Keys

Aztec Labs

Abstract.

Specs for:
- Master key derivation
- App-siloed nullifier key derivation
- Encryption
- Decryption
- Handshaking & Tagging
- Log Layout (not started)

The intention is to use this doc to:
- Discuss uncertainties with the Crypto Research team.
- Explore optimisations (in constraint counts, data broadcast size, privacy).
- Come up with a final spec that we're all happy with.
- Get sign-off on the correctness and security of these schemes from the Crypto Research team.

Table of Contents

1 Introduction

There's already information relating to keys in the protocol spec and in the big miro diagram. But the info in the spec is slightly outdated now, as is the info in the diagram. Also, it's hard to have discourse around keys with the crypto research team, to resolve the hand-waviness of the so-called "spec".
So, this doc will be a living document, where we can discuss uncertainties and problems, and slowly hone in on a good keys design.

Much of this is inspired by the ZCash Sapling and Orchard specs.

1.1 What's actually "in protocol"?

This section probably only makes sense after reading everything else in this doc.

What *is* part of the protocol? It's basically everything except the app-siloed viewing key derivation schemes.

- An Aztec Address, and its derivation method.
- The master keys $Npk_m, Ivpk_m, Ovpk_m$ and their derivation method.
- The Key Registry which stores those master keys.

- Features that the PXE should have:
  - The ability to compute (using the PXE's master secret keys):
    - Hardened app-siloed secret keys.
    - Schnorr proofs of knowledge of a secret key, for a given public key.
    - Chaum-Pedersen proofs of knowledge and equality of the discrete logs of two different points. This is used for proving correct derivation of app-siloed nullifier secret keys; app-siloed outgoing viewing secret keys (if the app chooses to do this); and decryption keys (in some use cases).
  - The ability to decrypt AES ciphertexts using the PXE's master incoming viewing secret key.
- Features imposed on the PXE:
  - an understanding of:
    - log layouts
    - ciphertext headers containing contract addresses
    - ciphertext headers containing symmetric keys

    - ciphertext headers containing pointers to other ciphertexts
    - when to forward ciphertexts & plaintexts to unconstrained app functions, so those functions may decrypt ciphertexts and/or process and store important private state.
- The Tagging Precompile contract and its methods for:
  - Deriving its own app-siloed keys;
  - Deriving a handshake shared secret;
  - Deriving tags;
  - Storing the shared secret and tag indices.
  - But note that the Tagging Precompile's choices don't impact other apps' scope to choose differently.
- The Tagging Precompile also imposes features that the PXE must support in typescript-land/C++-land:
  - Filtering for handshakes that originated from the Tagging Precompile contract, brute-forcing those handshakes to find pertinent handshaking secrets, and neatly storing those secrets.
  - Deriving tags.
  - Making batch queries: passing a collection of tags to a server, and receiving a blob of ciphertexts in return.

2 Notation

The AltBN-254 group is $E(\mathbb{F}_q)$ with scalars in $\mathbb{F}_r$. We won't be using this group at all, so ignore it immediately and forget you ever read this paragraph.

$\mathbb{G}$ is the Grumpkin group $E(\mathbb{F}_r)$ with scalars in $\mathbb{F}_q$. Note that both fields are 254 bits, but $q > r$. The "Field" type in Noir is $\mathbb{F}_r$. Operations in $\mathbb{F}_r$ are cheapest within a snark.

Grumpkin points are denoted with a starting capital letter. Scalars are denoted in lower-case. Elliptic curve group operations are written in additive notation, so operators $+$ and $\cdot$ are used to denote addition and scalar multiplication, respectively.

A subscript $_m$ often denotes "master" key. A subscript $_{app}$ often denotes "app-siloed" key.

poseidon2 : $\mathbb{F}_r^t \to \mathbb{F}_r$ is the Poseidon2 hash function (and t can take values as per the Poseidon2 spec).

Mike: Note that $q > r$. Below, we'll often define secret keys as an element of $\mathbb{F}_r$, because it's efficient to derive secrets as the output of a snark-friendly hash, which outputs an element of $\mathbb{F}_r$. We'll then use such secret keys in scalar multiplications with Grumpkin points. Strictly speaking, such scalars in Grumpkin scalar multiplication should be in $\mathbb{F}_q$. A potential consequence of using elements

of $\mathbb{F}_r$ as secret keys could be that the resulting public keys are not uniformly-distributed in the Grumpkin group, so we should check this. From an old chat with Patrick: The distribution of such public keys will have a statistical distance of $\frac{2(q-r)}{q}$ from uniform. It turns out that $\frac{1}{2^{126}} < \frac{2(q-r)}{q} < \frac{1}{2^{125}}$, so the statistical distance from uniform is broadly negligible, especially considering that the AltBN254 curve has fewer than 125-bits of security.

Mike: Below, illustrative domain separator strings are given as inputs to hashes. It's implied that these strings would be encoded in a single $\mathbb{F}_r$ element, so as to be compatible as an input to the hash function. If the current illustrative string is more bits than would fit, the string would need to be changed.

Mike: Throughout, poseidon2 is assumed to be a pseudo-random function.

The notation $value[0:256]$ is lazy shorthand for a slice - i.e. "decompose this value into bits and take the 0th-255th bits, inclusive".

3 Keys

Let $G \in \mathbb{G}$ be a generator for this key derivation scheme.

Mike: The details for how this G should be derived are fuzzy to me. Zac describes a method for deriving generators for our Poseidon hash here. Perhaps this approach can be adopted here, with some domain separator such as "az_keys"?

3.1 Seed

The seed is derived out of protocol. Different hardware software wallets might choose to derive these values in different ways. A seed is a secret from which all of a user's other keys may be derived. The seed can live on an offline device, such as a hardware wallet.

seed $\xleftarrow{\$} \mathbb{Z}_{2^{256}}$

3.2 Master keys

Todo: correct serialisation of the data into the hash functions. E.g. conversions from fields to bits, etc.

nsk_m := sha512-hmac("az_nsk_m", sk) mod $r \in \mathbb{F}_r$, "Master nullifier secret key". App-siloed nullifier secret keys are derived from this. See later for how.

$ivsk_m$:= sha512-hmac("az_ivsk_m", sk) mod $r \in \mathbb{F}_r$, "Master incoming viewing secret key". Used by a recipient to decrypt ciphertexts which are sent to them (hence "incoming").

$ovsk_m$:= sha512-hmac("az_ovsk_m", sk) mod $r \in \mathbb{F}_r$, "Master outgoing viewing secret key". Used by a sender to decrypt ciphertexts which they sent to someone (hence "outgoing"). Depending on some considerations below, this key might also be used when encrypting.

$Npk_m := nsk_m \cdot G \in \mathbb{G}$, "Master nullifier public key". If we do key rotation (more on this below), a sender of a note will need to include the recipient's Npk_m inside the note, to convey who may nullify the note in future.

$Ivpk_m := ivsk_m \cdot G \in \mathbb{G}$, "Master incoming viewing public key". Used by a sender to compute symmetric key for encryption.

$Ovpk_m := ovsk_m \cdot G \in \mathbb{G}$, "Master outgoing viewing public key". Depending on some considerations below, this key might be used when encrypting data to one's self.

Note: derivation of the master secret keys doesn't need to happen in a circuit, so sha512-hmac is acceptable.

Info: Note: the hardware wallet which stores the seed might not support sha512-hmac. Be careful using alternative hash functions to derive the randomness of these master keys. E.g. if the hardware wallet only supports sha256, then it would not be acceptable to compute these master keys as sha256(...) mod r, since the resulting output (of reducing a 256-bit number modulo r) would be biased towards smaller values in $\mathbb{F}_r$. More uniformity might be achieved by instead computing sk as (sha256(seed, 1)||sha256(seed, 2)) mod r, for example, as a modulo reduction of a 512-bit number is closer to being uniformly distributed in $\mathbb{F}_r$.

3.3 Aztec Addresses

An Aztec address is currently an element of $\mathbb{F}_r$, derived as a hash of lots of information.

Palla & Khashayar are rocking the boat here with some alternative suggestions; seeking other features. :P.

3.4 App-siloed keys

Here are some requirements:

- A smart contract (i.e. a circuit) must not be given (or be able to derive from other secrets that they're given) a user's master secret keys.
- 3rd-party circuits (which prove things about stuff that happened on-chain) must not be given (or be able to derive from other secrets that they're given) a user's master secret keys.

– Users might wish to share certain secret keys with 3rd parties that they trust. But they would *really* need to trust the 3rd party.

3.4.1 Do we even want / need app-siloed viewing keys?

Note: we definitely need app-siloed *nullifier* keys.

Info: Please see here for a very detailed discussion around siloing of viewing keys.
Tl;dr, there's a blue blox at the bottom of that section that suggests that we should recommend to most app developers that they use master viewing keys in their apps. They would still need app-siloed nullifier keys.
Advanced users can still app-siloe their viewing keys - particularly if the inconveniences of hardened viewing keys are acceptable for their use case.
Please do read that linked hackmd section, though (maybe after reading the rest of this doc) - it's quite thorough.

An app still has the option of implementing its own key siloing scheme. We explore some schemes below. But it's likely that we'll recommend that basic apps use master viewing keys.

