers from the burn registry which execute transactions on behalf of users. This is like a fuel tank for the leaf; a leaf can undergo transfers, swaps, etc and this pot of ETH pays relayers, and can also be topped up during these operations.

id_salt

is a random value which only the depositor should know. It should be different from the leaf_salt

.

token_type

: depending on the implementation, this can be the token contract address, or just the ID for a mapping of token data in the mixer contract.

balance

is the amount of tokens held by the leaf.

leaf_salt

is a random value which only the depositor should know. It should be different from the id_salt

.

Deposits

A user deposits by sending an Ethereum transaction to a contract function which transfers tokens to the contract and adds a leaf to the Merkle tree. The contract confirms that leaf corresponds to the balalnce that was transfered to the contract.

Withdraws

A user withdraws by calling a contract function with a snark proof. The proof is described below. If the proof is valid, the contract destroys their leaf in the Merkle tree and publishes it to the contract which allows them to exit. The function also takes a snark proof from the user.

Internal transctions (transfers, atomic swaps, join-splits)

In order to create an internal transaction the user signs a message:

hash(from_leaf, new_leaf[0], new_leaf[1], to_token, amount, fee)

for both input leaves.

Our transfer snark consumes two leaves and produces two more leaves.

Each transaction is defined by a EdDSA signature, this allows users to hold funds on hardware wallets and also seperates the control of funds from the snark prover. Allowing for atomic swaps.

Inside the snark we perform

1. sig verification first input
2. merkle proof that leaf 1 was in the tree
3. sig verificaion second input (for atomic swaps, the owner of leaf 1 needs to get the signature from the counterparty)
4. merkle proof that leaf_2 was in the tree
5. Destroy leaf_1 and make leaf[0].token, leaf[0].amount and leaf[0].fee available for new leaf creation
6. Destroy leaf_2 and makes leaf[1].token, leaf[1].amount and leaf[1].fee available for new leaf creation
7. The split and join mechanism now allows for new leaf creation adding or splitting the total balance as long as the token_type is the same.
8. The fee can be paid from the fee balance of either or both leaves.
9. The new leaves are checked against the signatures in the message to ensure that they are correct.
10. The EVM then adds new_leaf_1 and new_leaf_2 into the Merkle tree.

This can be used to create atomic swaps, transfers, splits and joins.

Atomic swap

In order to perform an atomic swap the maker creates a signature such that

1. Anyone can create a new leaf by consuming theirs but only IF they create a new leaf that is defined in the signature. ie new_leaf == sig.new_leaf[0]
2. This means that user_1 can create a signature that lets a user transfer their leaf if and only if they create a new leaf defined by user 1.

Privacy implications

1. The maker loses the privacy of their leaf the taker is able to see when that leaf was created.
2. The taker is not able to see when the taker spends the leaf their recived because they don't know the salt in the identity nullifier used to create the nullifier.
3. The maker knows the new leaf created by the taker because its the amount that they traded. But they cannot see when the takers leaf was created

Advertising orders

User cannot publish thier orders as an order book would reveal a lot of information.

Instead users who have an order add their IP address to a list stored in a smart contract and also pay a small fee.

They are then contracted by other people on that list. They perform a MPC protocol together. This MPC only returns a positive result if the two orders are matched and then the maker shares their leaf and signature with the taker who creates the proof and broadcasts it.

Example atomic swap

1. I deposit 200 DAI into the smart contract. I approve transfer of 200 DAI by the smart contract and then call the contract with 200 DAI passing my identity commitment

. The smart contract then hashes my identity commitment

with the token_type

and amount

from my deposit. The leaf_salt

is zero and this can be updated in the first internal transaction. But is not required.

1. You deposit 1 eth into the contract the same as above.
2. I create an order and advertise it on the smart contract.
3. You see my order contact me and we perform a multiparty computaion that will only return true if our orders match.
4. Our orders match and we agree to the trade.
5. I am the maker so I send you my signture where the message is

hash(from_leaf, to_new_id_commitment, to_token, amount, fee)

as well as secret information of my current leaf.

1. You use this to create a snark that destoys my leaf and yours if an only if you also create a new leaf for me and for you where our token balances and types are swapped.

Dummy leaves

Users need to be able to add dumy leaves if for example they only want to use a single input and output. We want all transfers to look the same. So we allow them to set hash(nullfiers, "empty_leaf")

and add that as a new leaf. We use a similar trick to always produce 2 nullifiers.

This should be secure as long as we use a collision resistant hash function.

Paying for transactions

Each leaf in the tree contains a second gas balance. This is denominated in eth and is only used to pay broadcaster for transactions. Each transaction also subtracts fee

wei from this gas and this is sent to the miner. Using something similar to burnRelay.

Conclusions

Here we deine a follow up to the current mixer architecture which requires no changes to Ethereum in order to deploy. It support atomic swaps which will allow us to build a single anonymity set from Ethereum’s many tokens.

Future work is required here to

1. Produce a proof of concept
2. Analyize the privacy properties, specifically the implications of allowing atomic swaps.
3. Implement the mpc for order matching mechanism and possibly investigate other methods to prevent abuse.