TLDR:

We present a new design for cross-chain unified smart contract accounts. In thedesign proposed by Vitalik, a preset keystore contract is needed and is used for wallet's verification information. User links differenet wallet instances on different chains to the keystore. We propose a new structure where the need of a preset keystore is eliminated. User can simply create a wallet instance on a chain and then get another wallet instance on another chain with the same address, as long as he can prove his ownership of the original wallet.

Definition

A smart contract account is considered cross-chain unified if it satisfies the following properties:

1. You can have multiple wallet instances on different chains with the same address.
2. Others cannot create a wallet instance with the same address as yours without stealing your account.
3. You can sync the state of the wallet between different instances on different chains at ease, e.g. the guardians of that wallet.

Solution

The third property of a cross-chain unified account is ensured similarly to the light version mentioned inVitalik's post: each wallet instance has an information stored on the chain at which the instance is located, and syncs through ZK or KZG proofs once an instance changes its state.

In this article we will focus on how to ensure the first and second property is accomplished.

First we have a wallet factory instance with the same address on different chains. This can be done viaEIP-2470 (aka Nick's method). When user wants to simply create a wallet, the wallet factory would take in the InitData, nonce (unique) and chainId to calculate a salt, e.g. salt = keccak256(encode(initData, nonce, chainId))

. The chainId is to prevent replay attack on another chain. This part is equivalent to creation of a normal account abstraction wallet. User can start to use it now and set different validation rules to the wallet (for example, in our wallet you can set different validator).

When the user wishes to use his wallet on another chain but remains his unique address, he then calles the CreateAccountOnTargetChain

function in his wallet. The wallet verifies his identity (with any kind of verification methods), and then passes the CreateAccountGiveSalt function call to bridge, with the salt that originally generated his first wallet. Any bridge can relay this message to the factory on a desired chain, and the factory would create a wallet instance with this salt using CreateAccountGivenSalt

, acquiring the same address. So the first property holds.

To ensure property 2 is accomplished, the wallet could verify the message when calling CreateAccountGivenSalt

and require the call is from the Bridge. In other words, users have to verify their ownership on one chain to create a new instance with the same address on another chain.

If one is concerned about the security of the bridge (like Vitalik said in his post), the bridge can be easily replaced by any low-level proofs such as ZK or KZG proofs, as long as thoe chosen method can relay verified messages.

The workflow is as illustrated in the following diagram:

[

1808×1166 265 KB

](https://ethresear.ch/uploads/default/original/2X/6/65405722097e326e0bd0661c05b8ba6f2d7e5603.jpeg)

Pros

There are several advantages of this implementation in comparison to the keystore solution:

- User experience.

Users do not have to deploy a keystore contract first and then link his wallet instance. Users can simply begin their journey on one chain, and transfer their identities (addresses) to another chain whenever they wish to.

- Backward compatibility.

User of the existing account abstraction wallet can simply upgrade his/her factory and does not have to change its wallet instance. Where the keystore solution requires an upgrade of the wallet instance.

- Gas efficiency.

If user decides to only use his / her account on one specific L2, he / she would never have to prove any cross-chain message. Where the keystore solution always require a keystore contract (which is normally on L1).

Cons

- State sync.

The sync process can only be implemented as the light version in Vitalik's proposal. You have to sync between all the networks, whereas the keystore solution (heavy version) allows you to upgrade in just one place, where the keystore contract resides.

- Harder to maintain privacy.

The wallet's verification information is always on the same chain with its instance, making private guardians harder to be implemented. In contrast, the keystore method can use ZK-based proofs to ensure privacy.

One more word on state syncing

The light version of cross-chain state syncing where you bridge the change of verification method change to each chain is quite burdensome. It actually would be more viable through a cross-chain paymaster and a chain-agnostic signature, i.e. you let your guardian sign a chain-agnostic signature to change your verification information. This signature could be replayed on different chains, by anyone. When you submit the transaction on one chain, using a cross-chain paymaster, the paymaster would submit the signature on all desired chains and deduct gas fee from the one chain you have asset on. This may look naive at first, but currently provide much better UX, gas efficiency and security than cross-chain syncing.