3.4.2 App-siloed incoming viewing keys

3.4.2.1 Non-hardened derivation

Non-hardened key derivation (named after the similar derivation scheme of non-hardened BIP32 keys) at first appeared to be quite useful scheme for incoming viewing keys. This scheme has the nice property that a *sender* could derive the app-siloed public key (aka "child" public key) of a recipient, knowing just the recipient's master public key, and without any further interaction with the recipient.

However, if someone learns the child secret key, they can use it to derive the master secret key. In our setting, that isn't very helpful. The main motivation for siloing keys was to be able to safely give app-siloed keys to 3rd parties or app circuits without leaking activity across other apps. If a malicious app or an auditor is fed a child secret key, they could use it to derive the master secret key and hence all activity. There are ways to avoid divulging a child secret key (be it to a circuit or to an auditor), through the schorr protocol or the chaum-pedersen protocol (or even through snarks if you wanted to use a hammer to crack a nut), but if using such protocols to prove knowledge of secret keys, then why not just use the master keys for that purpose?

We've concluded that non-hardened keys aren't of use in Aztec. For poster-

ity, we explain non-hardened key derivation (without rigour or review) in the appendix.

3.4.2.2 Hardened derivation

Hardened keys are safer than non-hardened keys, because the master secret key can't be derived from a child secret key. But they're also less convenient for usage as incoming viewing keys. Blockchains often seek a nice property that a sender can send a recipient funds, given only an address (and maybe some master public keys) even if the recipient has never interacted with the chain before. (This property is commonly referred to as "counterfactual addresses"). That's why non-hardened bip32, or Identity-based encryption (IBE) were such nice ideas: because they enabled this property. Hardened BIP32 requires the *recipient* to derive the app-siloed public key and actively share it with potential senders. They might share it off-chain, or they might share it by creating a transaction which broadcasts or stores the app-siloed public key.

But the recipient might end up in a catch-22, where they need to transact in order to broadcast their app-siloed public key so that they can be paid, but they can't transact because they don't have funds yet, but they can't receive funds to pay for a transaction because they haven't broadcast their app-siloed public key, but they can't transact because they don't have funds yet...

The recipient (and not any sender) can derive their own hardened app-siloed incoming viewing public key via:

$$ivsk_{app} := \text{poseidon2}(\text{"}az\_ivpk\_app\_hardened\text{"}, \text{app\_address}, ivsk_m) \in \mathbb{F}_r$$

$$Ivpk_{app} = ivsk_{app} \cdot G \in \mathbb{G}$$

Info: Note that this derivation deviates significantly from BIP32 hardened keys, by making huge simplifications. It's almost unrecognisable, but the name "hardened" has stuck with us, because it conveys that only the owner of the secret keys can derive the hardened child public key.

Question: Is this scheme secure?

3.4.2.3 Non-hardened, hardened, IBE, or master?

It depends, and apps can choose!
IBE is probably too slow to prove or expensive to broadcast. It also results in ciphertexts which are larger than AES ciphertexts, which would make IBE logs distinguishable (to observers) from non-IBE logs. In the interest of maintaining good privacy sets, we're not inclined to recommend this approach.

The above-described catch-22 of using hardened incoming viewing keys might make them unusable for many apps. But apps that assume a user already has funds on Aztec could make use of these keys, by requiring a would-be recipient to register their keys with the app up-front. And if an app enables off-chain interaction between a recipient and sender, then the would-be recipient can interact with a sender and provide the app-siloed public key off-chain.

Non-hardened key derivation is cool, but has these downsides:

- If an app wants to prove attempted decryption, then either:
  - The app would need to be given $ivsk_{app}^{\text{non-hardened}}$, which MUST NOT happen (because that would leak $ivskm$) - the PXE MUST prevent injection of $ivsk_{app}^{\text{non-hardened}}$ into circuits; or
  - The user would need to prove knowledge of $ivsk_{app}^{\text{non-hardened}}$, and prove correct derivation of the symmetric key of the ciphertext. See later for how this can be done. This process would be quite slow though, so probably not practical for trial-decrypting the entire blockchain's history.
- If you want to share $ivsk_{app}^{\text{non-hardened}}$ with a trusted 3rd party, you're effectively also sharing your *master* secret key with them.
  - Another approach would be to prove correct derivation of a symmetric key, for every (master public key, ephemeral public key, ciphertext) tuple.
- An approach to reducing the amount of data to brute-force (when proving attempted decryption) would be to share handshaking shared secrets with that party, enabling the 3rd party to derive the user's tags, and then prove derivation of every symmetric key of every ciphertext that corresponds to those tags, and provide the resulting plaintexts to the 3rd party. This is convoluted, but it does protect the master secret key $ivsk_m$.

Apps could also just have users use their *master* incoming viewing keypair for encrypting and decrypting, instead of app siloing. This would give a user less control over what a trusted 3rd party can see, because the 3rd party would be able to see everything with a master viewing secret key. Note, however, that the trick of proving correct derivation of a symmetric key (as mentioned directly above) is still possible with master keys. See later for how this works.

We'll likely recommend that app developers use a user's *master* viewing keys within their apps. Schnorr and Chaum-Pedersen proofs can be used to prove knowledge of master viewing secret keys, and to prove correct derivation of symmetric keys. If an app can afford the inconvenience of having users interact or register viewing keys, then hardened keys are a good approach.

3.4.3 App-siloed nullifier keys

The requirements for deriving siloed nullifier keys are different from those of siloed incoming viewing keys. Whilst we were seeking a way for a *sender* to be

able to derive a recipient's siloed incoming viewing public key, we don't have this complication when it comes to derivation of siloed nullifier keys. That's because:

- a siloed nullifier *public* key is not useful to us; a sender can just emplace the recipient's *master* Npk_m into notes that he creates for a recipient; and
- only the *recipient* needs to derive the app-siloed secret key (of course) in order to derive the nullifier for their notes.

Info: There are two competing approaches at the moment.

Approach A requires some key derivation logic in a kernel circuit, which isn't ideal.

Approach B enables the key derivation logic to be validated within an app-circuit, which is cleaner, but it could require more constraints because it uses more elliptic curve operations. It's very difficult to determine which approach is most efficient.

Please see here for further explanation of Approach B (although the maths notation is different from here, so maybe ignore the actual maths in that hackmd and focus on the other stuff), including arguments in favour of Approach B, and a table which attempts to compare the gate counts of the approaches. We'll also describe approach B below, without the discussion on gate counts.

Info: **We're currently leaning towards Approach B, as it removes logic and concepts from the protocol.**

Info: Here, app_address $\in \mathbb{F}_r$, but there are more-recent discussions that might change the definition and type of an app_address in search of other nice properties. We'll ignore those newer discussions, for now.

3.4.3.1 Approach A

$nsk_{app} :=$ poseidon2("az_nsk_app", app_address, nsk_m)

The downside to this approach is that we needed to introduce logic inside a "reset kernel circuit" which performed "key validation", because an app circuit is not allowed to be given nsk_m, whilst a kernel circuit can be trusted with a master secret key. Introducing this opinionated key derivation logic at the protocol level is unpopular, and might lead to non-obvious constraint inefficiencies (see here).

The logic inside the key validation reset circuit is:

- Input from app:
  - $Npk_m, nsk_{app}, \text{app\_address}$.
- Input injected by the PXE as a witness:
  - nsk_m.
- Assert:
  - $Npk_m == nsk_m \cdot G$
  - $nsk_{app} == \text{poseidon2}(\text{“az\_nsk\_app”}, \text{app\_address}, nsk_m)$

3.4.3.2 Approach B

This is the currently-preferred approach, if the maths is good. Is thie Chaum-Pedersen?

In addition to the G generator defined above in the "Master Keys" section, also define generators $G_{\text{silo}}, G_{\text{app\_address}} \in \mathbb{G}$, which are generated with some neat domain separators.

Let $Nsk_{app} \in \mathbb{G}$ be:

$Nsk_{app} := nsk_m \cdot G_{\text{silo}} + \text{app\_address} \cdot G_{\text{app\_address}} \in \mathbb{G}$

Notice, it's a point, which is kind of weird and inconsistent with the master secret keys' type, but let's continue.

Deriving the app-siloed secret key in this algebraic way means that knowledge of nsk_m can be proven to the circuit, without passing the secret key *into* the circuit. Recall: nsk_m must never enter an app circuit.

Prover $\mathcal{P}$ wants to prove they know nsk_m and that they used it to compute $nsk_m \cdot G_{\text{silo}}$, without revealing nsk_m:

Prover $\mathcal{P}$ does the following, off-chain (outside of a circuit, in typescript-land or c++-land):

- Let $A, B \in \mathbb{G}$ be defined as:
  - $A := Npk_m := nsk_m \cdot G$
  - $B := nsk_m \cdot G_{\text{silo}}$
- Choose random $\alpha \xleftarrow{\$} \mathbb{F}_r$.
- Let $A', B' \in \mathbb{G}$ be defined as:
  - $A' := \alpha \cdot G$
  - $B' := \alpha \cdot G_{\text{silo}}$
- Compute challenge $c \leftarrow \text{poseidon2}(A, B, A', B') \in \mathbb{F}_r$.
- Compute $t \leftarrow \alpha - c \cdot nsk_m \mod r \in \mathbb{F}_r$.

