Discussion on un-abstracting part of fee payments, by moving the act of actually paying the fee from an app circuit to a protocol circuit. This allows us to enforce fee payments in purely private txs, and lets us reduce the number of public function calls made related to fee payments.

# How do fee payments work today?

- Every tx has optional setup, app, and teardown phases.
- Setup has private and public logic, teardown only public.
- Reverts in setup or teardown render the entire tx invalid (meaning it cannot be included in the block and the sequencer cannot get paid for any work it did related to the tx).
- Sequencers whitelist public setup and teardown function calls that they know will work when paying fees.
- Fees can only be paid on a native untransferrable fee-payment asset ($FPA) that lives at the app layer, which has a distinguished pay_fee

function that needs to be called to pay the tx fee.

- Fee payment occurs in teardown by calling pay_fee

, and needs to be tracked by the AVM and public kernel by identifying the call to the pay_fee

function.

# Can we simplify things?

The current approach means every tx needs to execute at least one public function (teardown for fee payment), which require expensive AVM proofs (along with their corresponding public kernel proofs) just to manage fees. It'd be nice if we could remove reliance on public functions due to their high overhead.

At the very core, what we need to do is:

- Escrow enough funds during setup
- Pay to the sequencer and send rebate to the user in teardown

Given we have an enshrined function in an enshrined asset for fee payments, instead of requiring application code to explicitly call the pay_fee

, we can just collect the fee automatically by the protocol. This solves the second item above. As for the first one, the only thing we need is for an address to appoint themselves as the fee-payer for the transaction, which could happen in private-land via an application circuit output.

## Using an FPC

Let's use a regular fee-paying contract (FPC, aka Paymaster) scenario as an example. The user sends some asset to the fpc, who escrows the fee-paying asset $FPA, and then in teardown pays the sequencer via a call to the fpa and refunds the user the extra gas.

### Paying with a private token

If we want to trigger the escrowing from private-land, it means we cannot use public token balance to pay for gas. We use a private note that we break and pay the FPC, who in turn agrees to appoint itself as the fee-payer via a signal to the kernel.

The resulting L2Tx object submitted by a user would then have a new fee_payer

field. A tx would then be deemed valid only if the fee_payer

address has enough $FPA in balance to pay for the overall gas limit at max gas fees. Once the tx has been executed, the sequencer takes the computed fee_payment

out of the fee_payer

balance, which gets enforced by the base rollup circuit.

We could still have an optional public teardown function, as we do today, that is called by the sequencer after the total

fee_payment

has been computed. This teardown function could be enqueued by the FPC, and use it to send the user back their rebate using partial notes.

Note that the difference with the current approach is that the actual fee payment to the sequencer happens in the protocol and NOT in the teardown function in app-land. This removes the need to track if pay_fee

was called, and makes teardown optional.

We can take this one step further and make teardown revertible

(meaning a revert in teardown doesn't render the transaction invalid). This means sequencers no longer need to whitelist teardown, since they get paid anyway, removing a major responsibility for sequencers. However, as Phil pointed out, making teardown revertible can be a risk for the FPC. If teardown reverts and the FPC was relying on it to complete the partial notes and get paid, they could end up sponsoring the tx without getting paid for it. To solve this, it's important that it's the FPC itself who enqueues the call to the teardown function, and that it can check that there's enough teardown-gas-limit for successfully executing it. In other words, since the sequencer is no longer ensuring that teardown doesn't revert, it's the FPC job to do it now.

### Paying with a public token

Payout and teardown can happen exactly the same as in the private scenario, but here escrowing becomes more difficult. The FPC should not agree to escrow funds for a user unless they can secure their public token balance.

The tx should then have a public setup phase that transfers the public token balance from the user to the FPC, as it does today. Nevertheless, the escrowing of the $FPA would still be signalled from private land.

Note that, in this approach we still need the sequencer to whitelist the public setup function, ie the token.transfer

that moves the user-asset from the user to the FPC. Setup needs to be non-revertible (ie make the tx invalid if it fails) or otherwise the FPC has no means to ensure the user has funds for paying them.

## Native fee payment

Native fee payment is where we see the most benefits: the user sending the tx sets the fee_payer

as themselves. There is no public function execution involved, no setup or teardown phases, and no whitelisting needed from the sequencer.

# Nailing down the specs

Implementing the above would require the following:

- Add a new boolean field set_as_fee_payer

in the AppCircuitPublicInputs

, so application circuts can signal to the kernel their willingness to pay for the tx with their $FPA balance.

- Add a new address field fee_payer

to the PrivateKernelPublicInputs

, which gets set by the privat kernel to the current address when set_as_fee_payer

is true. The kernel should fail if more than one app circuit signals set_as_fee_payer

in the same tx.

- Add a new validity condition for a tx checked by the sequencer, where the tx is only valid if the fee_payer

$FPA balance is greater or equal to the sum of max_gas_price * gas_limit

on every dimension (l1, l2, da).

- Outputting a new public_data_update

to the endNonRevertibleData

of the tx in the base rollup circuit, that decreases the $FPA balance for the fee_payer

by the metered fee_payment

. * We do the accounting in the base rollup circuit since it is the first circuit that consistently runs after all app logic is done. Doing this in the public kernel tail would not work, since it doesn't run if the tx doesn't have any enqueued public calls.

- This requires a protocol circuit to have knowledge of how to update a public state value for a given contract. If we didn't like this, we could move the $FPA implementation itself to the protocol layer, and store its balances in a dedicated tree (like in Ethereum), but I don't like this approach.
- We do the accounting in the base rollup circuit since it is the first circuit that consistently runs after all app logic is done. Doing this in the public kernel tail would not work, since it doesn't run if the tx doesn't have any enqueued public calls.
- This requires a protocol circuit to have knowledge of how to update a public state value for a given contract. If we didn't like this, we could move the $FPA implementation itself to the protocol layer, and store its balances in a dedicated tree (like in Ethereum), but I don't like this approach.

# Questions

- Is the FPA transferrable?

If the FPA can be transferred in public app logic, then the sequencer must escrow the max payable fee before executing any public app functions. Otherwise, it's enough for the sequencer to check the funds are available in the beginning of the tx, and charge the exact amount at the end, much like Ethereum does.

- Should teardown be made revertible?

As mentioned above, making teardown revertible

removes the responsibility of whitelisting teardown public function calls from the sequencer. The sequencer currently needs to keep a list of valid FPC teardown calls, which makes it more difficult for new FPCs to be adopted. However, a revertible teardown means that any FPC must ensure there's enough teardown gas allocated to account for moving whatever token it's dealing with, which could change from one token to another.