Introduction

The need for privacy in digital transactions is paramount in today's digital age, yet most blockchain technologies fall short in offering complete anonymity. Current methods like mixers are often vulnerable to external threats and regulatory scrutiny. Our proposal addresses these challenges by providing a secure, decentralized solution, ensuring privacy for stakeholders including individual users and organizations requiring confidential transactions.

This document introduces a solution for achieving private transfers using a burn/mint Zero-Knowledge Proof (ZKP) mechanism, addressing the limitations of cryptocurrency mixers like TornadoCash. These mixers, while effective in obscuring Ethereum transactions, face potential government bans due to their traceable interactions. To enhance privacy protection for both senders and receivers, we propose a mechanism that maintains sender anonymity through plausible deniability.

The successful implementation of this proposal will significantly enhance privacy in blockchain transactions. This innovation addresses a critical problem in the digital asset space - the lack of true anonymity. By safeguarding transaction details and user identities, our solution will benefit the community by fostering a more secure and private blockchain environment, encouraging broader adoption and trust in blockchain technology.

[

Screenshot 2024-02-13 at 10.27.48

1552×1206 97.2 KB

](https://ethresear.ch/uploads/default/original/2X/3/334de24b02feed27fd3f91f1aedcf3bb13c68a5e.jpeg)

Example User Flow

1. User has 100 tokens, he has to pay a small amount each time he makes a transfer for network fees.
2. User's account 1 sends 1 token to 99 intermediary addresses (Nullifiers), effectively burning the token and the remainder to his account 2, to fund the account for later network fees.
3. User generates Proof of knowledge on how the Nullifiers were generated (ZKP)
4. User's account 2 submits hashes of Nullifier addresses, ZKP, recipient and ZKP public inputs to an on-chain verifier.
5. Verifier validates ZKP, invalidates Nullifier hashes and mints tokens to the recipient

Requirements for the protocol

To meet the stipulated requirements for ensuring privacy and security within the protocol, the following design principles must be adhered to:

1. Proof of Knowledge Without Key Disclosure:

The protocol must enable users to generate and submit proof of their knowledge of the private key corresponding to an intermediary address, without actually disclosing the private key itself. This can be achieved through the use of Zero-Knowledge Proofs (ZKPs), specifically designed to verify the possession of a secret (the private key and nonce) without revealing the secret itself.

1. Privacy of Key and Nonce:

The protocol must ensure that only the user knows their private key and the nonce used for Nullifier generation. This confidentiality is crucial for maintaining the security of the Nullifier and ensuring that no other party can replicate or misuse it.

1. Intermediary Addresses Concealment:

To prevent the possibility of transaction tracing back to its origin, intermediary addresses must never be revealed on-chain by the protocol. This can be facilitated through the use of hashed representations of these addresses during transactions or in any on-chain data structures.

1. Sender concealment

: In order to ensure privacy, the protocol must not disclose the sender address on-chain. This requires private transfers's phases to be a initiated from 2 separate accounts so it's infeasible to backtrack based on the mint transaction's initiator.

1. Standardized

Constant Transaction Amount:

The amount transferred to intermediary addresses must be constant across all transactions within the protocol. By eliminating variations in transaction amounts, the protocol further reduces the risk of transactions being linked to specific users or recipients based on the value transferred. Additionally, the constant amount ensures that the verifier does not need to know the specific value being transacted, thereby enhancing privacy and making it harder for external parties to correlate transactions with their participants. Transactions to intermediary addresses should involve a standardized transfer amount that is sufficiently generic to avoid detection through analysis of transaction volumes. This standardization helps in obfuscating the flow of funds, making it difficult for third parties to identify specific transactions based on the amount transferred.

1. OFAC compliance:

The rollup that uses the protocol can opt-in to allow the service to be used only by addresses that have not been restricted, ensuring legal compliance.