$\mathcal{P}$ passes the following as inputs to the circuit:

– A, B, A', B', t

Verifier $\mathcal{V}$ (in this case, the circuit) does the following:

– Looks-up Npk_m for $\mathcal{P}$'s address, and asserts $A == Npk_m$.
– Recomputes challenge $c \leftarrow \text{poseidon2}(A, B, A', B') \in \mathbb{F}_r$.
– Checks:
  • $A' == t \cdot G + c \cdot A$
  • $B' == t \cdot G_{\text{silo}} + c \cdot B$

Notice:

$$\begin{aligned} & A' == t \cdot G + c \cdot A \\ => & \alpha \cdot G == (\alpha - c \cdot nsk_m) \cdot G + c \cdot (nsk_m \cdot G) \end{aligned}$$

This proves that $\mathcal{P}$ knows nsk_m and used it in computing $B := nsk_m \cdot G_{\text{silo}}$.

The circuit can then derive $Nsk_{app} \leftarrow B + \text{app\_address} \cdot G_{\text{app\_address}}$, which is the whole objective of this section.

A nullifier for a note can then be derived as something like:

nullifier = poseidon2("az_nullifier", Nsk_{app})
See later for more rigour around deriving a nullifier properly.

Question: Is this secure? **Question**: Can we reduce the number of scalar multiplications in the "Checks" that $\mathcal{V}$ does, by combining the checks as:

$A' + B' == t \cdot (G + G_{\text{silo}}) + c \cdot (A + B)$

? Or does that give the prover freedom to construct malicious B, by fiddling A', B'?

3.4.4 App-siloed outgoing viewing keys

The requirements for deriving siloed *outgoing* viewing keys are different from those of siloed *incoming* viewing keys. Whilst we were seeking a way for a *someone else* to be able to derive a user's siloed incoming viewing public key, we don't have this complication when it comes to derivation of siloed outgoing viewing public keys. That's because:

– only the *owner* of an app-siloed outgoing viewing keypair ever needs to derive outgoing ciphertexts. There's never a case where *someone else* needs to derive a user's app-siloed outgoing viewing keypair.

Similarly to the preceding section on deriving nullifier keys, there are two possible approaches to deriving app-siloed outgoing viewing keys. The approaches are

basically the same as above, just with different letters, and we're leaning towards Approach B. Here's a more brief summary. The method for proving knowledge of $ovsk_m$

3.4.4.1 Approach A

$ovsk_{app} :=$ poseidon2("az_ovsk_app", app_address, $ovsk_m$)

The downside to this approach is that we needed to introduce logic inside a "reset kernel circuit" which performed "key validation". See the above nullifier section for how that would work.

3.4.4.2 Approach B

In addition to the G generator defined above in the "Master Keys" section, also define generators $G_{\text{silo}}, G_{\text{app\_address}} \in \mathbb{G}$, which are generated with some neat domain separators.

Let $Ovsk_{app} \in \mathbb{G}$ be:

$$Ovsk_{app} := ovsk_m \cdot G_{\text{silo}} + \text{app\_address} \cdot G_{\text{app\_address}} \in \mathbb{G}$$

Notice, it's a point, which is kind of weird and inconsistent with the master secret keys' type, but let's continue.

Deriving the app-siloed secret key in this algebraic way means that knowledge of $ovsk_m$ can be proven to the circuit, without passing the secret key *into* the circuit. Recall: $ovsk_m$ must never enter an app circuit.

For the prover $\mathcal{P}$ to prove they know $ovsk_m$ and that they used it to compute $ovsk_m \cdot G_{\text{silo}}$, without revealing $ovsk_m$, see the above section on nullifier keys.

3.5 Key rotation

Info: See a brief list of pros & cons of key rotation here.

Info: Current consensus seems to be to keep "key rotation" as a feature. It does cost constraints though, when reading the current keys, and this extra constraint cost might be enough for people to eventually want to scrap key rotation.

How does key rotation work?

Very briefly:

A user registers their master public keys $Npk_m, Ivpk_m, Ovpk_m$ against their address in a Key Registry contract which is designed, built, and deployed by the protocol devs. The idea would be that all users submit their public keys to this registry, and all apps read from this registry. An app would query the public keys for a given address.

Since the keys are being stored in Aztec's public data tree, reading the keys of a user results in 40+ poseidon hashes. A poseidon hash of 2 fields is approx. 80 constraints. These extra constraints is the main source of controversy.

To rotate their keys, a user would simply make a call to the Key Registry from their account contract, specifying the new set of public keys.

Note that this existence of key rotation impacts how apps must create notes. Previously (pre key rotation), a note on Aztec contained the *address* of a user. But with key rotation that results in double spends. A user could spend a note with a nullifier = hash(note, nsk_1). They could then rotate their nullifier public key, and create an equally-valid alternative nullifier for the same note as nullifier = hash(note, nsk_2). This would be bad. To combat this, the user identifier contained within a note must be their Npk. This way, if a user rotates their keys, the Npk inside the note doesn't change, so there's only ever one valid nullifier.

Patrick: (from the big diagram): If the nullifier key becomes part of the note-hash preimage, then one would have to keep that nullifier key as long as the note has not been spent. That defies key rotation, correct?

Mike: It's certainly not ideal. As you say, it means rotation of keys does not enable you to spend your pre-existing notes if you lost your old keys. Key rotation also doesn't save your pre-existing notes from an attacker who learned your old keys: the attacker won't be able to spend your pre-existing notes (because they would also need to know your account contract's auth key), but they'd be able to see when you spend your pre-existing notes.

Key rotation scenarios. Benefits in green.

Can you / Can the attacker:		Lost your seed, then rotated keys. Can you...	Keys compromised/stolen, then rotated keys. Can the attacker...	Keys compromised, but you Rotated keys regularly. As per left, but...
Spend your existing notes?	Before Rotation	No	No (protected by account contract auth, as long as the app does adequate checks against the user's address. Interestingly, the tagging contract might be susceptible to knowledge-of-nsk-only note edits). So actually, maybe.	
	After Rotation	No	No (protected by account contract auth, as long as the app does adequate checks against the user's address. Interestingly, the tagging contract might be susceptible to knowledge-of-nsk-only note edits). So actually, maybe.	
See when you've spent your pre-existing notes?	Before Rotation	n/a - you can't spend them	Yes	Mitigated to roughly the period the keys were active.
	After Rotation	n/a - you can't spend them	Yes	Mitigated to roughly the period the keys were active.
Rediscover / see your pre-existing notes (and other incoming private data)?	Before Rotation	No	Yes	Mitigated to roughly the period the keys were active.
	After Rotation	No	Yes	Mitigated to roughly the period the keys were active.
Rediscover / see your historic outgoing activity?	Before Rotation	No	Yes	Mitigated to roughly the period the keys were active.
	After Rotation	No	Yes	Mitigated to roughly the period the keys were active.
Discover new notes (and other incoming private data) sent to this address?	Before Rotation	No	Yes	
	After Rotation	Yes (if the sender used the latest Ivpk to create the note)	No	
Spend new notes from this address?	Before Rotation	No	No (protected by account contract auth, as long as the app does adequate checks against the user's address. Interestingly, the tagging contract might be susceptible to knowledge-of-nsk-only note edits). So actually, maybe.	
	After Rotation	Yes (if the sender used the latest Npk to create the note)	No (if the sender used the latest Npk to create the note)	
See when you've spent new notes from this address?	Before Rotation	n/a - you can't spend them	Yes	
	After Rotation	Yes (if the sender used the latest Npk to create the note)	No	
See your future outgoing activity from this address?	Before Rotation	No / n/a - you can't create such logs; you've lost ovsk (although we could use Ovpk for this)	Yes	
	After Rotation	Yes	No	

4 Encrypt

Alice wants to send Bob a private message, e.g. the contents of a note.

The goal of this section is to produce a so-called "log" which comprises:

– Ephemeral public key. (**Mike**: or maybe multiple ephemeral keys - see my questions later).
– Body ciphertext, which contains the actual note.
– Outgoing ciphertext, which enables the sender to derive the symmetric key that was used to encrypt the body ciphertext.
– Incoming header ciphertext. (**Mike**: See below for lots of discussion on log layouts, and what this header might contain, depending on some pending decisions).
– Outgoing header ciphertext.

Mike: Once we've been through the basic maths below, there's a section which discusses the difficult topic of log layouts in more detail.

Alice performs all the computations in this section, since she is the one sending data to Bob (and a record of the data to herself).

4.1 Ephemeral Keypair

$esk \xleftarrow{\$} \mathbb{F}_r$

$Epk := esk \cdot G \in \mathbb{G}$

Question: Now, we've touched on this briefly before: I'd like to use the same (esk, Epk) pair for all the ciphertexts described here:

– body ciphertext;
– outgoing ciphertext
– incoming header ciphertext;
– outgoing header ciphertext;

Maybe I'll elaborate on this in a standalone section.

4.2 Body ciphertext

The body ciphertext is denoted ct_{body}.

Derive a shared secret, using Bob's (possibly app-siloed) incoming viewing public key:

$S_{\text{body}} := esk \cdot Ivpk^B_{app}$

Note also: $S_{\text{body}} = ivsk^B_{app} \cdot Epk$

Info: Notice: as a demonstration of a more-complex flow, I'm using $Ivpk_{app}$

here, but it could just as easily be $Ivpk_m$ (in fact, using master keys will be recommended) or $Ivpk_{app}^{\text{non-hardened}}$ or some other app-siloed key - whatever the app wants to do!

Derive a body symmetric key:

Flag: My AES terminology might need to be corrected, here.

