This post outlines the requirements, functional and not, that we want to meet for addresses, encryption keys, and sending notes in Aztec, along with a proposal that fits the bill and a few other candidates we considered in discussions with @Mike. All feedback is welcome!

A fundamental action in Aztec is sending an encrypted note to another user. This involves looking up their encryption public keys, their preferred method of note tagging (and possibly encryption as well), executing and proving the execution of such methods, and broadcasting the resulting encyrpted note through the chain. See Challenges in abstracting encryption for a previous post on this topic.

Requirements

- Users must be able to choose their note tagging mechanism

. We know the solution we have today is not ideal, so we expect better note discovery schemes to be designed over time. The protocol should be flexible enough to accommodate them and for users to opt in to using them as they become available.

- Users must be able to receive notes before interacting with the network

. A user should be able to receive a note, eg representing a token transfer, just by generating an address. It should not be necessary for them to deploy their account contract in order to receive a note.

- Applications must be able to safely send notes to any address

. Sending a note to an account could potentially transfer control of the call to that account, allowing the account to control whether they want to accept the note or not, and potentially bricking an application, since there is no catching exceptions in private function execution. See the motivation for the pull-over-push pattern in Ethereum for more info, or theking-of-the-hill Ethernaut challenge.

- Addresses must be as small as possible

. Addresses will be stored and broadcasted constantly in applications. Larger addresses means more data usage, which is the main driver for cost. So we want addresses to fit in at most 256 bits, or ideally a single field element.

- Total number of function calls should be minimized

. Every function call requires an additional iteration of the private kernel circuit, which adds several seconds of proving time.

- Encryption keys should be rotatable

. As a nice-to-have, should a user have their encryption private key leaked, they should be able to rotate them so that any further interactions with apps can be private again, without having to toss out their account and create a new one.

Addresses and encryption abstraction

To meet the requirement of having small addresses, we define an address as the hash of a set of public keys, a class identifier (ie a pointer to the code for the contract), the constructor arguments (to guarantee deterministic deployment addresses to be used for counterfactual deployments), the deployer address, a salt, and an identifier for an encryption and tagging method.

This last parameter allows an address to define how notes should be encrypted and tagged for it. The protocol will keep a shortlist of valid methods, and support additional ones through upgrades. The rationale for not letting the account freely implement its own encryption and tagging methods is to avoid security issues when an app sends notes to an account (remember king-of-the-hill!).

Note that our initial requirement was to support note-tagging abstraction and not encryption abstraction. However, since we already need to support one, including the other was pretty much free.

As an alternative, we entertained the idea of using the public key as the address. However, couldn't come up with a scheme where the address could double as a public key and a commitment to all the data needed to guarantee deterministic deployments (ie the class identifier, constructor arguments, and deployer address), all while fitting in a single field element (well, we did come up with a scheme, but @jean demolished it).

Canonical registry

We propose a global singleton contract, deployed at a well-known address, where accounts register their public keys and their preference for encryption and tagging methods. This data is kept in public storage for anyone to check when they need to send a note to an account.

An account can directly call the registry via a public function to set or update their public keys and encryption method. New accounts should register themselves on deployment. Alternatively, anyone can create an entry for a new account (but not update) if they show the public key and encryption method can be hashed to the address. This allows third party services to register addresses to improve usability.

An app contract can provably read the registry during private execution via a merkle memebership proof against the latest public state root. Rationale for not making a call to the registry to read is to reduce the number of function calls. When reading public state from private-land, apps must set a max-block-number for the current transaction to ensure the public state root is not more than N blocks old. This means that, if a user rotates their public key, for at most N blocks afterwards they may still receive notes encrypted using their old public key, which we consider to be acceptable.

An app contract can also prove that an address is not registered in the registry via a non-inclusion proof, since the public state tree is implemented as an indexed merkle tree. To prevent an app from proving that an address is not registered when in fact it was registered less than N blocks ago, we implement this check as a public function. This means that the transaction may leak that an undisclosed application attempted to interact with a non-registered address but failed.

If an account is not registered in the registry, a user could choose to supply the public key along with the preimage of an address on-the-fly, if this preimage was shared with them off-chain. This allows a user to send notes to a recipient before the recipient has deployed their account contract.

Pseudocode

The registry contract exposes functions for setting public keys and encryption methods, plus a public function for proving non-membership. Reads are meant to be done directly via storage proofs and not via calls to save on proving times.

contract Registry

```
// Note that we may need to store additional information aside from encryption_key and method_preference
public mapping(address => {encryption_key, method_preference}) registry

// The valid_methods list should be expanded via protocol upgrades that update this contract
public enum valid_methods

public fn set(encryption_key, method_preference)
    assert method_preference in valid_methods
    registry[msg_sender] = {encryption_key, method_preference}

public fn set_from_preimage(address, encryption_key, method_preference, class_id, constructor_hash, salt, deployer)
    assert address not in registry
    assert hash(encryption_key, method_preference, class_id, constructor_hash, salt, deployer) == address
    registry[address] = {encryption_key, method_preference}

public fn assert_non_membership(address)
    assert address not in registry
```

