Thanks to Joe for coming up with this concept. Related doc from Joe: https://hackmd.io/c1L9SP7ORgyknjHBaKTy2A

Summary

We need a way of reading a note from the append-only tree before

it has been inserted into the tree by the sequencer.

This discussion focusses on being able to read such 'pending' notes in subsequent iterations of the kernel circuit. I.e. if a later nested function call wishes to read a note which was created as a new_commitment

in a previous function call (in some previous iteration of the kernel circuit).

This will allow use cases where:

- A private function calls itself recursively
- contract A calls contract B which calls contract A

Important Note

There's a related post here on whether (and how) to remove merkle membership checks from an app circuit.

If we adopt the suggestion in that post (of deferring 'reads' to the kernel circuit), it makes the interfaces much cleaner when trying to also solve this problem.

And so… in everything that follows, I'm assuming we'll perform all merkle membership checks in the kernel circuit, by passing read_request

data to the kernel circuit from the app circuit.

Updating the PublicInputs struct for an app's private circuit:

The newly required field is shown with // NEW

type PrivateCircuitPublicInputs = { call_context: CallContext,

args: Array<fr, ARGS_LENGTH>,
return_values: Array<fr, RETURN_VALUES_LENGTH>,

emitted_events: Array<fr, EMITTED_EVENTS_LENGTH>,

// NEW read_requests: Array,

new_commitments: Array<fr, NEW_COMMITMENTS_LENGTH>,
new_nullifiers: Array<fr, NEW_NULLIFIERS_LENGTH>,

private_call_stack: Array<fr, PRIVATE_CALL_STACK_LENGTH>,
public_call_stack: Array<fr, PUBLIC_CALL_STACK_LENGTH>,
l1_msg_stack: Array<fr, L!_MSG_STACK_LENGTH>,

historic_private_data_tree_root: fr,

contract_deployment_data: ContractDeploymentData,

}

We add a read_requests

field, which is an array of commitments which have been opened (read) within the app circuit. Noir would push any commitments which have been opened to this array.

Updating the Private Kernel's Inputs

and PublicInputs

structs:

Note: the PrivateKernelPublicInputs

don't need to be modified, because they already have a constant field for the old private data tree root.

The PrivateKernelInputs

has a new field called read_request_membership_witnesses

: an array of membership witnesses (a leaf_index, sibling_path tuple) – one for each read_request

in the app circuit's public inputs.

type PrivateKernelInputs = { signed_tx_request: SignedTxRequest;

previous_kernel: PreviousKernelData,

private_call: PrivateCallData, // CHANGE

}

type PreviousKernelData = { public_inputs: PrivateKernelPublicInputs, proof: Proof, vk: VK, vk_index: uint32, vk_sibling_path: Array, }

type PrivateCallData = { call_stack_item: PrivateCallStackItem, // CHANGE (see app circuit PublicInputs above)

private_call_stack_preimages: Array<PrivateCallStackItem, PRIVATE_CALL_STACK_LENGTH>,

proof: Proof,
vk: VK,

function_leaf_membership_witness: MembershipWitness<FUNCTION_TREE_HEIGHT>,
contract_leaf_membership_witness: MembershipWitness<CONTRACT_TREE_HEIGHT>,

// NEW read_request_membership_witnesses: Array, READ_REQUEST_LENGTH>

portal_contract_address: eth_address,

}

Changes to Noir circuit logic

The get()

method will no-longer try to do a merkle membership check. It will also no-longer populate the historic_private_data_tree_root

(because it's not doing a membership check anymore!).

SPIKE: do we even need this historic_private_data_tree_root

field anymore in the app circuit? Are there any use cases which need to read from the tree, without passing a read request to the kernel circuit?

Kernel circuit logic

The Kernel circuit will need to loop through each read_request

, and use the corresponding read_request_membership_witness

to compute the root

of the private data tree. This root

must be the same for every iteration of this loop. The root

can then be set equal to the private_kernel_public_inputs.constant_data.old_tree_roots.private_data_tree_root

.

The crux of this entire post

is that once we have all of the above in place, the kernel circuit can perform a conditional check:

- If the read_request_membership_witness

is identified to be nonsense (we could, for example identify an all-zero sibling path, or a leaf index which is -1

), we know this is an “optimistic” read request: * Loop back through all new_commitments

in private_kernel_inputs.previous_kernel.public_inputs.end.new_commitments

. * If the read_request

value (which is itself a commitment) matches one of those new_commitments

(which are basically pending insertion), then the read_request

is STILL valid. * So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- If the read_request

value (which is itself a commitment) matches one of those new_commitments

(which are basically pending insertion), then the read_request

is STILL valid. * So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- Loop back through all new_commitments

in private_kernel_inputs.previous_kernel.public_inputs.end.new_commitments

. * If the read_request

value (which is itself a commitment) matches one of those new_commitments

(which are basically pending insertion), then the read_request

is STILL valid. * So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- If the read_request

value (which is itself a commitment) matches one of those new_commitments

(which are basically pending insertion), then the read_request

is STILL valid. * So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- So ignore the root

which was calculated for that read_request

.

- Delete the new_commitment

from the new_commitments

array (TODO: think of how to do this efficiently).

- Delete the nullifier

which corresponds to this read_request

(because we just deleted the new_commitment that this nullifier nullifies!)*

- Note: we would only know which nullifier corresponds to a particular read_request

, if they occupy the same index in their respective arrays. Now, not every read will be accompanied by a nullifier (e.g. when reading a const note), so what do we do in those cases? We could have a zero nullifier entry in the nullifiers array, but that messes with our current in-circuit array push/pop logic (which views 0

as “end of array”).

Maybe (and it pains me to suggest this, because it’s ugly), we need another array (in the app circuit Public Inputs ABI) of the same length as new_nullifiers

, which contains index pointers to the read_requests

array. I.e. a way of saying “this nullifier entry corresponds to this read request entry”.

Aztec RPC Client logic

The Simulator no-longer needs to fetch membership witnesses for get

oracle calls.

For a particular function’s public inputs, the Client now needs to read the read_requests

and fetch membership witnesses from the DB. It can identify whether a read_request

is “optimistic” (and hence that the membership witness can be dummy) by looping through the new_commitments

of all earlier function calls in this tx.