$\text{aes\_randomness}_{\text{body}} := \text{sha256}(\text{"az\_sym\_key\_body"}, S_{\text{body}})$

Todo: Rigorous serialisation of data into functions, e.g. decomposing S_{body} into bits

Flag: Note to self: we can re-use S_{body} to derive randomness for the note! This saves a field element from being included in the plaintext, and hence from being broadcast to L1! The note randomness can be derived as

$\text{note\_randomness} := \text{poseidon2}(\text{"az\_note\_rand"}, S_{\text{body}}) \in \mathbb{F}_r$
Question: In Aztec Connect we did this trick too, but we used sha256 to derive the note randomness. Can we use poseidon2 instead?

Patrick: Comment from the diagram: The ephemeral shared secret is indistinguishable from a uniformly random group element, but not necessarily from a uniformly random 256-bit string. It must first go through an extraction mechanism to get a uniformly random AES key.

Mike: From a quick read of AES-128, it seems we only need 128 bits of secret randomness for a symmetric key, and 96 bits of public randomness for the IV. So 224 bits of randomness. Given that, can we do a poseidon2 hash instead of a sha256 hash, to get 254-bits of randomness for $\text{aes\_randomness}_{\text{body}}$? It would save a huge amount of constraints. Iiuc, we're assuming we can use poseidon2 as a random oracle?

$sym_{\text{body}} := \text{aes\_randomness}_{\text{body}}[0:128]$

$iv_{\text{body}} := \text{aes\_randomness}_{\text{body}}[128:128+96]$

$ct_{\text{body}} := \text{AES128\_enc}(pt_{\text{body}}; iv_{\text{body}}, sym_{\text{body}})$

Question: Do we want to use AES-128, AES-192, or AES-256?

Hooray! We have ct_{body}!

4.3 Outgoing ciphertext

The outgoing ciphertext is effectively an encryption of the symmetric key sym_{body}, for the sender, so that they may re-sync a record of their historic activity in future.

The outgoing plaintext is $pt_{\text{out}} := [esk, Ivpk^B_{app}, \text{address}_B]$.

Mike: This is borrowed from Zcash (like most of this spec). You might wonder why we don't encrypt the $\text{aes\_randomness}_{\text{body}}$ or sym_{body} directly. I wonder the same. I think it's because by including the esk in the plaintext, it serves as a MAC of sorts, enabling Alice to quickly identify that she's successfully decrypted ct_{out} by computing $esk \cdot G$, and comparing the result against the published Epk. If esk weren't given, Alice would perhaps need to do more computation before realising that the message is or isn't for her: she'd need to decrypt the body ciphertext ct_{body}, and then check whether the decrypted note actually exists on chain. One could maybe argue that if everyone uses tags, then Alice will already know whether or not the ciphertext is pertinent to her, so the MAC wouldn't be needed. I'm not sure if everyone will use tags.

We'd be able to save on the amount of data being broadcast to L1 if we could replace $[esk, Ivpk^B_{app}]$ with sym_{body} (my understanding of AES is that iv is appended to the ciphertext and publicly visible? So we put this shorter value sym_{body} inside pt_{out} instead of the longer value $\text{aes\_randomness}_{\text{body}}$). It's a nice potential optimisation to keep in mind. Unless Zcash had other reasons for designing pt_{out} in this way...

$\text{aes\_randomness}_{\text{out}} := \text{poseidon2}(\text{"az\_sym\_key\_out"}, ovsk_{app}, Epk)$

Mike:
See p143 of the ZCash spec for why the Epk is needed in this hash. I don't fully understand the reasoning, but you guys might!

I guess including Epk also ensures this randomness is unique for each ciphertext; otherwise the sender would be computing the same $\text{aes\_randomness}_{\text{out}}$ for every outgoing ciphertext they create!

Do you agree that it should be there?
Also: zcash includes the note commitment and the note commitment's value (the value of zcash contained within the note) in this preimage and I'm not sure why. Any ideas? Maybe it's to further guarantee uniqueness of $\text{aes\_randomness}_{\text{out}}$, since the sender could intentionally repeat their choice of Epk for multiple transactions - but I'm not sure why they'd want to do that? If the sender wanted to maliciously leak the recipient's note information, they could just leak it! I've removed the note commitment and note commitment's value in this derivation.

ZCash use blake2b here. I'm using poseidon2 - is that ok? I ask a similar question in the previous "Body Ciphertext" section.

$sym_{\text{out}} := \text{aes\_randomness}_{\text{out}}[0:128]$

$iv_{\text{out}} := \text{aes\_randomness}_{\text{out}}[128:128+96]$

$ct_{\text{out}} := \text{AES128\_enc}(pt_{\text{out}}; iv_{\text{out}}, sym_{\text{out}})$

Mike: See the "Body Ciphertext" section for questions which also apply identically here.

Hooray! We have ct_{out}!

Khashayar: Comment from the big Miro diagram: General question: do we need to prove correct computation of outgoing ciphertexts anyways? I might emit wrong longs for myself but I can't effect the public state (note hashes and nullifiers) with this. I could just create wrong logs that cause me to not be able to spend a note that is mine. Does this matter?

Mike: I think it will depend on the app. In ZCash, for example, they don't constrain computation of any ciphertexts, since in the setting of money transfer, disputes around not recieving money can arguably happen offchain. Some apps might wish to guarantee these outgoing logs, so that the app developer or some 3rd party can access them, e.g. for some jurisdiction's regulatory reasons.

4.4 Header ciphertexts

Ok, so we've created two ciphertexts:

- ct_{body}, which can be decrypted by Bob (the recipient), using his app-siloed incoming viewing key.
- ct_{out}, which can be decrypted by Alice (the sender), using her app-siloed outgoing viewing key.

But how do Alice and Bob know which app_address to use in order to derive the app-siloed secret keys that can decrypt these ciphertexts? The below proposes that the contract address gets encrypted with Alice and Bob's *master* keys. **Info**: Note: apps can choose how they silo keys - see the section on App-siloed keys for some options and their tradeoffs.

Mike: Note: if we have tags, and if the tags are designed in such a way that they are app-siloed (see the section on tags), then the tags themselves might convey which contract address to use. Some apps might not use tags, though, so I'd rather solve this problem as outlined below.

4.4.1 Incoming header ciphertext

$pt_{\text{header, in}} :=$ app_address

Mike: In a later section on log layouts, the header plaintext becomes more complicated: it will include an id that describes the type of data included in the header, and it will include "offset" and "length" details of where to locate the ct_{body} and ct_{out} in a big opaque blob of lots of ciphertexts. The reason for bundling ciphertexts into an opaque blob is to improve the privacy of what functions were executed in a transaction.

$S_{\text{header, in}} := esk \cdot Ivpk_m^B$

Mike: Notice: as mentioned before, I'm wanting to re-use the same ephemeral keypair here. Pleeeeeeeease. It would save lots of bytes of data form being submitted to L1 (and therefore $$$).
Notice also we're using Bob's *master* Ivpk here.

$\text{aes\_randomness}_{\text{header, in}} := \text{sha256}(\text{"az\_sym\_key\_header\_in"}, S_{\text{header, in}})$**Mike**: It'd be very good if we could use poseidon2 here instead.

$sym_{\text{header, in}} := \text{aes\_randomness}_{\text{header, in}}[0 : 128]$

$iv_{\text{header, in}} := \text{aes\_randomness}_{\text{header, in}}[128 : 128 + 96]$

$ct_{\text{header, in}} := \text{AES128\_enc}(pt_{\text{header, in}}; iv_{\text{header, in}}, sym_{\text{header, in}})$

4.4.2 Outgoing header ciphertext

$pt_{\text{header, out}} :=$ app_address

$S_{\text{header, out}} := esk \cdot Ovpk_m^A$

Mike: Notice: as mentioned before, I'm wanting to re-use the same ephemeral keypair here, to save $$$.

Notice also we're using Alice's *master outgoing* viewing *public* key here. The reason we have to use the public key $Ovpk_m$ instead of the secret key $ovsk_m$, is because a *master* secret key MUST NOT enter an app circuit ever. So the sender encrypts $pt_{\text{header, out}}$ to their own public key, and must use Epk to derive a symmetric key to decrypt.

$\text{aes\_randomness}_{\text{header, out}} := \text{sha256}(\text{"az\_sym\_key\_header\_out"}, S_{\text{header, out}})$

Mike: It'd be very good if we could use poseidon2 here instead for all the aes randomness values throughout. It would save thousands of constraints.

$sym_{\text{header, out}} := \text{aes\_randomness}_{\text{header, out}}[0:128]$

$iv_{\text{header, out}} := \text{aes\_randomness}_{\text{header, out}}[128:128+96]$

$ct_{\text{header, out}} := \text{AES128\_enc}(pt_{\text{header, out}}; iv_{\text{header, out}}, sym_{\text{header, out}})$

4.4.3 The final log

$[Epk, ct_{\text{header, in}}, ct_{\text{header, out}}, ct_{\text{out}}, ct_{\text{body}}]$

Mike: This ignores tags for now. And has the open plea that we can re-use the same *esk* to derive the symmetric keys for three of these ciphertexts (ct_{out} is derived using the secret $ovsk_{app}$ instead of *esk*). The log layout gets much more consideration later.

5 Decrypt

$[Epk, ct_{\text{header, in}}, ct_{\text{header, out}}, ct_{\text{out}}, ct_{\text{body}}]$

5.1 Incoming Viewing Header

Learning the app_address that informs which app-siloed key to derive to then decrypt ct_{body}.

