TLDR: We can allow encryption keys to be rotated by just publishing new keys to an append-only unencrypted array in the private data tree, and trusting the sender to cooperate and pick the latest one they can see.

Encryption keys are currently enshrined in the protocol and embedded into each address, meaning theycannot be rotated. This means that, while a user can freely rotate their signing keys (assuming their account contract allows it), they cannot rotate the encryption keys. So if the encryption keys are compromised, users have to fully migrate to a new account in order to regain privacy of their operations.

Rotating encryption public keys seems to be a similar problem toupgrading contract implementations: both encryption public keys and implementation addresses are pieces of data that need to be read by multiple users within private executions, and both change sporadically.

However, there's a major difference: we don't need to enforce that the value read for an encryption public key is the latest one

. This seems counter-intuitive, since it allows a malicious sender to use a discarded (and possibly compromised) encryption public key. But this doesn't open the door to any new attacks: a malicious sender can also just publicly disclose the information they've encrypted for the recipient, and anyone can verify the disclosure is correct by re-encrypting the data and checking it matches the emitted encrypted note. So, given the privacy of a note already depends on the cooperation of the note sender, we can assume that they will cooperate and use the latest encryption public key available for a recipient.

This means we can now store encryption public keys in an unencrypted append-only structure backed by the private data tree. A sender needs to prove they've used an encryption public key present in the private data tree, which we can easily do, but they don't need to prove it's the latest one.

Alternatively, we can use the slow updates tree for this as well, which will enforce that the sender uses the latest encryption key, though it comes with the drawbacks described here.

Get public key in Aztec-nr

Implementing this would require changing the get-public-key

function from Aztec-nr. Today, the oracle call returns the fixed public key for a given address, and then the circuit verifies that it can be re-hashed (along with the partial-address) back into the input address.

Now, the oracle would have to return either the same value, or a value read from the private data tree, which would require a read request to a standardised location in the tree, such as hash(address, pub-key-index, pub-key-domain-separator)

.

Public private notes

To support this, we need to add a concept of "public" private notes. In other words, notes that are emitted unencrypted and commited to the private data tree.

While nothing in the protocol prevents it, we need to add APIs for doing this from Aztec-nr. We also need to tweak the note processing service so it recognises these unencrypted notes and tracks them as if trial-decryption had been successful.

Nullifiers

While rotating the encryption public key is good, rotating the nullifier secret can lead to double-spends, since a user would be able to generate fresh nullifiers for notes already consumed. This means that, in order to allow for the nullifier key to be rotated, then we need to include a reference to a specific nullifier key within a note, and have the circuit verify that the nullifier is emitted using that specific key. This binds a specific nullifier key to each note, which also makes it easier for the note processing service to identify when a note was spent.

However, this means that if a nullifier private key is compromised, then the user loses privacy over the spending of all notes already created, even if they rotate the nullifier private key. This seems an acceptable compromise.

Another issue in this approach is that now there is an impact if a malicious sender chooses an outdated compromised public key. While the malicious sender today can leak the information they've encrypted, they could not identify when that note was spent. However, if they encrypt the note with a reference to a nullifier key that's been compromised, then the spending of the note is leaked as well. This means there is a consequence if the sender is not enforced to use the latest key.

Both of the issues above could be mitigated by adding a flow to rotate nullifiers in notes: a user should be able to send a tx that just consumes a note and recreates it under the new nullifier secret. However, this requires each individual contract to support it, via an external function similar to the current compute_note_hash_and_nullifier

. Alternatively, the second issue can be solved by just using the slow updates tree, which enforces usage of the latest keys.

Note that this discussion is orthogonal to whether we abstract away encryption or not. Even if we implement an efficient way to catch “reverts” in private executions and allow account contracts to expose an encrypt_note_for_me

method, the problem remains of where

the public key used in that function is stored.