Rabbit Mixer

u/bigdaddyrabbit00

March 20, 2018

Rabbit Mixer is a trustless RingCT based Ether mixer, deployed as a Ethereum smart contract and an accompanying DApp. Rabbit Mixer allows Ethereum users to store and transfer Ether (technically a token that's exchangeable 1:1 with Ether) without revealing the amounts involved.

Rabbit Mixer's cool features are:

- Use stealth addresses to transfer Ether to yourself or others without connecting the addresses on the blockchain.
- Transfer and withdraw Ether just by signing messages, removing the need to first acquire Ether to pay for gas.
- Further obscure transfers by mixing it with other people's transactions.
- Since the amounts are hidden (even from the mixer), pay only a reasonable, flat fee to use the mixer.

1 Introduction

The Rabbit Mixer is an Ether Smart Contract and DApp that allows users to change their Ether into a Stealth Ether Token (at a 1:1 ratio, always). Stealth Ethers are stored and transferred using Pederson Commitments of the form $bG + vH$ where b is the blinding factor and v is the value of the token. G and H are points on the BN-128 curve.

1.1 Stealth Addresses

RingCT is used to transfer tokens between stealth Ether addresses. A stealth Ether address is just a regular address, but it never receives or sends Ether, and never makes transactions on the blockchain. Stealth addresses only store Stealth Ether in the contract. To transact, stealth addresses sign messages containing the amount, destination address etc... and send the signed messages to the mixer, which then executes them on behalf of the stealth address. You can even withdraw stealth Ether into a regular, unlinked address on-chain by simply signing a withdraw transaction.

This means that a stealth address never needs to acquire Ether to pay for gas to transact with the smart contract, which allows for true unlinkability on the blockchain. That is, deposit of stealth Ether from a source address and its subsequent withdrawal to a destination address are not linked on the blockchain. (See privacy section, this may still be possible if you don't use the mixer properly)

1.2 Rabbit Mixer DApp Account

An account on theRabbit Mixer DApp is an ordinary private key and its corresponding Ether address. We recommend you use a fresh Ether address for security reasons. The Account consists of the primary address (corresponding to the private key) and a series of stealth addresses derived from the private key.

Each Stealth Address can receive, transfer and withdraw Stealth Ether. To work with RingCT, each Stealth Address needs to keep track of its blinding factor b and the balance amount v. Receiving and Transferring Stealth Ether modifies both the blinding factor and the balance, and the DApp keeps track of these.

However, to make the DApp easy to use, the entire account, including all the stealth addresses, their balances and blinding factors are entirely recoverable from the primary private key alone. Also note that the account is not dependent on the DApp. The private key and a full Ether node can recover the entire account. See the (Blinding Factors and Shared Secrets section) for how this works.

2 RingCT and Gas Costs

The Rabbit Mixer uses the RingCT scheme as described by [G Maxwell and others]. The primary drawback of RingCT is that large size of the proofs. This hits the Ethereum implementation particularly hard, because verifying a single RingCT range proof can cost over 2 million gas. The Rabbit Mixer uses 3 range proofs per sender, which is prohibitively expensive. 6 million gas per transaction at today's prices can cost over USD 100, much too expensive for the average Ethereum user.

To get around the high gas costs, Rabbit Mixer doesn't actually verify the RingCT range proofs on chain. Instead, we use a Truebit-like scheme to offload the verification off chain by offering a bounty to anyone for showing that a published range proof is wrong.

The way it works is that the mixer stores the range proofs in the contract, along with a bounty. Anyone can then verify the range proofs off chain, and if the mixer has published a fraudulent proof, force a verification of the stored proof on chain, which will show that the proof was fraudulent, and claim the bounty and rollback the transaction.

This scheme cuts the gas costs by one order of magnitude. Storing the proof in the contract costs approximately 200,000 gas instead of 2 million for the full on chain verification per range proof. Additionally, this scheme can be extended to verify other parts of the transaction, like verifying the totals of the pederson commitments of the mix, moving more costs off chain.

With this scheme, the mix transaction costs approximately 700,000 gas per sender, translating to USD 5 - USD 10 per transfer.

3 How Stealth Transfers work

While the contract implements the RingCT scheme as is (don't roll your own crypto!), a description of how the contract uses RingCT is helpful. Also remember that all the Elliptic Curve math is on the BN-128 curve, which is the curve Ether's precompiled *ecAdd* and $ecMul$ contracts support (and not the secp256k1 curve used by the underlying Ethereum).

3.1 Transferring Stealth Tokens

To transfer stealth tokens from one address to another, the sender needs to sign a message and send it to the Rabbit Mixer, which executes the transfer along with others.