$S_{\text{header, in}} = ivsk_m^B \cdot Epk$

$\text{aes\_randomness}_{\text{header, in}} := \text{sha256}(\text{"az\_sym\_key\_header\_in"}, S_{\text{header, in}})$

$sym_{\text{header, in}} = \text{aes\_randomness}_{\text{header, in}}[0:128]$

$iv_{\text{header, in}} = \text{aes\_randomness}_{\text{header, in}}[128:128+96]$

$pt_{\text{header, in}} = \text{AES128\_dec}(ct_{\text{header, in}}; ivheaderin, symkeyheaderin)$

$\text{app\_address} \leftarrow pt_{\text{header, out}} \in \mathbb{F}_r$

Mike: Hmmm... there's no MAC here, so how does the decryptor know whether they've successfully decrypted this? In Aztec Connect, and maybe in ZCash, the user continued further, until they checked whether the eventually-computed note actually exists on-chain in the note hashes tree or not. That's a lot of extra computation before realising decryption failed. A MAC would be an extra field to broadcast on L1, which would be a big shame.

5.2 Outgoing Viewing Header

Learning the app_address that informs which app-siloed key to derive to then decrypt ct_{out}.

$S_{\text{header, out}} = ovsk_m^A \cdot Epk$

$\text{aes\_randomness}_{\text{header, out}} := \text{sha256}(\text{"az\_sym\_key\_header\_out"}, S_{\text{header, out}})$

$sym_{\text{header, out}} = \text{aes\_randomness}_{\text{header, out}}[0:128]$

$iv_{\text{header, out}} = \text{aes\_randomness}_{\text{header, out}}[128:128+96]$

$pt_{\text{header, out}} = \text{AES128\_dec}(ct_{\text{header, out}}; ivheaderout, symkeyheaderout)$

$\text{app\_address} \leftarrow pt_{\text{header, out}} \in \mathbb{F}_r$

Mike: Hmmm... there's no MAC here, so how does the decryptor know whether they've successfully decrypted this? In Aztec Connect, and maybe in ZCash, the user continued further, until they checked whether the eventually-computed note actually exists on-chain in the note hashes tree or not. That's a lot of extra computation before realising decryption failed. A MAC would be an extra field to broadcast on L1, which would be a big shame.

5.3 Outgoing ciphertext

Learning the symmetric key, by decrypting ct_{out}, so that the sender can then decrypt ct_{body}.

First derive the correct $ovsk_{app}$ from $ovsk_m$ and the app_address that we learned from decrypting $ct_{\text{header, out}}$.

$\text{aes\_randomness}_{\text{out}} := \text{poseidon2}(\text{"az\_sym\_key\_out"}, ovsk_{app}^A, Epk)$

$sym_{\text{out}} := \text{aes\_randomness}_{\text{out}}[0:128]$

$iv_{\text{out}} := \text{aes\_randomness}_{\text{out}}[128:128+96]$

$pt_{\text{out}} := \text{AES128\_dec}(ct_{\text{out}}; iv_{\text{out}}, sym_{\text{out}})$

$[esk, Ivpk_{app}^B, \text{address}_B] \leftarrow pt_{\text{out}}$

We can now do $esk \cdot G$ and compare the result against Epk, as a sort-of MAC. **Mike**: See an earlier question from me (in the 'encrypt' section) asking whether we need this check.

$S_{\text{body}} := esk \cdot Ivpk_{app}^{B}$

See the next section for how to decrypt from here, as the recipient Bob will perform the same computations from here:

5.4 Body ciphertext

Learning the note data, by decrypting ct_{body}.

First derive the correct $ivsk_{app}$ from $ivsk_m$ and the app_address that we learned from decrypting $ct_{\text{header, in}}$.

$S_{\text{body}} := ivsk_{m}^{B} \cdot Epk$

$\text{aes\_randomness}_{\text{body}} := \text{sha256}(\text{"az\_sym\_key\_body"}, S_{\text{body}})$

$sym_{\text{body}} := \text{aes\_randomness}_{\text{body}}[0 : 128]$

$iv_{\text{body}} := \text{aes\_randomness}_{\text{body}}[128 : 128 + 96]$

$pt_{\text{body}} := \text{AES128\_dec}(ct_{\text{body}}; iv_{\text{body}}, sym_{\text{body}})$

$note \leftarrow pt_{\text{body}}$

Hooray! We've decrypted the note! We can now check whether its note hash exists on-chain.

Mike: Note: the note can be hashed by calling a special unconstrained function in the app contract, since we know the app_address.

5.5 Proving decryption without leaking secret keys to a contract or 3rd party

Suppose an app cannot adopt hardened app-siloed viewing keys, but it wants to protect its users from leaking their master secret keys to an app or 3rd party.

Is there a way the user $\mathcal{P}$ can prove correct decryption (to a 3rd party or to a circuit $\mathcal{V}$) without leaking the master secret keys?

The user $\mathcal{P}$ wants to prove, for a given tuple $(Ivpk_m, Epk, ct)$:

- Knowledge of discrete log $ivsk_m$ of $Ivpk_m$, with base G.
- Knowledge of discrete log $ivsk_m$ of S, with base Epk
- That the same $ivsk_m$ was used in both the above.

The verifier $\mathcal{V}$ is then convinced of the correctness of S, and can use it to decrypt ct.

I think we can use the Chaum-Pedersen protocol for this:

- Define: $A := Ivpk_m$, $B := S$.
- $\mathcal{P}$ samples random $\alpha \in \mathbb{F}_r$.
- $\mathcal{P}$ computes:
  - $A' \leftarrow \alpha \cdot G$
  - $B' \leftarrow \alpha \cdot Epk$
- $\mathcal{P}$ computes challenge $c \leftarrow \text{poseidon2}(G, Epk, A, B, A', B') \in \mathbb{F}_r$. **Mike**: If $\mathcal{P}$ is proving this outside of a circuit, then they should use a faster bit-twiddly hash here, e.g. sha256.
- $\mathcal{P}$ computes $t \leftarrow \alpha - c \cdot ivsk_m \mod r \in \mathbb{F}_r$
- $\mathcal{P}$ sends $((Ivpk_m =: A, Epk, ct), S =: B, A', B', t)$ to $\mathcal{V}$.
- $\mathcal{V}$ computes challenge c for themselves: $c \leftarrow \text{poseidon2}(G, Epk, A, B, A', B') \in \mathbb{F}_r$.
- $\mathcal{V}$ checks:
  - $A' == t \cdot G + c \cdot A$
  - $B' == t \cdot Epk + c \cdot B$

$\mathcal{V}$ can then use $S =: B$ to attempt to decrypt ct, to hopefully uncover pt

So when interacting with an auditor, the auditor can ask "which of these (Epk, ct) tuples are decryptable by you (your $Ivpk_m$)?

The user can then provide lots of tuples $(S =: B, A', B', t)$ for every ciphertext, and the auditor can attempt to decrypt ct using S.

This approach might be slow, if there are a lot of ciphertexts to prove attempted decryption. The set of ciphertexts to brute force with this approach can be greatly reduced if the auditor is also privy to the user's tags, or the user's handshake secret for the tags. Then the interaction becomes "Please give me the symmetric keys for all of these ciphertexts, because I know you were tagged".

In the case of interacting with an auditor, this can all be done off-chain and outside of circuits, which will speed things up a lot.
In the case of proving attempted decryption to a smart contract... well, the use case of proving attempted decryption to a smart contract is by its nature very slow, if there are lots of ciphertexts to trial.

6 Tagging

Mike: I've reverted to writing some stuff with pseudocode here, as it's easier for me to think about. So now we have a weird hybrid of maths symbols and code.

Tagging is the act of appending 'some extra data' to a ciphertext, so that the intended recipient can more-quickly identify that the ciphertext is for them.

Note, we're being intentionally vague with the wording 'some extra data', because we want to leave the door open for better ideas to come along in future.

This section explains the tagging scheme we'll initially be adopting in Aztec. The following logic will be implemented in a "precompile contract"; a special contract that is defined at the protocol level.

6.1 High-level flow

- Alice wants to send a note to Bob, via an app.
- The app uses Bob's address to looks-up Bob's other info in the key registry. This info includes his master public keys, as well as the precompile contract address for his preferred tagging scheme. (Note: in the early days of the protocol, there are only two tagging options: don't tag; or use the scheme detailed below).
- The app computes the note, and a ciphertext is computed.
- The app then makes a call to Bob's preferred tagging precompile (the one detailed below) with a request to generate a tag.
  - If Alice has never-before tagged Bob (using this tagging precompile), the tagging contact will establish a handshake shared secret, broadcast it, and store it for future reference.
  - If Alice has already established a handshake shared secret with Bob (in this tagging precompile), then the tagging contract will read that shared secret and use it to compute the next tag in the sequence. The tag will be concatenated with the ciphertext.

6.2 Handshaking

The first time Alice wants to send a ciphertext to Bob:

Inputs: Some identifiers for Alice & Bob. *Which* identifiers is still being debated (see some topics below). Whatever identifiers aren't passed-in as inputs would need to be looked-up from the Key Registry.

It would be most efficient if the tagging contract didn't have to re-read the sender and recipient's master public keys from the Key Registry, and it could instead just accept some 'purported' master public keys passed to it by the app. This imparts some extra requirements on the app:

- 1. If an app wants to make a call to such a tagging contract interface, it would need to contain logic to 'understand' the protocol-specified Grumpkin master keys, and the Key Registry. This makes it trickier for apps to experiment with "key abstraction" ideas.
- 2. The app will need to constrain the relationship between the user addresses and their keys.

