Summary of discussions with @Mike on where to implement the steps needed for private message delivery, either in app code or embedded into the protocol, favoring performance and flexibility versus upgradeability and enforcement.

We understand that sending a private message (ie an encrypted note) to a recipient in the Aztec Network requires the following steps:

1. Retrieve the master public keys for the recipient(s) and their tagging scheme preference.
2. Derive an app-specific key from the master key.
3. Derive a shared secret for encryption.
4. Derive a symmetric encryption key from the shared secret.
5. Encrypt the plaintext of the note.
6. Tag the note for easy discovery.
7. Assemble the message to be emitted including the tag and encrypted cyphertext.
8. Emit the message.

Now, each of these steps can be either implemented in aztec-nr as a library, which then becomes part of the compiled application contract, or via a "call" to the protocol (more on the shape of this "call" later).

In-app vs external calls

Running each of these steps as part of the application is the fastest option, since a call has an extra overhead that the user who is creating a proof for the given application will have to bear. However, embedding the code into the application has a major downside: it cannot be modified once the application has been deployed. If the application makes an external call to (for example) encrypt a plaintext, and a better encryption scheme is devised in the future (eg faster to prove in a SNARK, or because a bug was found in the current implementation), the protocol can decide to upgrade the encryption implementation, which will automatically upgrade all apps to use this scheme. If the code was embedded on the app, the app stuck with that version of the code once it's deployed.

Another reason for doing certain operations in protocol circuits as opposed to app circuits is to limit what an application can do. For instance, assembling the message and emitting it from a protocol circuit means that the app is not free to choose the format of the message. This can reduce overhead for wallets, since they can safely assume that all messages will conform to a specific interface. And it can help with privacy, since having apps that implement custom log formats could leak information that those apps were executed in an otherwise fully private transaction. This harms not just the apps that adopt a custom format, but the entire set of users by reducing the size of the privacy set. Of course, the tradeoff here is flexibility: we may want to allow app developers to come up with custom schemes we haven't thought of.

There's also a limitation on where each operation happens based on the interfaces between them. For instance, encryption key derivation is tightly coupled with the encryption scheme chosen. And if we choose to enforce the format of encrypted logs, we cannot just add a protocol call for assembling the message if apps can choose to circumvent it by not calling it and just going directly to emitting.

We are then inclined to the following:

1. Applications retrieve user master keys from a canonical registry using an aztec-nr library, and then silo these addresses (note that siloing may require offloading some verifications to the kernel). Rationale for leaving this as an app concern is that, even if we devised new keys formats or siloing schemes, existing users will keep using the current key format, so there's no point in trying to upgrade apps to a new key scheme if there are users still using old schemes.
2. With the app-siloed key, the app makes a call to the protocol to derive the shared secret, encrypt, and tag. The shared secret, encrypted cyphertext, and tag are then returned to the app. This allows the protocol to migrate to a new encryption scheme in the future, or to introduce new tagging schemes that existing apps can support for the users that opt into them. Otherwise, if we wanted to add a new tagging scheme for users to pick, existing apps wouldn't be able to support them, and we'd end up with a fragmentation in the ecosystem based on supported tagging methods.
3. Given the shared secret, encrypted cyphertext, and tag, the app assembles the log and emits it via an oracle call. This gives flexibility to app developers to coordinate with wallet developers to devise new log formats. We envision several moving pieces here, related to how to handle outgoing data (ie encrypting for the sender), or handling private notes that are meant for multiple recipients. The intention would be to give developers freedom during testnet stage, and once we have data on the different patterns and the tradeoffs on privacy, decide whether to enshrine log formats or not.

What is an external call?

So far we've considered these "external calls" to the protocol to be implemented using "precompiled contracts" or "protocol contracts". These are well-known contracts at distinguished addresses that implement some protocol-wide functionality to be used by apps. Since they are available at a distinguished address, their implementation can be modified via a protocol upgrade that changes the code that the kernel loads for its address.

However, every call to another contract has an extra overhead: it requires an additional proof for the application circuit, as well as an extra kernel iteration. This can add several seconds to the proof generation in the user device. We had envisioned introducing a "batch call" feature to mitigate this, which could help in transactions that emit multiple messages, but this requires implementing a new call method.

Another possible improvement is "merging" each protocol contract with a kernel iteration. Instead of having to execute two proofs for a call to a protocol contract, we can bundle together the protocol contract features with its corresponding wrapping kernel. We believe that, with proper code modularization, we can keep app and kernel responsibilities clearly differentiated, even if they are compiled as a single circuit.

A different option is pushing more features to "reset kernels". So when an app needs to, say, encrypt a cyphertext, it first performs an oracle call to perform the encryption, and pushes into its public inputs (ie outputs to the kernel) a request for the kernel to verify that the encryption result was correct. These requests are passed down from one kernel iteration to another, and once enough of these requests have been accumulated, we run a "reset kernel" that verifies all these requests and clears them from the kernel data. While this seems more performant than the alternatives, pushing more responsibilities into reset kernels means far more data being copied around (we count on DataBus to optimize this!). It also means heavier kernels with multiple responsibilities, which will take more time to prove, or having many more specialized flavors of reset kernels, which means more proving keys that need to be stored in users' wallets.

We believe we need more data on these tradeoffs before we can make a decision here. In the time being, we suggest implementing everything in aztec-nr and bundling into application code, since it's the simplest approach, and then start moving pieces out to either precompiles or reset kernels depending on benchmarks and use cases. Same applies to enforcing log standards: once we start generating usage data, we can understand the impact of flexibility on the size of privacy sets.