3.1.1 Sender

First, the sender calculates the following:

eph = An ephemeral private key, randomly generated
P_{eph} = The public key on the BN-128 curve corresponding to eph
P_{rec} = The public key of the receiver. The mixer maintains a mapping $address => publickey$
f = The fee amount that the mixer is requesting

The sender then calculates 3 blinding factors

b_s = Blinding factor for S, the amount to send to the receiver, derived using $ECDH(eph, P_{rec})$
b_t = Blinding factor for T, the total amount to be deducted from the sender's account, including fees. This is deterministically calculated using $keccak256(privkey||nonce_{token})$
b_f = Blinding factor for F. Calculated $= b_t - b_s$

The sender calculates the pederson commitments and range proofs.

S = $b_sG + sH$, pederson commitment of amount receiver will receive
F = $b_fG + fH$, pederson commitment of the fee for this transaction
T = $b_tG + tH = S + F$, pederson commitment of the total amount that will be deducted from the account
$RProof(T)$ = Range proof for T
$RProof(S)$ = Range proof for S
$RProof(bal_{new})$ = Range proof for bal_{new}. $bal_{new} = bal_{current} - T$
ENC_s = The value of s, the amount hidden in S, encrypted by the ECDH shared secret

Additionally, we'll include the meta data for this transaction

$from$ = Sender's Address
to = receiver's Address
$nonce_{token}$ = Nonce for the token as stored in the contract alongside the stealth balance. This is different from the address's nonce on the blockchain.

The sender also includes a hash of the receiver's details, in case we need to challenge the mixer if it goes rogue

$$H_{receiver} = keccak256(to||S||random)$$

The sender saves $random$ and $H_{receiver}$ in case the sender needs to challenge the mixer's transaction. It hides the receivers details in the message hash, and then signs it.

$H_{msg} = keccak256(from \| nonce_{token} \| T \| H_{receiver})$
$v, r, s = Sign(H_{msg})$

The final message that is sent to the mixer is

$$(P_{eph}, S, T, F, b_f, RProof(T), RProof(S), RProof(bal_{new}), from, nonce_{token}, to, H_{receiver}, v, r, s)$$

3.1.2 Mixer

On recieving a transaction request from a sender, the mixer combines it with other transaction requests from other senders into a single "mix". For each mix:

Step 1: Prepare Senders

For each sender, the mixer publishes 3 transactions.

1. Publish the sender's signature proof. This contract transaction verifies, on chain, that the mixer has a signature from the sender to transfer some stealth Ether. The stealth Ether nonce is also a part of the signature, which ensures that the mixer cannot reuse some old signature to double-spend.

   The mixer publishes $(Address_{sender}, T, nonce_{token}, H_{receiver}, v, r, s)$ to the contract

2. Publish two range proof contract transactions, one each for T and bal_{new}. The contract simply stores these range proofs rather than verifying them.

   The mixer publishes $(T, RProof(T), RProof(bal_{new}))$ to the contract

Step 2: Prepare receivers

The mixer publishes one contract transaction with all the receivers. This contract transaction lists all the receivers and the pederson commitments of amounts that they will receive, along with the range proofs for these amounts.

The mixer publishes $(Address_{receiver}, S, RProof(S))$ of each of the sender to the contract.

Note that the pederson commitment is S, while the sender's contract transaction has publised the pederson commitment T, which means the senders and receivers cannot be linked by comparing the amounts. The number of receivers is one more than the number of senders, since the fee collected from each sender is aggregated and sent to a fee address. From the contract's perspective, the fee address is not distinguishable from the rest of the receivers.

As a part of the receiver transaction, the contract sets the status of the mix to 'prepared', which means it is now open to be challenged. Senders (anyone, actually) can verify

1. The two range proofs for each sender and one range proof of the receiver.
2. That the sum of the amounts deducted from senders equals the sum of amounts added to receivers
3. Each sender can verify that the mixer has sent their S to the correct receiver.

The contract provides convinent $verify$ and *challenge* functions to verify that the integrity of the mix. The contract allows for some time for the world to challenge the mix. A successful challenge results in the transaction being rolled back and a bounty paid out to the challenger. Note that up until now, the balances have not been actually modified.

Step 3: Execute the mix.

The mixer then publishes one last transaction that modifies the balances of the senders and receivers, completing the mix.

As a part of this contract transaction, the mixer also passes in to the function encrypted data that is passed along to the receiver, so that the receiver may be able to spend the received stealth Ether.

The mixer sends (P_{eph}, ENC_{b_s}) to the contract.

3.1.3 receiver

The receiver receives two values logged as Ether Events. The P_{eph} of the sender and the actual stealth Ether sent, encrypted by the ECDH key. The receiver can then decode the pederson commitment S, using the ECDH shared secret, which is the blinding factor, and the actual amount, which is decrypted using the ECDH shared secret.