We could perhaps offer a second, alternative interface to the tagging contract, that enables an app to instead simply specify the sender and recipient *addresses* (instead of public keys), and the tagging contract then reads the registry.

In the interest of commencing an implementation, let's say the inputs are:

$[Npkm^A, Ovpkm^A, Ivpkm^B, \text{app\_address} := \texttt{msg\_sender}]$

Then, the tagging precompile does the following computations:

$esk_{hs} \xleftarrow{\$} \mathbb{F}_r$

$Epk_{hs} := esk_{hs} \cdot G \in \mathbb{G}$

Mike: Can we re-use the ephemeral public key that the *app* will have used to derive its ciphertext? E.g. by having the app pass that Epk as another input to this circuit? It would remove a field from being broadcast on-chain.

$S_{hs}^{A \to B} := esk_{hs} \cdot Ivpk_m^B$

Mike: Note: this notation is lazy, in the sense that Alice and/or Bob might rotate their keys. If they rotate their keys, a new shared secret would need to be established. A more-accurate notation would be $S_{hs}^{esk_{hs} \to Ivpk_m^B}$, but that's a mouthful.

$\text{hashed\_secret} := \text{poseidon2}(\text{"az\_fuzzy\_tag"}, S_{hs}^{A \to B})$

$\text{fuzzy\_tag}_{hs} := \text{hashed\_secret}[0:8]$

Mike: Note: this is an optimisation borrowed from EIP-5564: Stealth Addresses. Upon reflection, I don't think this approach works here, because it doesn't give a 100% guarantee that you are the intended recipient (it gives false positives to lots of users). We'd need a more conventional MAC. One approach might be to introduce another "tagging keypair", as I talk about in the big miro diagram - it would be a bit ugly to introduce yet another keypair just for this, though.

`handshakes.slot` $\leftarrow 1$ - assign storage slot 1 to the handshakes "state variable", which we'll loosely model as a mapping `handshakes[from_Ovpkm][to_Ivpkm]`.

`slot` $\leftarrow \text{poseidon2}(\text{poseidon2}(\texttt{handshakes.slot}, Ovpk_m^A), Ivpk_m^B)$

$$\text{note\_hash}_{S^{A\to B}} := \text{poseidon2}(\text{"az\_silo\_note"}, \text{tag\_address}, \text{poseidon2}(\texttt{slot}, \text{poseidon2}(S_{hs}^{A\to B})))$$

where tag_address is the address of the tagging precompile contract. Notice: randomness isn't needed within this note, since the shared secret is already plenty random enough.

Prevent Alice from spamming Bob with more handshakes:

$$S_{hs}^{ovsk_m^A \to Ivpk_m^B} := ovsk_m^A \cdot Ivpk_m^B$$

Mike: We could maybe use nsk_m^A here instead, to avoid the extra constraints of proving correctness of $ovsk_m$.
`slot` $\leftarrow$ *todo*

$$\text{nullifier}^{ovsk_m^A \to Ivpk_m^B} := \text{poseidon2}(\text{"az\_silo\_nullifier"}, \text{tag\_address}, \text{poseidon2}(\texttt{slot}, \text{poseidon2}(S_{hs}^{ovsk_m^A \to Ivpk_m^B}, nsk_m^A)))$$

Mike: Notice: we're using the master nullifier secret key here, as an optimisation to avoid app-siloed nullifier key derivation constraints. We're assuming we can trust a protocol-defined circuit with this information. If we think it's unacceptable for the PXE to inject this master secret key into any circuit, we'll need to instead derive the app-siloed nullifier secret key here.

Mike: This "spam prevention" step might be of limited usefulness. Perhaps we should scrap it. If Alice rotates her $ovsk_m$, she can generate a new handshake easily. It will cost money to broadcast handshakes, so maybe that cost is a sufficient spam deterrent?

This also adds constraints. If we're doing this once per note, those constraints will really add up to something horrible.

I guess in addition to preventing spam, it also prevents *accidental* duplication of handshakes. E.g. if Alice is generating a later tx to Bob using a different device that hasn't-yet synced all Alice's historic outgoing state, her PXE on the second device would claim that Alice hasn't handshaked with Bob yet. This check would prevent Alice from transacting that second time until she's brute-force located her previous handshake with Bob. Maybe that's a good thing, maybe that's a bad thing?
Note, we would also need to be careful reviewing this handshaking protocol: can Alice or Bob 'brick' this handshaking protocol if they change only one of their keys in isolation? E.g. if Alice rotates her $ovsk_m$ and not $nskm$?

On balance, I think we should scrap this "spam prevention" step.

6.2.1 Option A (informing the sender of the handshake)

Mike: There are two options for informing the sender of a handshake. My first attempt was Option A. My now-preferred attempt is Option B (see reasons below) - although the cryptography doesn't seem secure yet, so would like review and new ideas from the Crypto Research team, please!

Compute an outgoing ciphertext, so that the sender (Alice) can re-discover this transaction:

Mike: This copies the same approach as encrypting the outgoing body (see the 'Encrypt' section).

$pt_{\text{hs, out}} := [esk_{hs}, Ivpk_m^B, \text{address}_B]$

$\text{aes\_randomness}_{\text{hs, out}} := \text{sha256}(\text{"az\_sym\_key\_hs\_out"}, S_{hs}^{A \to B})$

Mike: It'd be very good if we could use poseidon2 here instead for all the aes randomness values throughout. It would save thousands of constraints. We could perhaps then re-use hashed_secret as $\text{aes\_randomness}_{\text{hs, out}}$.

$sym_{\text{hs, out}} := \text{aes\_randomness}_{\text{hs, out}}[0 : 128]$

$iv_{\text{hs, out}} := \text{aes\_randomness}_{\text{hs, out}}[128 : 128 + 96]$

$ct_{\text{hs, out}} := \text{AES128\_enc}(pt_{\text{hs, out}}; iv_{\text{hs, out}}, sym_{\text{hs, out}})$

Emit a so-called "unencrypted log" containing:

$[\text{tag\_address}, Epk_{hs}, \text{plus some kind of "MAC"}, ct_{\text{hs, out}}]$

along with:

$[\text{note\_hash}_{S^{A \to B}}]$

Mike: Note: "unencrypted log" is a confusing name in this context. This kind of log was originally designed to emit publicly-visible data from a publicly-visible contract address. Here, we want to make use of the property that the contract address emitting the log is publicly-visible, so that users can filter for logs solely from this contract, in order to efficiently brute-force handshakes. The contents of the log – in this case – are very much encrypted, as we've just shown.

6.2.2 Option B (informing the sender of the handshake)

Mike: There are two options for informing the sender of a handshake. My first attempt was Option A. My now-preferred attempt is Option B (see reasons below) - although the cryptography doesn't seem secure yet, so would like review and new ideas from the Crypto Research team, please!

All the AES encryption and hashing in Option A is getting expensive (constraint-wise). Is there something 'lighter' that we can do, to notify Alice about the handshakes she's establishing with Bob?

Maybe Alice could emit her own shared secret from the ephemeral keypair, and use it to derive $sharedsecreths$. This is all very speculative. E.g.:

$S^A_{hs} := esk_{hs} \cdot Ovpk^A_m$

$S_\Delta := S^{A\to B}_{hs} - S^A_{hs}$

Then the data to broadcast would be:

[tag_address, Epk_{hs}, plus some kind of "MAC", S_Δ]

along with:

[note_hash$_{S^{A\to B}}$]

The sender would then learn $S^{A\to B}_{hs}$ by computing $ovsk^A_m \cdot Epk_{hs} + S_\Delta = S^A_{hs} + S_\Delta = S^{A\to B}_{hs}$.

Mike: **Is that safe?** It feels dangerous to broadcast this S_Δ perhaps? Is there a way for an observer to reverse-derive any of the important secrets $S^{A\to B}_{hs}$, $Ivpk^B_m$, $Ovpk^A_m$? It seems an attacker would need either $ovsk^A_m$ or $ivsk^B_m$ to do that.

The reason it might be acceptable (with Option B) to omit details of *who* the sender has established a handshake with, is if the calling app is already conveying this information via the app's own outgoing ciphertext. **Mike**: I'm not sure if this is reasonable or not? It imposes on apps a deeper understanding of the optimisations of this tagging contract, and where its checks 'fall short'.

6.2.3 Spanners

Important spanner: How can the sender (Alice) tag herself? Since we have a vision that multiple tagging options might exist in future, it could be that Alice and Bob will want to use differing tagging schemes in future. The tagging contract that the app has called (in this worked example) is that which

Bob prefers. If Alice prefers a different tagging contract, then this tagging contract would need to also be called, so that Alice can attach her own tags to each ciphertext.

There are some ongoing discussions relating to log layouts (see this hackmd). If we choose the option of rearranging logs in a dedicated circuit, Alice could potentially tag herself once at the start of the transaction, via her account contract. If we choose the simpler option of tagging every ciphertext, the app will need to make two tagging calls: a call to Bob's preferred tagging precompile contract (as explained above) and a call to some "tag myself" function in Alice's preferred tagging precompile contract.

Alternatively, maybe we could simplify things and have one tagging precompile exist at a time, but it's unclear how the transition would happen, and that might result in a bad UX for users. Also, future tagging schemes might have tradeoffs that different users want to choose between.

