Seeking thoughts and comments, please

To ‘read’ a note is to*:

- open it within an app circuit
- prove the note’s existence in the tree

This post asks the question of whether app circuits should contain the logic to “prove the note’s existence in the tree”, or whether that logic should be ‘punted’ to the private kernel circuit.

Thanks to Zac, Joe, Charlie for chatting about this from time to time.

Pros (of deferring the membership proof to the kernel circuit)

- It doesn’t add any extra constraints for the user: they generate both the app proof and the private kernel proof.
- It would allow the decentralised community to ‘hard-fork’ more easily, without breaking app circuits.
- Examples of such breaking hard-forks:
- A change in the depth of the private data tree
- A change in the hash type used in the private data tree (e.g. to increase security of the hash)
- Unforeseen bug fixes relating to the private data tree
- A change in the depth of the private data tree
- A change in the hash type used in the private data tree (e.g. to increase security of the hash)
- Unforeseen bug fixes relating to the private data tree
- If membership proofs were done in app circuits, they would be broken by such hard forks (the app wouldn’t be usable anymore). But the community will be able to ‘swap out’ the verification keys of kernel circuits for newer kernel circuits (via a hard fork), and so the kernel circuit logic can be updated to use a different hash type or tree depth.
- Examples of such breaking hard-forks:
- A change in the depth of the private data tree
- A change in the hash type used in the private data tree (e.g. to increase security of the hash)
- Unforeseen bug fixes relating to the private data tree
- A change in the depth of the private data tree
- A change in the hash type used in the private data tree (e.g. to increase security of the hash)
- Unforeseen bug fixes relating to the private data tree
- If membership proofs were done in app circuits, they would be broken by such hard forks (the app wouldn’t be usable anymore). But the community will be able to ‘swap out’ the verification keys of kernel circuits for newer kernel circuits (via a hard fork), and so the kernel circuit logic can be updated to use a different hash type or tree depth.
- It makes Noir Contract logic marginally simpler (although it’s actually quite easy to perform a membership check with Noir, so this point is moot; all we need for A3 compatibility is the ability to specify the Pedersen domain separator, which is in the works).

Cons (of deferring the membership proof to the kernel circuit)

- It imposes an upper bound on the number of note ‘reads’ which can be performed by an app circuit.
- Currently (with membership checks being done within the app circuit), an app circuit can perform practically unlimited** reads, because all we need to validate those reads is a single field in the app circuit Public Inputs ABI: the old_private_data_tree_root

field.

- If reads are deferred to the kernel circuit, the app circuit needs to now communicate a list of all

of the note commitments which have been ‘opened’ within the app circuit, so that the kernel circuit can ensure to prove existence of each of these note commitments in the tree. Let's call such a list read_requests

(a list of reads that the app circuit is requesting be performed). * Given that the kernel circuit must have a static number of inputs, which can accommodate many/most app circuits, we'll need to fix the size of the read_requests

array to some READ_REQUESTS_LENGTH

constant. This is why the number of reads is restricted under this approach.

- Given that the kernel circuit must have a static number of inputs, which can accommodate many/most app circuits, we'll need to fix the size of the read_requests

array to some READ_REQUESTS_LENGTH

constant. This is why the number of reads is restricted under this approach.

- Having said that, it will be possible to generate many permutations of kernel circuits, with varying lengths of input arrays. So we could have a kernel circuit for 8, 16, 32, 64, 128, 256, etc read_requests

. We need to do more thinking on this subject though. Given the number of arrays the kernel circuit handles, if we wanted to permute all of them, the number of kernel circuit permutations would be huge, and quite a tricky engineering feat to generate such permutations and ensure they're secure.

- Currently (with membership checks being done within the app circuit), an app circuit can perform practically unlimited** reads, because all we need to validate those reads is a single field in the app circuit Public Inputs ABI: the old_private_data_tree_root

field.

- If reads are deferred to the kernel circuit, the app circuit needs to now communicate a list of all

of the note commitments which have been 'opened' within the app circuit, so that the kernel circuit can ensure to prove existence of each of these note commitments in the tree. Let's call such a list read_requests

(a list of reads that the app circuit is requesting be performed). * Given that the kernel circuit must have a static number of inputs, which can accommodate many/most app circuits, we'll need to fix the size of the read_requests

array to some READ_REQUESTS_LENGTH

constant. This is why the number of reads is restricted under this approach.

- Given that the kernel circuit must have a static number of inputs, which can accommodate many/most app circuits, we'll need to fix the size of the read_requests

array to some READ_REQUESTS_LENGTH

constant. This is why the number of reads is restricted under this approach.

- Having said that, it will be possible to generate many permutations of kernel circuits, with varying lengths of input arrays. So we could have a kernel circuit for 8, 16, 32, 64, 128, 256, etc read_requests

. We need to do more thinking on this subject though. Given the number of arrays the kernel circuit handles, if we wanted to permute all of them, the number of kernel circuit permutations would be huge, and quite a tricky engineering feat to generate such permutations and ensure they're secure.

- We might be inadvertently eliminating interesting applications which need to perform a huge

number of reads within a function. [Help wanted, to identify such applications!]

Hybrid?

There is a "best of both worlds" hybrid solution, where an app developer can choose

whether to perform membership checks in their app circuit (against the old_private_data_tree_root

) or to communicate a list of read_requests

to the kernel circuit. This would mean an app developer would need to make a choice about how 'future proof' they want their app to be. I can't imagine many devs would want to opt for an app which only works until the next hard fork!!!

Thoughts?

Thoughts?

- Also, for non-const notes, a read will often be accompanied by emitting the note's nullifier, as a way of proving it's not-yet been nullified… but Noir Contract syntax currently separates the syntax for nullifying (with a remove()

method), to give devs more control.

** the number of reads an app circuit can do is currently restricted by the size of the SRS, and by the RAM limits of typical user hardware, and RAM limits imposed by wasm (when proving in the browser).