Note that P_{eph} is never published in the contract and is unlinkable to the sender by the world (except by the mixer itself, which can link the sender and the receiver)

3.2 Depositing Ether

Depositing Ether into an account is performed in two steps.

First, the user sends a contract transaction with the value of the Ether they want to deposit. This turns the deposited ethers into Stealth Ethers, and are added to the user's Ether address that it was sent from. Obviously, this transaction is visible to the world, since the amount of Ether sent is visible to everyone. Also, since we have no way of generating a secret blinding factor, the blinding factor for this transaction is 0.

The second step does a Stealth transfer from the user's primary Ether address into one of his stealth addresses. This is done as a mix transaction, which ensures that the deposit becomes unlinkable to the original depositor.

3.3 Withdrawing Stealth Ether

Withdrawing Stealth Ether from the contract is done as a mix transaction.

For withdrawals, the sender, instead of sending the pederson commitment S to the mixer, sends the amount s directly to the mixer. The mixer then performs all the steps as usual.

When the contract is executing the mix, it notices that the amount is not a pederson commitment, and so instead of incrementing the balance of the receiver, pays out Ether to the receiver. All the other invariants of the mix still hold.

It is worth noting that when Ether is withdrawn, it still cannot be linked to the sender, since the withdrawals are done mixed in with other transactions.

4 Contract and Transaction Safety

The contract has a number of safety mechanisms in place to ensure that neither the mixer nor the users can cheat. They are performed as both on-chain checks and off-chain verification of the integrity of the contract.

4.1 Auditing Contract Balance

Even though the balances and transfer amounts are hidden, we can audit that the contract is storing exactly the amount of Ether equal to the total of all the hidden balances.

For each Stealth transfer:

$$
\begin{aligned}
T &= S + F \\
(b_t G + tH) &= (b_s G + sH) + (b_f G + fH) \\
&\implies \\
b_t G &= b_s G + b_f G \\
tH &= sH + fH \\
&\implies \\
t &= s + f \\
b_t &= b_s + b_f
\end{aligned}
$$

where:
- T = Pederson commitment of Total amount sent
- S = Pederson commitment of amount Sent to the recieving address
- F = Pedersom commitment of the Fee
- b_t, b_s, b_f = Blinding factors for T, S and F
- t, s, f = Amounts hidden in the pederson commitments for T, S and F

In every transfer, both the amount and the blinding factors are conserved. Additionally, note that when converting Ether into Stealth tokens and withdrawing Stealth tokens into Ether, the blinding factors are set to 0.

This implies that the sum of all the blinding factors in the contract will always equal 0. So, if we add up all the Stealth balances, the sum should have a net 0 blinding factor. That is,

$$\sum_{\forall A} B(A) = b_{contract} G + 0H$$

where:
- $B(A)$ = Stealth Balance of address A
- $b_{contract}$ = Total amount in the contract

While this computation is too expensive to perform on chain, anyone can perform this audit on their own nodes and verify that the contract really has all the Ether that the balances are hiding.

4.2 Invalid Range Proofs

If the mixer publishes an invalid range proof, anyone can claim a bounty against it. However, if the mixer publishes a range proof that throws while it is being verified, it cannot be challenged, as the challenge transaction will throw and revert.

While the code doesn't throw anywhere, primitives can throw if passed invalid arguments, especially the ecAdd and ecMul precompiles. The contract tries its best to validate all arguments so that it will never throw (and only return false), but there's no gaurantee right now of this. Solidity is really missing a catch for exceptions thrown from precompiles.

4.3 Stealth Balance Nonce

Each address that has a balance in the contract also has a nonce that indicates the number of transfers sent out. When initiating a stealth transfer, the sender has to sign a message including the current nonce, which is checked on-chain by the contract.

This ensures that the mixer cannot steal from the contract, and that a sender really has authorized a transaction to a given receiver.

4.4 Preventing double spending

Along with the balance and nonce, the contract also locks a sender if they have initiated a transaction until the transaction is complete or rolled back. This prevents a sender from sending two transactions at the same time.

If a sender were allowed to send two transactions at once, they could potentially spend more than their balance, because the contract keeps track of only the current balance and current stealth transfer, and the sender has no way to provide a range proof for $(bal - T_1 - T_2)$. To get around these issues, we restrict to only a single outgoing stealth transfer from an address.

4.5 Receiving before/during/after a stealth transfer

While we prevent multiple outgoing stealth transfers, an address can receive multiple incoming stealth transfers at the same time, even when an outgoing stealth transfer is in progress. This is possible because for all incoming transactions, there's a range proof provided for S, which proves that the amount S is hiding is > 0.