Another important spanner: Does this tagging contract need to make an authwit call back to Alice's account contract? So far in Aztec, we've made the security assumption that knowledge of someone's nsk_m and $ovsk_m$ is insufficient to spend a users funds (or other kinds of notes): an attacker would also need to know whatever secret(s) are needed to pass the auth checks in the user's account contract. If this tagging precompile *doesn't* do an authwit call back to Alice's account contract to ask "Are you sure you want this contract to be able to create/modify state, owned by you, on your behalf?", then we're kind-of violating that security assumption. Without the authwit call, any app could say "Please handshake/tag between A and B", and the tagging contract would have no guarantee that A's account contract was ever asked about this. It's a tricky question. Making the authwit call would add lots of extra constraints.

What would be the consequences of *not* doing an authwit call?

- If taking the branch of creating a new handshake, an attacker would be able to generate new handshakes between the victim's keys and anyone else. What effect does this have?
  - It might establish connections that the user didn't want to establish.
  - It would make non-repudiation difficult, since the user could deny having performed a transaction, due to a compromise of their PXE keys.
  - As long as generation of the outgoing handshake ciphertext is constrained, the actual user would be able to decrypt and discover the handshaking secrets that the attacker created. This is because the attacker would be forced to use the correct $ovsk_m$ in deriving the outgoing handhsake ciphertext. If creation of this ciphertext is not constrained, the user wouldn't be able to learn the $S_{hs}^{A \to B}$, and so might not be able to send notes to the recipient. If we relax the "spam prevention" step, then the sender would be able to send notes to the recipient, because they would be able to generate a new, replacement handshaking secret

with the recipient. Sending a replacement handshaking secret under these same compromised keys, however, would be silly; a better thing to do would be for the user to rotate their keys and then generate a new handshake.
    - If the user is not tagged (see the section immediately above about how the sender can tag themselves), they would need to brute-force the handshakes to discover what handshakes they've created. This is perhaps acceptable, since a recipient also needs to brute-force the handshakes.
    - It might be confusing for the recipient. And again, it might result in connections that the recipient didn't want to establish.
- If taking the branch of computing a tag from an already-previously-established handshake secret, an attacker would be able to increment the tag counter by nullifying the current counter value.
    - It would make non-repudiation difficult, since the user could deny having performed a transaction, due to a compromise of their PXE keys.
    - Assuming the user was able to learn of the handshake shared secret, they would be able to see these tags being emitted, so it wouldn't cause problems to the user wishing to transact in future. (And actually, the user should rotate their keys before they transact again).
    - It might be confusing for the recipient.

If tags are app-siloed, then the impact of all this is lessened. Good, well-designed apps will already do checks against `msg.sender` or will do authwit calls back to the sender's account contract. If badly designed contracts don't do these checks, and that results in an attacker creating spammy, app-siloed tags, then because they're app-siloed it reduces the confusion that the actual sender and the recipient will feel. It's akin to receiving a spam, worthless airdrop on Ethereum; you just ignore it.

Mike: At the moment we're not doing authwit calls from the tagging contract. Thoughts and criticisms are welcome!

6.3 Tagging

We would ideally like tags to be app-siloed, so that an app can quickly hone in on tags relating only to that app, at the time a user logs in to the app.

With that in mind, we could define a sequence of tags for $i \in \mathbb{Z}$ and for some app_address as:

$$\{tag_i^{\text{app\_address}} := \text{poseidon2}(\text{"az\_tag"}, S_{hs}^{A \to B}, \text{app\_address}, i)\}_{i \in \mathbb{Z}}$$

(Arguments to the hash function are ordered from least specific to most specific).

`tag_indices.slot` $\leftarrow 2$ - assign storage slot 2 to the tag_indices "state variable", which we'll loosely model as a mapping:

`tag_indices[from_Ovpkm][to_Ivpkm][app_address]`.

`slot` $\leftarrow$ poseidon2(poseidon2(poseidon2(`tag_indices.slot`, $Ovpk_m^A$), $Ivpk_m^B$), app_address)

$$\begin{aligned}\text{nullifier}_{tag\_i}^{\text{app\_address}} :=& \text{poseidon2}(\texttt{"az\_silo\_nullifier"}, \text{tag\_address},\\ & \text{poseidon2}(\texttt{slot}, \text{poseidon2}(i, nsk_m^A))\\ &)\end{aligned}$$

Mike: Notice: we're using the master nullifier secret key here, as an optimisation to avoid app-siloed nullifier key derivation constraints. We're assuming we can trust a protocol-defined circuit with this information. If we think it's unacceptable for the PXE to inject this master secret key into any circuit, we'll need to instead derive the app-siloed nullifier secret key.

Since this is a protocol circuit, and if PXE security allows, we could demonstrate nsk_m^A's correctness directly, via $nsk_m^A \cdot G == Npk_m^A$.

Broadcast:

- $tag_i^{\text{app\_address}}$.
- $\text{nullifier}_{tag\_i}^{\text{app\_address}}$

Chicken and Egg: if an app-siloed tag is defined as $tag := hash(S, \text{app\_address}, i)$, then how does the intended recipient (who only knows S and $i = 0$ initially) passively discover state for apps they're not inclined to interact with (e.g. airdrops)? It seems infeasible for a recipient to trial-hash every app_address in existence, to compose a list of potential tags, because they'd have to then repeatedly query a server with all resulting potential tags, which costs too much time and money.

Furthermore, the recipient doesn't know all app addresses (because some app addresses are not public), so they could never derive the tags created for private contract addresses with this method - they'd need to be told about the existence of such contract addresses off-chain. But actually that's always true of private contract addresses: you always need to be actively told about their existence, so maybe this paragraph is moot?

One possible solution would be to create a new handshake for every app_address. But since we're already nervous about the sheer number of handshakes that will be created if each pair of people handshakes only once, it seems completely impractical to blow-up the number of handshakes to be "per pair of people, per app".

6.4 How badly did we just increase proving time and costs?

Possibly badly... let's have a rough look (and I do mean rough, given all the questions above):

- Handshake (Option B):
  - 3 ECMULS
  - 1 EDADD
  - 6 Poseidon2 hashes
  - Broadcast: 4+ fields in a log, and 1 new note_hash.
- Tag:
  - 1 note read request (the handshaking shared secret).
  - 1 nullifier read request (the previous counter)
  - 7 Poseidon2 hashes
  - 1 nsk derivation
  - 1 tag, 1 new nullifier

But remember: if we want the tagging circuit to *conditionally* compute a handshake or a tag (depending on whether a handshaking secret has been established in the past), we'll need to execute at least the constraints of the most constraint-heavy path (minus the read requests, which can be conditionally performed).

Mike: New idea: Store the handshaking shared secret *inside* the tag_index nullifier, to avoid having to perform two reads each time you tag. ALSO, have this nullifier *be* the tag! So at the time you first perform a handshake, create a nullifier containing just the shared secret. When creating subsequent tag nullifiers, include the shared secret inside that nullifier, so that we don't have to also read the original shared secret nullifier. The tag nullifier can be considered to be *the* tag, so we wouldn't have to emit both a tag *and* a nullifier.