Apps that want to provably encrypt and tag a note for an address should check the registry first, and then optionally fall back to the preimage if supplied via an oracle call. This would be implemented as a helper function in the aztec-nr library.

fn send_note(recipient, note)

```
let block_number = context.latest_block_number
let public_state_root = context.roots[block_number].public_state
let storage_slot = calculate_slot(registry_address, registry_base_slot, recipient)

let encryption_key, method_preference
if storage_slot in public_state_root
    context.update_tx_max_valid_block_number(block_number + N)
    encryption_key, method_preference = indexed_merkle_read(public_state_root, storage_slot)
else if recipient in pxe_oracle
    address_preimage = pxe_oracle[recipient]
    assert hash(address_preimage) == recipient
    encryption_key, method_preference = address_preimage
else
    call_public_function(registry_address, "assert_non_membership", [recipient])
    return

execute method_preference with encryption_key, recipient, note
```

Note the three code paths when sending a note to a recipient:

- If the recipient is in the public registry, the application uses the public key and encryption method declared there.
- If not, the user assembling the tx can choose to supply the address preimage and load the public key and encryption method from it.

- If not, the application can prove

that it doesn't know how to create a note for a given address, and skip it.

The last code path is what prevents attacks that could brick an application trying to encrypt private state to an address for which its public key is unknown. A simple example would be the king-of-the-hill contract in which the king defines an arbitrary payout address.

Execution

Let's drill down on the last pseudocode line. We propose having a contract for each valid encryption and tagging method. Each contract would have a standard interface with an encrypt_and_tag

function that receives a recipient address, their public key, any additional data associated with the address in the registry, and the note. This function would be responsible for encrypting the note, tagging it, and broadcasting it.

These singleton contracts would potentially be stateful. This allows the implementation of, for example, note tagging methods that require creating additional state, without having to create state in the account contract itself. This solves any issues associated with creating state in an undeployed contract in the event of having to send a note to an undeployed account.

This pattern requires one function call per note, which may not be acceptable given the requirement of minimizing the number of function calls. We then propose a new method for batching calls: instead of a directly issuing a private call, a contract may enqueue

a message to be handled by a target contract. This message, much like an enqueued public call, is not executed synchronously, but enqueued for later execution. Once the private call stack has been emptied, the kernel circuit is then responsible for executing these calls, grouped by target contract.

In pseudocode, the application would do something like:

fn send_note(recipient, note) let encryption_key, method_preference = ... context.enqueue_call_to(method_preference_contract_address, "encrypt_and_tag", [recipient, encryption_key, note])

And a specialized kernel iteration, once the private call stack is empty, would dispatch these calls like:

fn execute_enqueued_private_calls(context): let unique_targets = unique context.enqueued_calls[].target for target of unique_targets let messages = context.enqueued_calls[].filter(enqueued_call.target == target) call_private(target.address, target.function_selector, messages)

By implementing encryption contracts as message handlers, we can batch all note encryption and tagging operations that use the same method within a transaction in a single function call. Given we anticipate few unique encryption and tagging methods, batching should provide considerable improvements.

As a further optimization, we could have handler contracts implement the same function with multiple batch sizes. The message could then be targeted not to a specific function selector, but to a merkle root of all the implementations of the same function, and then the kernel chooses the one that best fits the current batch size.

In pseudocode, the handler contract would look like the following:

contract SimpleEncryptor for BATCH_SIZE in 1..32 fn encrypt_and_tag(requests[BATCH_SIZE]) for request in requests ...

The calling application contract would then enqueue a call to the merkle root of all 32 encrypt_and_tag(requests[BATCH_SIZE])

selectors, and the kernel would pick one, and constrain it to be in the merkle tree.

Alternatives

The original idea was not to have whitelisted contracts that implement the encryption methods, but to introduce the concept of protocol functions

, which are functions blessed by the protocol that a contract could include in its interface. However, this new concept required a bigger change in the protocol to implement. It was also unclear how calls could be batched, or how a contract could update their choice of protocol functions. Also, protocol functions were expected to run in the context of the account contract itself, which required running code and potentially updating state in undeployed contracts (though we may end up supporting this, stay tuned!).

Aside from how the encryption methods were implemented (whitelisted contracts or protocol functions), we considered storing the public key and encryption method preferences in the account contract public storage, removing the need for a

registry. However, this requires standardising a set of storage slots for keeping the public key and encryption method preferences, which could get accidentally overwritten by a contract. It also made it difficult to keep an easy-to-update whitelist of valid encryption methods. It also requires the deployment of an account contract in order to publicly store its public key.

We also considered not storing the public key and encryption method preference at all, and just broadcasting them as an unencrypted event on deployment, along with the entire address preimage. Nodes would need additional logic to track these broadcasted preimages, so that app contracts could load them via an oracle call, and prove the preimage hashes to the recipient address. This is a small performance improvement over keeping keys in storage, since it requires less hashing for the storage membership proof. However, this disallows key rotation and updating encryption method preferences, which is a nice property to have.