Challenges and possible solutions for moving from an enshrined encryption and note tagging scheme to an abstracted one implemented in the account contract via a standard interface.

Context

Aztec today has full native account abstraction, as in every account is implemented as a contract and there are no EOAs in the system. This allows each user to define their tx authorization mechanism as part of their account contract, thus not enshrining any signing scheme in the protocol.

However, being a private network, Aztec requires the ability to encrypt

data: when sending an encrypted note to a user, the sender needs to be able to provably encrypt this note with the recipient's public key and send it. Today, encryption keys and the encryption scheme are enshrined in the protocol, and while we have ideas on how to allow rotating these keys, the encryption cryptographic scheme is still set in stone.

Motivation

While abstracting away the encryption scheme may not sound too interesting, abstracting the note tagging is certainly promising. Today, a user discovers their notes by trial-decrypting every single note emitted on chain, which is horribly inefficient. We're exploring privacy-preserving schemes where a user can query a node for their notes without revealing which they are

, which requires tagging

each note with a string for discovery. However, this is an active area of cryptographic research, so it'd be useful to be able to iterate fast on it without having to execute a protocol upgrade for every improvement we want to roll out.

Moving the logic of both encryption and note tagging toupgradeable account contracts would allow wallet developers to rapidly adopt new cryptographic techniques around note tagging, and roll them out as part of their account contract implementations.

High-level design

The high-level design is relatively simple. Whenever an app needs to send an encrypted note to a recipient, they call into an encrypt_and_tag(plaintext)

method in the recipient's account, which takes care of encrypting and tagging the data, and potentially broadcasting it as well. Different account contract implementatiosn can use different standards, as long as they are supported by the accompanying wallet software.

Challenges

Security: King of the hill

A malcious user could define an encrypt

method that fails in certain circumstances, bricking application flows that depend on them receiving a private note. While this issue already exists in Ethereum when transferring ETH (see the king of the hill), it can be worked around by "catching" a revert in the recipient. However, in Aztec, private functions are compiled to "native" circuits (unlike public functions which run in a VM circuit), so any execution failure in a private function makes the entire transaction unprovable. This means it is not possible to catch errors in calls to private functions.

Usability: Initialization

Having to call into an account contract in order to interact with it (eg by sending private assets to it) means that the contract must have been deployed beforehand. This means that it's not possible to send funds privately to an address before that account has interacted with the chain by deploying their account contract, which is a usability pain.

Performance: Increased function calls

With the current design, the cost of creating a proof of private execution is proportional to the number of function calls, since every call requires another recursive proof of the kernel circuit. Having to do an extra call for every emitted note will increase proving time substantially.

Mitigations

Security: Precompiles

A mitigation to the king-of-the-hill issue is forcing account contracts to only use a set of pre-vetted and enshrined functions for encryption, similar to “precompiles” in the Ethereum world. These functions could be guaranteed not to fail when called, and by being embedded in the protocol, also allows to ship protocol-wide bugfixes or preformance improvements via protocol upgrades.

The downside of this solution is precisely that it depends on protocol upgrades for shipping new improvements, which was one of the main reasons for abstracting encryption in the first place. Nevertheless, by making it an opt-in upgrade for wallets, it reduces the burden of coordination when rolling out new designs.

Usability: Embedding default contract classes and keys in the address preimage

We can embed in the address preimage an identifier for its default contract class. This means that the initial implementation of a contract is known just from the complete address which can be shared by the recipient. The same applies to the initial encryption public key: in that it can be embedded in the address preimage.

So, as long as the encrypt

method depends only on data in the address preimage and does not access storage (ie does not require the contract to have been deployed and initialized), it can still be executed just by having access to the complete address.

Alternative designs

Abstracting at the application contract

An alternative design is to move the choice of encryption and tagging scheme from the account to the application contract. This eliminates the three challenges above, but opens the door to more fundamental issues around adoption and fragmentation. New encryption and tagging schemes require adoption by wallets, so apps shipping new schemes would first need to get buy-in from wallet vendors to add support for them. This eventually leads to apps that can only be used by certain wallets, potentially fragmenting the ecosystem. Furthermore, defining the encryption scheme that a user wants to use for their data is a concern of the user (and hence the wallet they choose), not the app they are interacting with.