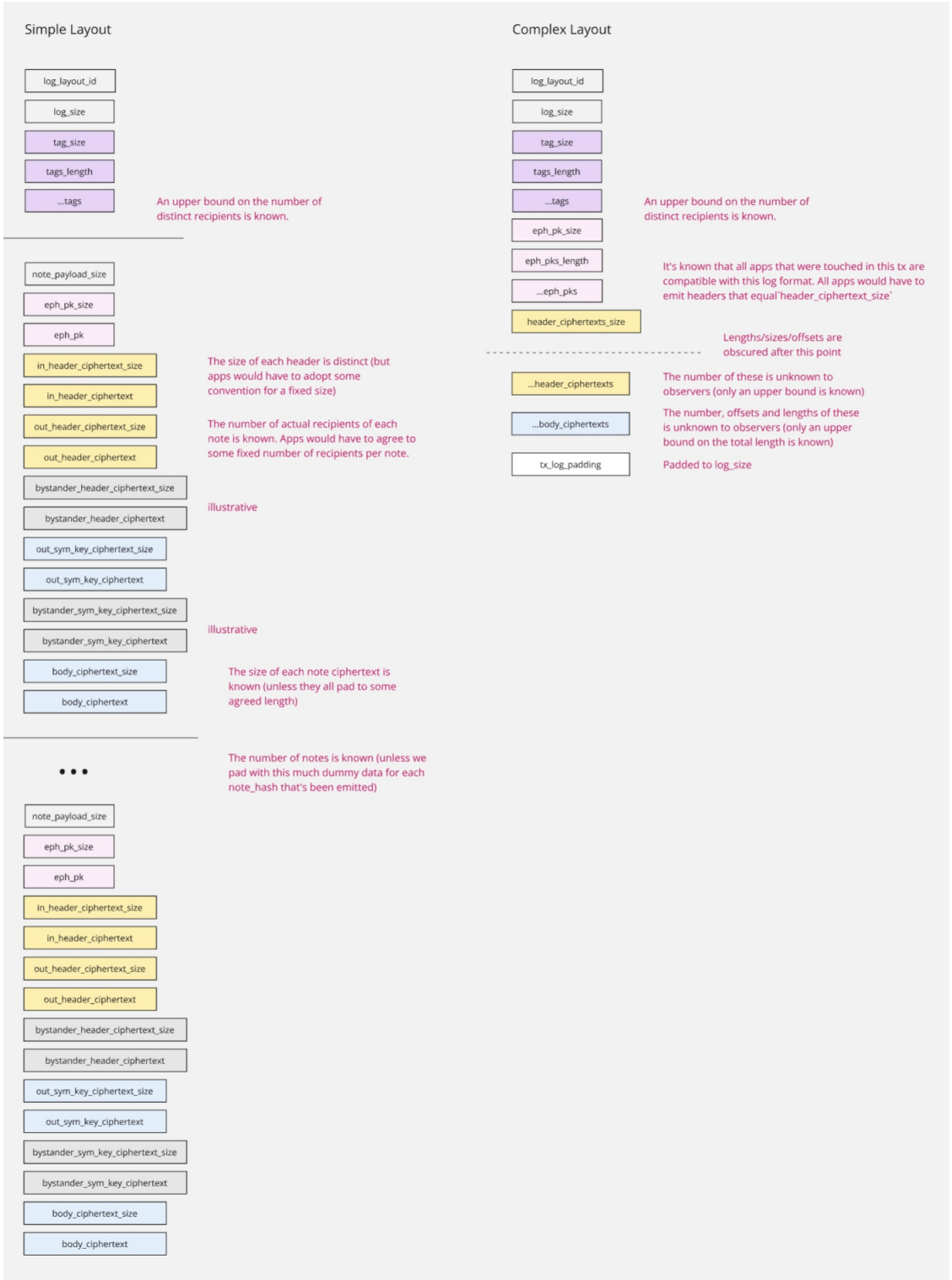
This new idea causes problems elsewhere, though. See the log layouts section: tags are located in a special position in logs, so that the PXE (and big PIR servers) can locate the tags quickly. If instead we said that this tag nullifier *is* the tag, then we've muddled the previously-distinct concepts of a tag and a nullifier. How, then would a user/server be able to identify just the tags? Perhaps we could still have a dedicated location in a log for tags, and we could make some tweaks to the kernel to inject these tags into the nullifier tree? This approach might work for our one tagging scheme, but if in future we introduce more tagging schemes, this might result in a complicated protocol. Furthermore, if we eventually get a VM in private-land, then users' account contracts can contain the tag computation logic. Then we'd need to give apps this ability to push tags to the tags array, and inject them into the nullifier tree..

7 Log Layout

This is a very involved topic, touching on lots of outstanding decisions.

The most comprehensive writings on this so far are in this hackmd.

Here's a pic of two competing approaches, along with a brief table comparing pros & cons:


Simple Layout
log_layout_id
log_size
tag_size
tags_length
...tags
An upper bound on the number of distinct recipients is known.
note_payload_size
eph_pk_size
eph_pk
in_header_ciphertext_size
in_header_ciphertext
The size of each header is distinct (but apps would have to adopt some convention for a fixed size)
out_header_ciphertext_size
out_header_ciphertext
The number of actual recipients of each note is known. Apps would have to agree to some fixed number of recipients per note.
bystander_header_ciphertext_size
bystander_header_ciphertext
illustrative
out_sym_key_ciphertext_size
out_sym_key_ciphertext
bystander_sym_key_ciphertext_size
bystander_sym_key_ciphertext
illustrative
body_ciphertext_size
body_ciphertext
The size of each note ciphertext is known (unless they all pad to some agreed length)
• • •
The number of notes is known (unless we pad with this much dummy data for each note_hash that's been emitted)
note_payload_size
eph_pk_size
eph_pk
in_header_ciphertext_size
in_header_ciphertext
out_header_ciphertext_size
out_header_ciphertext
bystander_header_ciphertext_size
bystander_header_ciphertext
out_sym_key_ciphertext_size
out_sym_key_ciphertext
bystander_sym_key_ciphertext_size
bystander_sym_key_ciphertext
body_ciphertext_size
body_ciphertext
Complex Layout
log_layout_id
log_size
tag_size
tags_length
...tags
An upper bound on the number of distinct recipients is known.
eph_pk_size
eph_pks_length
...eph_pks
It's known that all apps that were touched in this tx are compatible with this log format. All apps would have to emit headers that equal`header_ciphertext_size`
header_ciphertexts_size
Lengths/sizes/offsets are obscured after this point
...header_ciphertexts
The number of these is unknown to observers (only an upper bound is known)
...body_ciphertexts
The number, offsets and lengths of these is unknown to observers (only an upper bound on the total length is known)
tx_log_padding
Padded to log_size

Comparison

	Simple Layout	Complex Layout
Flexibility up to some upper bound		• Num distinct recipients in the tx • Num recipients for each note • Num ciphertext headers • Num actual notes • Note sizes
Social constraints (Flexible, but not really, and should actually be socially constrained)	Apps would have to adopt these conventions: • all headers must be some fixed size. • all notes must be "chunked"/padded to some fixed size. • all notes must have 2 recipients Apps which don't adopt this format would spoil privacy for everyone. Also: • Dummy data must be emitted for every note_hash, to obscure the number of actual notes. Otherwise, padding the number of note_hashes is pointless. Where would this happen? Account contract?	• Apps would need to agree to adopt this format. Sure, apps can opt out of having their logs be routed via the 'reformatting circuit', but that spoils privacy for everyone.
Constraints		All apps which use this format _must_: • conform to emitting data to the data bus, categorised under "eph_pk", "header", "body". • emit headers that are the prescribed, fixed size (say, 40 bytes).

A Non-hardened key derivation

We've concluded that non-hardened keys aren't useful in Aztec. For posterity, we explain non-hardened key derivation (without rigour or review) here.

$i := \text{poseidon2}(\text{"}az\_ivpk\_app\_non\_hardened\text{"}, \text{app\_address}, Ivpk_m) \in \mathbb{F}_r$

Mike: Note: it's not decided how $Ivpk_m$ should be serialised as $\mathbb{F}_r$ values when passed to this hash function. Either as an x-coord and a y-coord; or as an x-coord and a sign bit. There will be no difference in constraints, given how poseidon2 works.

$Ivpk_{app}^{\text{non-hardened}} := i \cdot G + Ivpk_m \in \mathbb{G}$

Khashayar: suggested in the Miro diagram: we don't think we need to add $Ivpk_m$ as an argument to the hash. The value of i can be the same across users

for the same app. This shouldn't cause an issue as i is a public value anyways.

Mike: Interesting observation. If that's true, we could perhaps remove i altogether and define $Ivpk_{app}^{\text{non-hardened}} := \text{app\_address} \cdot G + Ivpk_m$. But a concern I have with removing $Ivpk_m$ from the derivation of i, or removing i altogether, is that it's easier for a user to then cleverly choose strange and potentially-malicious values for $ivsk_m$.
E.g. in the case of defining $i := \text{poseidon2}(\text{"}az\_ivpk\_app\text{"}, \text{app\_address})$, one could choose $ivskm \leftarrow -i$, which seems bad, to me, as it results in $Ivpk_{app}^{\text{non-hardened}} = O$.

The recipient can derive the same key as:

$$ivsk_{app}^{\text{non-hardened}} := i + ivsk_m \mod r \in \mathbb{F}_r$$

Mike: Notice: the suggestion is to use $\mod r$ instead of using the grumpkin scalar field $\mod q$, just because working with $\mathbb{F}_r$ is more efficient in a snark.

$$Ivpk_{app}^{\text{non-hardened}} = ivsk_{app}^{\text{non-hardened}} \cdot G = (i + ivsk_m) \cdot G = i \cdot G + Ivpk_m$$

Mike: Note that the proper BIP32 spec uses SHAHMAC512 instead of poseidon2 to derive a 512-bit value for i. The left-hand half of their 512-bit i gets used in a similar fashion to our $i \in \mathbb{F}_r$ above, whilst the right-hand half of their 512-bit i is used as a "chain code". I've removed the notion of a "chain code" from this derivation, as even the author of BIP32 later admitted it's unnecessary here.

There's a big problem with non-hardened BIP32-style derivation: if someone learns your $ivsk_{app}^{\text{non-hardened}}$, then they can compute your master secret key as:

$$ivsk_m = ivsk_{app}^{\text{non-hardened}} - i \mod r$$

That's pretty bad. It means a user can't safely share their $ivsk_{app}^{\text{non-hardened}}$ with a 3rd party (to give that 3rd party view access into the notes being sent to the user), without enabling the 3rd party to derive the user's $ivsk_m$.

It also means a user can't use this $ivsk_{app}^{\text{non-hardened}}$ to decrypt within an app contract, because we can't trust the app not to derive the user's $ivsk_m$ and pass it to the dapp's javascript code, or emit it as an event to the world!

Now, one could argue that if a user has read the contract themselves to check for such malicious app behaviour, or if the contract has some "seal of approval" from a reputable auditor, then perhaps the app could safely derive keys in this way. But blockchain users aren't renound for their due diligence.

There is actually a way to avoid divulging $ivsk_{app}^{\text{non-hardened}}$ to a 3rd party or

to an app contract, but its performance might not be fast enough for some use cases. The user could prove (not a zk-snark; a schnorr-protocol-type proof) correct derivation of a symmetric key - given some ciphertext and ephemeral public key - without divulging $ivsk_{app}^{\text{non-hardened}}$. But then, we argue that this approach can just be used with master keys, and so non-hardened app-siloed keys remain unnecessary.