Spendable and Unspendable Address (Nullifier)

An account’s balance on the Ethereum Virtual Machine (EVM) is considered spendable

if it meets one of the following conditions:

1. There is a known private key s

, where the account address A

is derived as the last 20 bytes of the keccak hash of the public key. The public key itself is calculated by multiplying a secret scalar s

with the generator point G

(s x G) in Elliptic Curve cryptography.

1. The account is associated with a smart contract c

, where A

is derived as the last 20 bytes of the keccak hash of the contract’s parameters.

Accounts that do not fulfill either condition are deemed unspendable

and are referred to as Nullifiers

in this document.

It’s presumed that for a Nullifier generated using a private key s

we create, no other party can access its balance. This is due to the EVM’s design, which supports a vast number of potential addresses (2^256).

Generating a Nullifier

To create a Nullifier, we generate a random address using a known private key s

and demonstrate control over its creation process. Employing a hash function different from EVM’s native keccak

, such as poseidon

, gives us ZK-friendliness and a source of randomness, assuming it’s computationally infeasible to find two inputs that produce the same hash across different hash functions. This method, combined with Zero-Knowledge Proofs (ZKPs), allows us to validate the existence of an Ethereum account holding ETH in the stateRoot

without revealing the value of s

.

A Nullifier is generated by hashing the concatenation of the secret key s

and a nonce using the poseidon

hash function. The resulting nullifierAddress

is confirmed to be a non-existent account on-chain.

Verifying Non-Existence of a Nullifier On-Chain

To verify that an address is unspendable

, we check if it has already been recorded on-chain usingeth_getAccount and checking the nonce, code hash, balance and storage. If the account is empty, it confirms the address's non-existence and thus its status as unspendable

. This verification is essential before sending funds to a Nullifier.

Sending Funds to a Nullifier

Transactions sent to a Nullifier are recorded on-chain, rendering the ETH effectively burnt since the address is unspendable

. Only the creator of the Nullifier

can assert its status. Through ZKP, we can substantiate our knowledge of the Nullifier's creation, facilitating the minting of associated amounts to a new, untraceable address.

Nullifier Address Generation Service

An off-chain service will be developed to facilitate the generation of Nullifier addresses, verification their non-existence on-chain, execution of burn transactions, and initiation of the proof generation and minting processes.

ERC20 vs native token for the protocol

The protocol can operate with either an ERC20 token or a network's native token. Using an ERC20 token simplifies implementation but centralizes control to a single smart contract, potentially subject to regulatory pressures. A native token implementation requires more extensive modifications but enhances privacy and direct minting capabilities.

ZK Circuit Design

The circuit needs to have the following public inputs. The verifier (think smart contract) will get these and provide it as input to the verification circuit.

- blockhash
- It allows the circuit to reason about the validity of the block header and the underlying stateRoot
- nullifierAddressHash[]
- a list of hashes

of Nullfier addresses.

- blacklistHash
- hash a list of blacklisted addresses, maintained by legal parties.
- receiver
- address of the target receiver. Checked against the blacklist

for compliance

The private inputs

of the circuit will include the above constituents in their raw form. In addition the following inputs will be passed:

- nullifierAddress[]
- a list of raw Nullifier

addresses.

- rlpHead
- All the blockhash

constituents up to the stateRoot in their RLP encoded form. It should also include the prefix indicating that the complete concatenated RLP encoded block header size when rlpHead

, stateRoot

and rlpTail

are concatenated.

- stateRoot
- state tree root of the chain in this block. Should be 32 bytes.
- rlpTail
- All the blockhash

constituents after the stateRoot in their RLP encoded form.

- storageRoot
- The storage tree root for this account.
- eip1186Proofs
- EIP1186 objects, containing an array of nodes of the MPT proving the nullifierAddress

exists in this state tree and its root is stateRoot

. storageProof

is not checked against and can be omitted. Balance needs to be positive.

