With the goal of stress-testing the current Account Abstraction design, we would want to implement some new use cases. We recently built a TouchID account contract as part of an offsite, and it'd be nice to tackle a social recovery scenario next. For this, we can take a page out of Argent's Guardians feature, as described by Julien:

[…] the original Argent implementation of guardians on L1 (e.g. for social recovery). On L1 guardians are accounts, i.e. they can be EOA or smart contracts when the guardian is another Argent user. And these guardians approve actions by providing a signature (so that they don't have to pay for their guardian duty). However, that model of guardians is not working with 4337 […] since you cannot access the storage of another contract during validation making it impossible to verify the signature of a guardian contract (i.e. you cannot call is_valid_signature

on the guardian).

In other words, we'd need to implement something likeEIP1271 on our account contracts, so an account can "vouch" for something without having to actively submit a tx. This way:

- Guardians sign a message that "the signing key for account 0x123 should be changed to pub2"
- A tx is submitted to account 0x123 with the signatures
- Account 0x123 validates that each signature is valid asking the respective guardian account contract
- The account updates its pubkey

However, there's a complication: if the signing key for an account is private, how can the sender of the tx create a proof that includes it, without having access to the decryption keys of the guardian? We have a few options, some of them good, some of them not so much:

1. Guardians share their decryption keys with tx sender

Given decryption keys are scoped by contract, they'd just be revealing their signing public key to the tx sender (typically the user behind 0x123). This is probably the easiest way out, but sharing private keys is not a really nice approach. It's also interactive, and not well-suited for other EIP712-like scenarios.

2. Store signing public keys unencrypted

Another easy way out: if the keys are not encrypted, you don't need decryption keys to read them (I believe a meme withthis template should go here). In general, less private state makes composition easier. And we could argue that the signing pubkey is not something that needs to be kept private. But we can do better.

3. Each guardian sends an individual tx

Same to how the original Gnosis MultisigWallet worked: instead of each owner signing an offchain message, they actually submit a tx that stores their intent on-chain, which can be executed once the threshold is reached. This works with the current design: each guardian sends a tx from their account to 0x123's, creating a private note encrypted for 0x123 that states that they validate a pubkey change.

Note that the "encrypted for 0x123" could be a problem if the user behind that account has lost their keys. Maybe we need to support encrypting with an arbitrary pubkey in that scenario?

4. Recursive ZKPs

EIP1271 uses the same mechanism as the protocol but at the app level (signature verification) to check that a claim by a user. Why not do the same? Guardians could simply share a ZKP that they have signed a message, and then the single submitted tx would recursively verify all these ZKPs before making any changes.

However, this means we need a primitive for verifying a ZKP from a Noir contract

. This is probably not the same as verifying a ZKP in vanilla Noir, since this ZKP will need access to state (eg the guardian's current signing key).

It's not clear to me whether the ZKP submitted by the guardian would be an instance of an App Circuit, or needs to be wrapped in a Kernel Circuit, or could be an arbitrary one. My gut feeling is that, if we need to prove stuff of the state of the chain, we probably need something very similar to a Kernel Circuit proof, but

it needs to somehow specify that it is an "offchain" proof and not an actual tx that can be submitted.

Thoughts?