Introduction

The Ethereum ecosystem has the most developers and users activities, yet the public by default nature is a barrier to more real world use cases and adoption. A private accounts module where users can manage multiple private burner addresses with one public address could be a solution. Each burner address can be used for a different use case, making user activities unlinkable, while only needing one single private key to manage all these addresses and make it user-friendly. Essentially, this adds a layer of privacy to Ethereum and its corresponding L2s without requiring consensus or a protocol layer change. From the user's perspective, there is no behaviourial change needed as the user still signs transactions with a single account like they currently do, as the burner addresses are mostly abstracted away from end users. The motivation for such a module is to easily enable privacy on EVM chains. Stealth addresses manage multiple addresses with a meta keypair but don't protect sender privacy.

Architecture

There are a few components to achieve this. First, a multi-asset shielded pool where any ERC20, ERC721, ERC1155 tokens can be supported whilst sharing the same anonymity set. The multi-asset shielded pool uses an incremental merkle tree, where its leaves are UTXO-style commitments to the users' transaction outputs. Users have to generate a zero knowledge proof to prove ownership of assets within the shielded pool (ZK proof that it has the corresponding private key to the commitment note), which will then allow them to interact with their assets. Once a commitment note is published, the nullifier is published on-chain. With a multi-asset shielded pool, it opens up the design space as any tokens including long-tailed assets can now be deposited into the shielded pool, and it remains possible to perform swaps inside the shielded pool, making it easier to bootstrap the privacy set. (The UTXO commitment design is adapted from Tornado Nova while the support for multi asset is inspired by Anoma.)

The second component is the generation of burner accounts. Each burner account is a smart contract account that is ERC4337 compatible. Users generate entropy for these burner accounts by using deterministic signatures (RFC6979), where entropy is used by the factory contract to create the smart contract accounts. Since the signature is deterministic, it is possible to calculate the counterfactual address of these burner accounts. The users will then be able to fund these burner addresses from the multi-asset shielded pool using a relayer network. To be fully ERC4337 compatible however, we use a paymaster module to fund the gas fee of the shielded pool withdrawal, while getting a fee from the withdrawn assets. As such, it improves on censorship resistance as the relayer network is essentially the bundler network for the account abstraction system.

[

private-accounts (1)

802×503 29.8 KB

](https://ethresear.ch/uploads/default/original/2X/d/d6b4b35dc8f078783bccd150f062c039f1712a5c.png)

The above diagram shows the flow of generating burner accounts. Essentially, users will route their funds through a multi-asset shielded pool and rely on a paymaster module to fund these burner accounts initially.

The third component is a client side SDK that allows a user to prove ownership (and thus spend assets) of their burner accounts, using a keypair from its public and permanent account. This requires generating a zero knowledge proof to prove that the user knows the secret to the entropy used to create the burner account. Generating the ZK proof also requires having the nonce of the burner account as a public input to prevent replay attack of the authentication flow. The secret is generated through the deterministic signature, therefore users do not need to store the secret anywhere.

Putting these pieces together, the user flow looks like this:

1. The user has an existing EOA account that is public.
2. The user will be able to deposit some funds (any assets) into a multi-asset shielded pool.
3. Each time the user would like to make a transaction privately for whichever use cases, a new burner address is generated and funded through the paymaster, where the user will then be able to make the transaction from the burner address. All these are managed by the client side SDK, so abstracted away for the user.
4. From the user perspective, it is still only signing a transaction using its public account. The user will only need to manage a single keypair, which is the keypair of the public account.

To showcase the use of the private account module, we think it might be best presented in the form of a wallet interface. Users are able to manage multiple burner accounts within a single interface. The reason for using burner accounts to achieve privacy is to ensure easy integration and full composability into existing dapps, as it does not require maintaining a separate addressing system.

Future Work

There could be an optional compliance mechanism embedded. There has always been risk of bad actors utilizing privacy protocols for illicit activities, including the infamous Tornado Cash. We define a module where governance will be able to add particular deposits into a blocklist. Using a sparse merkle tree, users have to generate a zero knowledge proof to prove their non-membership in this blocklist each time they would like to interact with their deposited assets. This is a modular component as applications can decide whether to include this feature. Existing implementations for this include Proof of Innocence by Chainway and Privacy Pools by Ameen.

Ask for Feedback and Collaborations

We believe privacy is a fundamental building block for more real world adoption of the Ethereum blockchain. Our mental model is that for privacy to be truly adopted, it has to be adopted as a by-default layer in a dominant consumer product, which is why we are building this open source module on the accounts level, rather than building another private DeFi protocol which requires users to explicitly opt-in for privacy. We believe the private accounts module could be a middleware to be integrated by wallets, as it does not require much behavioral change from the end users.

The private accounts module is currently open-sourced at GitHub - eerkaijun/private-accounts (adapted from ZrcLib which is a shielded pool open source library that we built during a one-month hacker house, shoutout to Rudi and ZP). We would like to obtain community feedback on the architecture design and on any related aspects, so would love to hear your thoughts and suggestions for improvements. If you are interested in contributing, ping me on Telegram at @kaijuneer

.

We propose the above architecture as the baseline to manage private accounts. A few open ended questions:

- With the use of a single keypair to manage all burner accounts, there is inherently a risk of key loss/compromise. We can’t use a smart contract account as the main account though, as a smart contract can’t make a signature, which is required to prove ownership of the burner accounts. An alternative would be to use a keystore contract as proposed by Vitalik.
- For users to prove ownership of their burner accounts, the smart contract account needs to do an on-chain zero knowledge proof verification. This inherently increases transaction costs significantly, even on L2s. Proof aggregation might be a design space worth exploring further.