- s
- private key of the wallet that generated the unspendable

nullifier addresses

- nonce_start
- start index of the auto increment keys for the nullifier address generation
- nonce_end
- end index of the auto increment keys for the nullifier address generation
- blacklist[]
- list of blacklisted addresses to be checked against. Supplied by legal parties
- sender
- address of the initial transaction sender.

The circuit has to prove for each nullifierAddress

the following things:

1. nullifierAddress = h(s | nonce)

, where s

and nonce

are supplied as private inputs. h

is the poseidon hash

1. nullifierAddressHash = h(nullifierAddress | nonce)

, where nullifierAddress

is a private input, nonce

is derived from the private input nonce_start

and nullifierAddressHash

is a public input.

1. nullifierAddress

exists in the stateRoot

as of block identified by blockHash

and corresponds to an account with positive balance. Requires a valid block header to be supplied as private input. The 3 private input constituents keccak256

hashed together must equal the blockHash

Additionally, the circuit needs to ensure blacklistHash = h(blacklist[])

as well as validate sender

and receiver

are not part of the blacklist[]

.

On-chain Verification

The On-chain Verification process involves a specialized Solidity smart contract, referred to as EIP7503Verifier

, designed to validate zero-knowledge proofs (ZKPs) and ensure the integrity of transactions related to Nullifiers. Here's an explanation of its operation:

1. Receiving and Validating Proofs

: The EIP7503Verifier

contract initiates its process by receiving a submission that includes a ZKP (proof

), the hash of a block (blockHash

), an array of hashed Nullifier addresses (nullifierAddressHash[]

), a hash of the list of blacklisted addresses by regulators (**blacklistHash**

) and a recipient address (receiver

). The contract first validates the blockHash

against the last 256 block hashes stored by the Ethereum Virtual Machine (EVM). This step confirms that the proof pertains to a recent state of the blockchain, ensuring its relevance and timeliness. The assumption is that this time frame is adequate for generating and submitting proofs. Future enhancements may include integrating a trusted oracle to access a broader range of historical block hashes, further strengthening the verification process.

1. Invalidating Used Nullifiers

: Upon successful proof verification, the contract marks the nullifierAddressHash

as used by recording it in an internal mapping. This critical step prevents the reuse of Nullifiers, addressing potential issues like double spending or double minting, which could undermine the token's integrity. By storing only the hash of the Nullifier address rather than the address itself, the system enhances privacy and security, making it more challenging to trace transactions back to their original Nullifiers.

1. Minting Tokens

: After verification, the EIP7503Verifier

contract mints a predefined amount of ERC20 tokens and transfers them to the specified receiver address. This action completes the process, allowing the Nullifier to be converted into a tangible asset on the blockchain while maintaining the anonymity of the original transactions. The choice to use ERC20 tokens provides flexibility and compatibility with a wide range of wallets and other smart contracts.

Additional Considerations

:

- Accessibility and Security

: The design philosophy behind the EIP7503Verifier

ensures that any account can initiate a mint transaction following successful proof verification. This inclusivity fosters a decentralized and user-friendly environment. The absence of ownership modifiers in the protocol design eliminates centralized control points, which could be targeted for censorship or blacklisting, thereby enhancing the protocol's resistance to external pressures.

- Quality Assurance

: > 95% test coverage. This level of testing is crucial for verifying the contract's functionality across all possible scenarios, ensuring that the contract behaves as expected under various conditions and effectively mitigates risks associated with smart contract vulnerabilities.

Node Enhancements

To enable a rollup to mint new native tokens, it is imperative to upgrade the Node with a new transaction type that receives the ZKP and public input constituents. The upgraded Node will validate the proof and the public input constituents, ensuring they don't collide with the expended (partially or fully) nullifiers. Afterwards it will do accounting necessary to prevent double mint. This enhancement is not only pivotal but also a prerequisite for the successful implementation of Phase 2 of our proposal. By facilitating the minting process of the native token, this modification represents a significant stride in advancing the network's privacy.