Note: This post entirely about singleton

UTXOs (not to be confused with a Set of UTXOs).

Note: The syntax in this post builds on the syntax proposals in my other “UTXO Syntax” posts. Consider reading those ones first, please!

Modifying a singleton UTXO feels pretty simple, behind the scenes:

- ‘Read’: Prove existence of the UTXO’s commitment in the tree
- Compute a nullifier for it.
- Do some computation on the UTXO’s value (modification).
- Store the new value in a new commitment.
- ‘Write’: Add that commitment to the circuit’s new_commitments

field (based on the private circuit public inputs ABI).

Or to put it more simply:

- grab old one
- nullify
- insert new one

BUT…

How do we distinguish between initialising

my_utxo

vs modifying

a previously-initialised my_utxo

?

To initialise, we can’t do the first two steps (grab old one; nullify), because there’s no UTXO in the tree yet for us to grab and nullify.

Well… Zac’s UTXO syntax (with a few additions from other posts) allows us to clearly separate our initialisation from modification, in two distinct functions:

Example code snippet:

UTXO my_utxo; address my_utxo_owner;

private function initialise_my_utxo(uint new_value) { require(msg.sender == my_utxo_owner);

```
my_utxo.insert(new_value, { owner: my_utxo_owner });
```

}

private function modify_my_utxo(uint new_value) { require(msg.sender == my_utxo_owner);

```
// Get & remove.
my_utxo.get({ owner: my_utxo_owner }).remove();

my_utxo.insert(new_value, { owner: my_utxo_owner });
```

}

BUT…

How do we prevent the owner of this UTXO from calling initialise_my_utxo()

multiple times, and hence creating multiple

, non-nullified, versions of the my_utxo

state!!!

I.e.:

- Initialisation 1:
- Emit dummy_nullifier_1

: it looks random to observers.

- Emit commitment_1 = h(storage_slot, value_1, owner, salt_1, nonce=dummy_nullifier_1)

- It’s essential that only one

(non-nullified) version of this commitment ever exists.

- Emit dummy_nullifier_1

: it looks random to observers.

- Emit commitment_1 = h(storage_slot, value_1, owner, salt_1, nonce=dummy_nullifier_1)

- It’s essential that only one

(non-nullified) version of this commitment ever exists.

- Initialisation 2:
- Emit dummy_nullifier_2

: it looks random to observers.

- Emit commitment_2 = h(storage_slot, value_2, owner, salt_2, nonce=dummy_nullifier_2)

- Emit dummy_nullifier_2

: it looks random to observers.

- Emit commitment_2 = h(storage_slot, value_2, owner, salt_2, nonce=dummy_nullifier_2)

Poop. Two versions of this state are now floating around.

So how do we prevent multiple versions of the state from coming into existence, via multiple ‘initialisations’?

Solution 1:

(Not a good solution).

Only allow initialisations within a constructor

function, which by definition can only be called once.

Problems:

- All private singleton UTXOs would need to be created and assigned owners at contract creation.
- That’s infeasible if we have a more complex state variable such as:mapping(address => UTXO);

… because there are ~2^254 such variables; so they can’t all be initialised!

- That’s infeasible if we have a more complex state variable such as:mapping(address => UTXO);

… because there are ~2^254 such variables; so they can’t all be initialised!

NOTE: there’s a separate problem to discuss: currently the syntax does not convey who

the owner of a singleton UTXO<>

should be! We’ll discuss new syntax in a section further below.

Solution 2:

(Inspired by a suggestion made by Zac in a chat a few weeks ago)

Track an initialisation flag:

Solution 2a - public init flag

First let’s consider tracking a public

initialisation flag:

UTXO my_utxo; address my_utxo_owner; bool is_my_utxo_initialised; // public state defaults to falsey.

private function initialise_my_utxo(uint new_value) { require(msg.sender == my_utxo_owner);

```
my_utxo.insert(new_value, { owner: my_utxo_owner });
toggle_init_flag(); // Need to make a public call, because it's a public state.
```

}

public function toggle_init_flag() { require(is_my_utxo_initialised == false); is_my_utxo_initialised = true; }

// etc...

This works!

Problems:

- The world will see when

my_utxo

is initialised, because its init flag is public, and the flag is toggled in a public function! * This might be a problem for some apps!

- This might be a problem for some apps!

Solution 2b - private init flag

Let’s try making the initialisation flag private:

UTXO my_utxo; address my_utxo_owner; UTXO is_my_utxo_initialised; // private to prevent people seeing that // it's been initialised

private function initialise_my_utxo(uint new_value) { require(is_my_utxo_initialised == false); require(msg.sender == my_utxo_owner);

```
my_utxo.insert(new_value, { owner: my_utxo_owner });
is_my_utxo_initialised = true;
```

}

// etc...

Problems:

BUT WAIT… How do we prevent the same re-initialisation problem for this private state: is_my_utxo_initialised

?!?

Poop. We’ve veered towards a cyclic approach.

So approach 2b is no good.

Solution 2c - syntax to allow custom nullifier creation

Let’s try to avoid the problem of 2a – of revealing to the world (via the public function) the state which has been initialised, and the exact smart contract which has been executed.

Perhaps we could emit a unique nullifier

, at the point of initialisation, to prevent the my_utxo

state from ever being initialised again.

This would require syntax which gives control over the preimage of a nullifier.

Ordinarily, a nullifier

is a nullifier of a commitment

, and is derived as:

nullifier = h(commitment, owner_secret_key)

In the initialisation case, we don't have a commitment to nullify. We could

use a 'dummy commitment' in order to create a 'dummy nullifier', but a dummy commitment is ordinarily derived in a way which doesn't ensure uniqueness of that commitment, because the syntax doesn't allow control over the dummy commitment's salt or nonce:

dummy_commitment = h(storage_slot, value, owner, salt, nonce, is_dummy=true)

.

…

Long story short, if we allow direct access to the ACIR++ opcode which pushes nullifiers to the circuit's public inputs (name tbd - "NULL_PUSH" or something), then it's quite simple:

UTXO my_utxo; address my_utxo_owner;

private function initialise_my_utxo(uint new_value) { require(msg.sender == my_utxo_owner);

```
my_utxo.insert(new_value, { owner: my_utxo_owner });

// Create a CUSTOM, unique nullifier, to prevent re-initialisation:
field owner_secret_key = get_secret_key(my_utxo_owner);
field init_nullifier = hash(my_utxo.storage_slot, owner_secret_key);

// Allow direct access to the NULL_PUSH opcode via some function:
new_nullifier(init_nullifier);
```

}

// etc...

Note: we also need access to an ACIR++ opcode which loads a secret key for a particular public key. The secret key gives the nullifier a 'hiding' property, which prevents people from seeing which storage slot has been initialised.

Tentative Conclusion:

- We can initialise singleton UTXOs via a constructor, but that's of limited use.
- If initialising a singleton UTXO in a regular (non-constructor) function, we'll either need:
- A public init flag which can be toggled in a public

function - with the downside that this reveal to the world which state has been initialised.

- Control over calculation of a custom nullifier, and access to the opcode which pushes new nullifiers.
- A public init flag which can be toggled in a public

function - with the downside that this reveal to the world which state has been initialised.

- Control over calculation of a custom nullifier, and access to the opcode which pushes new nullifiers.