Public state is great, it allows us to work on a synced global state.

If we need to perform operations on a globally available state, to which we don't have access on private-land, we can just call a public function that for example increases the total_users

counter.

doing this in private-land would require a contract to have a "leaked" Note which users would need to orderly nullify and update, forcing them to have reverted transactions if another user already has an in-transit tx that created the same nullifier.

they can also create individual "leaked" Notes that would later need to be aggregated. which is easier said than done.

So the solution is pretty easy right? Just use Public state for these type of scenarios.

Not so fast. What about privacy?

Using Homomorphic Encryption addition we can add integers to a public variable without revealing it's value.

This enables users to add private values to a public encrypted state.

This would work as follows:

- a new value is created using a random number as the starting point.
- this is saved to then subtract it from the final value.
- this is saved to then subtract it from the final value.
- users will generate proofs and submit transactions as usual
- internally it will encrypt the user value and call public_add(encrypted_value)

, this is encrypted for a specific public_key previously registered.

- on the sequencer, the public_add(encrypted_value)

will be triggered and the value will get updated.

- this can be done without the need for the users to know, at proof generation time, the value of the public state.
- internally it will encrypt the user value and call public_add(encrypted_value)

, this is encrypted for a specific public_key previously registered.

- on the sequencer, the public_add(encrypted_value)

will be triggered and the value will get updated.

- this can be done without the need for the users to know, at proof generation time, the value of the public state.
- finally, anyone with access to the secret_key of the registered public_key can call a private function that will:
- privately re-encrypt the values (original & updated) with the secret key.
- obtain final value by doing updated - original

, and do whatever computation privately with this value.

- then call public_check(encrypted_original, encrypted_updated)

to validate the TX against the values stored on the public state * this could fail if in-between proof-generation and execution the public value changes.

- this could fail if in-between proof-generation and execution the public value changes.
- privately re-encrypt the values (original & updated) with the secret key.
- obtain final value by doing updated - original

, and do whatever computation privately with this value.

- then call public_check(encrypted_original, encrypted_updated)

to validate the TX against the values stored on the public state * this could fail if in-between proof-generation and execution

the public value changes.

- this could fail if in-between proof-generation and execution the public value changes.

This is a great cryptographic tool to have as we dive into more complex designs on Aztec.

As always, we love to get your feedback on this, and hope is useful!

We'll also be working on a PoC on this topic, so if you are interested please reach out or comment below

Note: If the encrypted_value

is visible on the blockchain, and the range of possible encrypted values is not large, it's possible for a computationally-capable adversary to bruteforce the preimage by hashing values with the public_key

and matching the output against the encrypted_value

.

There solution to this, is to add a random_value

alongside the encrypted_value

which would add into the original seeded value.

PS: thanks to @joshc , @Maddiaa and @spalladino for the great insight into the zk magic required for this.